



User Guide

User Guide: Open Build Service

Publication Date: 25 Jul 2024

<https://documentation.suse.com> 

Copyright © 2006– 2024 SUSE LLC and contributors. All rights reserved.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or (at your option) version 1.3; with the Invariant Section being this copyright notice and license. A copy of the license version 1.2 is included in the section entitled “GNU Free Documentation License”.

For SUSE trademarks, see <http://www.suse.com/company/legal/> . All other third-party trademarks are the property of their respective owners. Trademark symbols (®, ™ etc.) denote trademarks of SUSE and its affiliates. Asterisks (*) denote third-party trademarks.

All information found in this book has been compiled with utmost attention to detail. However, this does not guarantee complete accuracy. Neither SUSE LLC, its affiliates, the authors nor the translators shall be held liable for possible errors or the consequences thereof.

Contents

About this Guide **xv**

- 1 Available Documentation **xv**
- 2 Feedback **xv**
- 3 Documentation Conventions **xvi**
- 4 Contributing to the Documentation **xvii**

I FIRST STEPS **1**

1 Beginner's Guide **2**

- 1.1 Target Audience **2**
- 1.2 Conceptual Overview **2**
 - Build Recipe **3** • Build Hosts and Packages **3** • Projects and Packages **4**
- 1.3 Requirements for Working with the **osc** Command-Line Tool **5**
- 1.4 Covered Scenarios **5**
- 1.5 Configuring Your System for OBS **6**
- 1.6 Setting Up Your Home Project for the First Time **8**
- 1.7 Creating a New Project **9**
- 1.8 Patching Source Code **13**
- 1.9 Branching a Package **15**
- 1.10 Installing Packages from OBS **18**
- 1.11 Other Useful **osc** Commands **19**

II CONCEPTS 21

2 Supported Build Recipes and Package Formats 22

- 2.1 About Formats 22
- 2.2 RPM: Spec 23
- 2.3 Debian: Dsc 23
- 2.4 Arch: pkg 24
- 2.5 KIWI Appliance 24
- 2.6 SimpleImage 25
- 2.7 ApplImage 25
- 2.8 Flatpak 25
- 2.9 mkosi 28

III SETUP 30

3 osc, the Command Line Tool 31

- 3.1 Installing and Configuring 31
- 3.2 Configuring osc 31
- 3.3 Usage 33
 - Getting Help 33 • Using **osc** for the First Time 33 • Overview of Brief Examples 33

4 Build Configuration 37

- 4.1 About the Build Configuration 37
- 4.2 Configuration File Syntax 38
- 4.3 Building with ccache or sccache 48
- 4.4 Macro Definitions in the Build Configuration 49
 - Macros for the Build Configuration Only 50 • Macros Used in Spec Files Only 50

IV USAGE 51

5 Basic OBS Workflow 52

- 5.1 Setting Up Your Home Project 52
- 5.2 Creating a New Package 53
- 5.3 Investigating the Local Build Process 56
 - Build Log 56 • Local Build Root Directory 57
- 5.4 Adding Dependencies to Your Project 58
 - Adding Dependencies to Your Build Recipes 58 • Associating Other Repositories with Your Repository 59 • Reusing Packages in Your Project 60
- 5.5 Manage Group 61

6 Local Building 62

- 6.1 Generic Local Build Options 62
- 6.2 Advanced Local Build Environment Handling 64

7 Using Source Services 65

- 7.1 About Source Services 65
- 7.2 Modes of Source Services 66
- 7.3 Defining Source Services for Validation 68
- 7.4 Creating Source Service Definitions 69
- 7.5 Removing a Source Service 69
- 7.6 Trigger a service run via a webhook 70
 - Creating a webhook on GitLab 70 • Creating a webhook on GitHub 70

8 SCM/CI Workflow Integration 71

- 8.1 SCM/CI Workflow Integration Setup 71
 - Introduction 71 • Prerequisites 71 • Supported SCMs 71 • Token Authentication 71 • Webhooks 73 • OBS Workflows 77 • Status

	Reporting 89 • Workflow Runs 89 • Errors 92 • Equivalence Table 92
8.2	SCM/CI Workflow Steps Reference Table 93
8.3	SCM/CI Workflow Versions 98 Workflow Version Table 99
8.4	SCM/CI Workflow Integration Use-Cases 99 OBS SCM Service 99 • Test Build a Package For Every Pull Request on GitHub 99 • Rebuild a Package for Every Change in a Branch 101 • Set Flags on a Package to Disable Builds for an Architecture 102 • Create Package on OBS for Every Software Release With Git Tags 103 • Using a Custom Configuration File URL in Combination with Placeholder Variables 105
9	Staging Workflow 107
9.1	Working with Staging Projects 107 Overview of All Staging Projects 107 • Overview of a Single Staging Project 108 • Copy a Staging Project 109
9.2	Working with Requests 109 Assign Requests into a Staging Project 109 • Remove Requests from a Staging Project 109 • List Requests of a Staging Project 110 • Exclude Requests for a Staging Workflow 110 • Bring Back Excluded Requests from a Staging Workflow 110 • Accept Staging Project 110
10	Notifications 112
10.1	Notifications Configuration 112
10.2	Where Can We Find the Notifications? 113
10.3	Notifications Content 114
10.4	Mark Notification as Read or Unread 115
10.5	Notifications Filters 116
10.6	API 118

11 Moderation 120

- 11.1 Code of Conduct 120
- 11.2 Reporting Problematic Content 121
 - Who Can Report? 121 • What Can Be Reported? 121 • How To Report? 122
- 11.3 Acting as a Moderator 123
 - Who Is a Moderator? 124 • How Do Moderators Know About the Reports? 124 • How To Moderate? 124
- 11.4 Reverting Moderator's Actions 126
- 11.5 User Appeal 126
- 11.6 Canned Responses For Moderators 127

V BEST PRACTICES 128

12 Using the OBS Web UI 129

- 12.1 Homepage and Login 129
- 12.2 Home Project 131
 - The Project Page 131 • Changing a project's title and description 132 • Creating Subprojects to a Project 133
- 12.3 My Projects, Server Status 133
- 12.4 Create a link to a package in your home 135
 - Add Link to Existing Package 135 • Package Page, Build Log and Project Monitor Page 137
- 12.5 Repository Output: Built Packages 139
- 12.6 Managing Repositories 140
 - Adding a repository 140 • Add Download on Demand repositories to a project 141 • Adding DoD Repository Sources to a Repository 143 • Editing DoD Repository Sources 146 • Editing DoD Repository Sources 148

12.7

Image Templates 148

Creating Own Image Templates 149 • Publishing Image Templates on the Official Image Templates Page 153

12.8

KIWI Editor 153

Accessing the KIWI Editor 153 • Adding Repositories in the KIWI Editor 157 • Adding Packages in the KIWI Editor 159

12.9

Manage Group 160

12.10

Staging Workflow 161

Creating a Staging Workflow 165 • Start Using Staging Workflow 166 • Delete a Staging Workflow 168 • Configure a Staging Workflow 169 • Staging Project 172 • Working with Requests in Staging Workflow 174

13

Basic Concepts and Work Styles 176

13.1

Setup a project reusing other projects sources 176

13.2

Contributing to External Projects Directly 176

13.3

Contributing to Foreign Projects Indirectly 176

14

How to integrate external SCM sources 177

14.1

How to create a source service 177

Follow upstream branches 177 • Fixed versions 178 • Avoid tar balls 178

15

Publishing Upstream Binaries 180

15.1

Which Instance to Use? 180

Private OBS Instance 180 • openSUSE Build Service 180

15.2

Where to Place Your Project 180

Base Project 181 • Supporting Additional Versions 181

15.3

Creating a Package 182

15.4

Getting Binaries 182

Examples 184

16	Bootstrapping	186
16.1	The Issue	186
16.2	A Cheat Sheet	186
	Creating Your First Project	186 • Importing Your Base Linux Project 186
16.3	Creating a First Project	189
17	osc Example Commands	190
17.1	Package Tracking	190
18	Advanced Project Setups	191
18.1	Rebuilding an Entire Project with Changes	191
18.2	Integrating Source Handling	191
18.3	Using OBS for Automated QA	191
19	Building Kernel Modules	192
20	Common Questions and Solutions	193
20.1	Working with Limited Bandwidth	193
	Using the Web Interface	193 • Using osc with Size Limit 193 • Using download_url 193 • Using Source Services in trylocal Mode 194
VI	REFERENCE	195
21	OBS Architecture	196
21.1	Overview Graph	196
21.2	Communication Flow	198
22	OBS Concepts	201
22.1	Project Organization	201
	Project Metadata	201 • Project Build Configuration 202 • Project Build Macro Configuration 203 • An OBS Package 203
22.2	The OBS Interconnect	203

22.3	Download on Demand Repositories (DoD) 203
	Motivation 203 • XML Document Hierarchy 204 • The Daemon 204 • The download Element 204 • The master Subelement 205 • The pubkey Subelement 205 • Repository Types 205
22.4	Integrating External Source Repositories 208
	Motivation 208 • Creating an Reference to an External SCM 209 • Working with Local Checkouts 210 • Managing Build Recipes in a SCM 210
23	Build Process 211
23.1	Phases of a Build Process 211
	Preinstall Phase 211 • Install Phase 211 • Package Build 212 • After the Build 212
23.2	Identify a build 212
	Read DISTURL from an RPM 213 • Read DISTURL from a container 213
24	Build Containers 214
24.1	Supported Container Formats 214
24.2	Container Registry 215
24.3	Container Image Signatures 216
25	Source Management 218
25.1	Find Package Sources 218
25.2	Generating SLSA Provenance Data 218
25.3	Generating SBOM (Software Bill Of Material) Data 219
26	SCM Bridge 220
26.1	SCM Bridge 220
	Introduction 220 • Setup a package using the scm bridge 220 • Setup an entire project using the SCM bridge 221 • Implementation and Limitations 221 • SCM Source Updates 223

27 Supported Formats 224

- 27.1 Spec File Specials (RPM) 224
- 27.2 OBS Extensions for (KIWI) Appliance Builds 225
- 27.3 OBS Extensions for Dockerfile based builds 226

28 Request And Review System 229

- 28.1 What a request looks like 229
 - Action Types 230 • Request states 231 • Reviewers 232 • Request creation 233 • Request operations 233

29 Image Templates 234

- 29.1 Structure of Image Templates 234
- 29.2 Adding Image Templates to/Removing Image Templates from the Official Image Template Page 234
- 29.3 Receiving Image Templates via Interconnect 234

30 Multiple Build Description File Handling 236

- 30.1 Overview 236
- 30.2 How Multibuild is Defined 236

31 Maintenance Support 238

- 31.1 Simple Project Setup 238
- 31.2 Project setup for the Maintenance Process 239
- 31.3 Using the Maintenance Process 240
 - Workflow A: A Maintainer Builds an Entire Update Incident for Submission 240 • Workflow B: Submitting a Package Without Branching 241 • Workflow C: Process Gets Initiated By the Maintenance Team 242 • Maintenance Incident Processing 242 • Incident Gets Released 243 • Incident Gets Reopened and Re-Released 243 • Using Custom Update IDs 244

31.4	OBS Internal Mechanisms	244
	Maintenance Incident Workflow	244 • Searching for Incidents 246
31.5	Setting Up Projects for a Maintenance Cycle	247
	Defining a Maintenance Space	247 • Maintained Project Setups 247
31.6	Optional Channel Setup	247
	Defining a Channel	247 • Using Channels in Maintenance Workflow 248
32	Binary Package Tracking	249
32.1	Which Binaries Are Tracked?	249
32.2	What Data Is Tracked?	249
	Binary Identifier	249 • Binary Information 250 • Product information 250
32.3	API Search Interface	251
33	Scheduling and Dispatching	252
33.1	Definition of a Build Process	252
33.2	Scheduling Strategies	252
	Build Trigger Setting	253 • Block Mode 253 • Follow Project Links 254
33.3	Release Number Handling	254
	Build Counter Syncing via Architectures	255 • Build Counter Syncing via multiple packages 255
34	Build Constraints	256
34.1	Build Resource Usage and Statistics	256
	Displaying the build statistics	256
34.2	Constraint Qualifiers	257
34.3	Constraint scope and precedence	257
	Project-scoped constraints	258 • Package-scoped constraints 258 • Build recipe-scoped constraints 259
34.4	Constraint syntax	260
	hostlabel	260 • sandbox 261 • linux 261 • hardware 262

- 34.5 Constraint Handling 267
 - At least one compliant worker is available 267 • More than half of existing workers satisfy the constraints 267 • Less than half of existing workers satisfy the constraints 267 • No existing workers satisfy the constraints 268

- 34.6 Checking Constraints with **osc** 268

35 Building Preinstall Images 270

36 Authorization 271

- 36.1 OBS Authorization Methods 271
 - Default Mode 271 • Proxy Mode 271 • LDAP Mode 271
- 36.2 OBS Token Authorization 271
 - Managing User Tokens 272 • Executing an Action 272

37 Quality Assurance(QA) Hooks 274

- 37.1 Source Related Checks 274
- 37.2 Build Time Checks 274
 - In-Package Checks 274 • Post Build Checks 275 • Post Build Root Checks 275 • KIWI Specific Post Build Root Checks 275
- 37.3 Workflow Checks 275
 - Automated Test Cases 275

Glossary 277

A GNU Licenses 288

About this Guide

This guide is part of the Open Build Service documentation. These books are considered to contain only reviewed content, establishing the reference documentation of OBS.

This guide does not focus on a specific OBS version. It is also not a replacement of the documentation inside of the [openSUSE Wiki \(https://en.opensuse.org/Portal:Build_Service\)](https://en.opensuse.org/Portal:Build_Service). However, content from the wiki may be included in these books in a consolidated form.

1 Available Documentation

The following documentation is available for OBS:

Book "Administrator Guide"

This guide offers information about the initial setup and maintenance for running Open Build Service instances.

Book "User Guide"

This guide is intended for users of Open Build Service. The first part describes basic workflows for working with packages on Open Build Service. This includes checking out a package from an upstream project, creating patches, branching a repository, and more. The following parts go into more detail and contain information on backgrounds, setting up your computer for working with OBS, and usage scenarios. The *Best Practices* part offers step-by-step instructions for the most common features of the Open Build Service and the openSUSE Build Service. The last part covers ideas and motivations, concepts and processes of the Open Build Service.

2 Feedback

Several feedback channels are available:

Bugs and Enhancement Requests

Help for openSUSE is provided by the community. Refer to <https://en.opensuse.org/Portal:Support> for more information.

Bug Reports

To report bugs for Open Build Service, go to <https://bugzilla.opensuse.org/>, log in, and click *New*.

Mail

For feedback on the documentation of this product, you can also send a mail to doc-team@suse.com. Make sure to include the document title, the product version and the publication date of the documentation. To report errors or suggest enhancements, provide a concise description of the problem and refer to the respective section number and page (or URL).

3 Documentation Conventions

The following notices and typographical conventions are used in this documentation:

- /etc/passwd: directory names and file names
- PLACEHOLDER: replace PLACEHOLDER with the actual value
- PATH: the environment variable PATH
- ls, --help: commands, options, and parameters
- user: users or groups
- package name: name of a package
- **Alt**, **Alt + F1**: a key to press or a key combination; keys are shown in uppercase as on a keyboard
- *File*, *File > Save As*: menu items, buttons
- *Dancing Penguins* (Chapter *Penguins*, ↑Another Manual): This is a reference to a chapter in another manual.
- Commands that must be run with root privileges. Often you can also prefix these commands with the sudo command to run them as non-privileged user.

```
root # command
geeko > sudo command
```

- Commands that can be run by non-privileged users.

```
geeko > command
```

- Notices



Warning: Warning Notice

Vital information you must be aware of before proceeding. Warns you about security issues, potential loss of data, damage to hardware, or physical hazards.



Important: Important Notice

Important information you should be aware of before proceeding.



Note: Note Notice

Additional information, for example about differences in software versions.



Tip: Tip Notice

Helpful information, like a guideline or a piece of practical advice.

4 Contributing to the Documentation

The OBS documentation is written by the community. And you can help too!

Especially as an advanced user or an administrator of OBS, there will be many topics where you can pitch in even if your English is not the most polished. Conversely, if you are not very experienced with OBS but your English is good: We rely on community editors to improve the language.

This guide is written in DocBook XML which can be converted to HTML or PDF documentation. To clone the source of this guide, use Git:

```
git clone https://github.com/openSUSE/obs-docu.git
```

To learn how to validate and generate the OBS documentation, see the file [README](#).

To submit changes, use GitHub pull requests:

1. Fork your own copy of the repository.
2. Commit your changes into the forked repository.

3. Create a pull request. This can be done at <https://github.com/openSUSE/obs-docu> .

It is even possible to host instance-specific content in the official Git repository, but it needs to be tagged correctly. For example, parts of this documentation are tagged as `<para os="open-suse">`. In this case, the paragraph will only become visible when creating the openSUSE version of a guide.


I First Steps

1 Beginner's Guide 2

1 Beginner's Guide

Copyright © 2006– 2024 SUSE LLC and contributors. All rights reserved.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or (at your option) version 1.3; with the Invariant Section being this copyright notice and license. A copy of the license version 1.2 is included in the section entitled “GNU Free Documentation License”.

For SUSE trademarks, see <http://www.suse.com/company/legal/> . All other third-party trademarks are the property of their respective owners. Trademark symbols (®, ™ etc.) denote trademarks of SUSE and its affiliates. Asterisks (*) denote third-party trademarks.

All information found in this book has been compiled with utmost attention to detail. However, this does not guarantee complete accuracy. Neither SUSE LLC, its affiliates, the authors nor the translators shall be held liable for possible errors or the consequences thereof.

This guide describes basic workflows for working with packages on Open Build Service. This includes checking out a package from an upstream project, creating patches, branching a repository, and more.

1.1 Target Audience

This document is intended for users and developers interested in building packages from source code for different platforms and Linux distributions. Basic knowledge of Linux and the command line usage is recommended.

1.2 Conceptual Overview

Created in 2005, the Open Build Service (OBS) is a generic system for building and distributing packages or images from source code in an automatic, consistent, and reproducible way. OBS can create images and installable packages for a wide range of operating systems (SUSE, Debian, Ubuntu, Red Hat, Windows, etc.) and hardware architectures (x86, AMD64, z Systems, POWER etc.).

1.2.1 Build Recipe

To create a package in OBS, you need a *build recipe* which contains the following information:

- **Metadata.** The package name and the description are mandatory. Other data such as the version, the license, the upstream URL is optional.
- **Requirements.** Packages depend on other packages to function properly. There are two types of requirements: *build requirements* and *installation requirements*.
Build requirements are dependencies which are needed during the build process in OBS. For example, a C++ program needs a C++ compiler.
Installation requirements are dependencies which are needed when installing the final package.
- **A Package List.** To successfully install and remove a package and all its contents, the package manager needs to know which files and directories belong to which package.

For RPM-based operating systems such as openSUSE, SUSE Linux Enterprise, or Red Hat Enterprise Linux, all the information above is included in a file with the file extension .spec.

1.2.2 Build Hosts and Packages

The OBS server provides a Web interface and an API. The API is used by the osc command-line tool.

To build the package, the back end creates a sandbox with the respective distribution. This sandbox is isolated from the rest of the host system. Depending on the build recipe (on RPM-based systems, this is a *spec file*), other packages are downloaded and installed prior to building. The build process executes all the instructions that it finds in the build recipe. If the build is successful, the files which belong to the package are installed into the sandbox. From those sandboxed files, the final RPM package is created and moved into a download area, the *download repository*.

End users can install the package using their preferred package management tools. On a SUSE-based system, you can use YaST or Zypper command-line tool to install an RPM version of the package.

Other OBS-related services (like the notification server, mirror interface, etc.) perform very specific tasks and therefore beyond the scope of this guide.

The schematic in *Figure 1.1, “Conceptual Overview of Open Build Service”* shows the components in context.



FIGURE 1.1: CONCEPTUAL OVERVIEW OF OPEN BUILD SERVICE

1.2.3 Projects and Packages

In OBS, packages are organized in *projects*. A single project can contain several packages, and it usually serves a specific organizational purpose. Generic access control, related repositories, and build targets (operating systems and architectures) are all defined on the project level.

Projects can also have other projects (subprojects) to structure work. They are isolated from their parent project and can be configured individually.

Each project name is separated by colon. For example, in the openSUSE Build Service, packages for fonts are collected in the project `M17N:fonts` which is a subproject of `M17N`. Packages for the Python programming language are available in the `devel:languages:python` project which is a subproject of `devel:languages` which itself is a subproject of `devel`.

As a user, you will normally build packages in your *home project*, available in OBS as `home:USER-NAME`. Home projects serve as a personal working area in OBS to define build targets, upload, build, and download packages. Users are also permitted to create subprojects for temporary subprojects to work on other people's packages.

Sometimes, you will see the `obs://DOMAIN/PROJECT` notation. The `obs://` schema is a shorthand to abbreviate the long URL and needs to be replaced by the real OBS instance URL.

1.3 Requirements for Working with the **osc** Command-Line Tool

Before you start working with Open Build Service, make sure that the following requirements are met.

Software Requirements

Install the **osc** command line tool from your preferred distributions or from the OBS project `openSUSE:Tools`:

- For SUSE related systems, install the **osc** package with the **zypper** command (replace `DISTRIBUTION` with your distribution):

```
root # zypper ar https://download.opensuse.org/repositories/openSUSE:/  
Tools/DISTRIBUTION/openSUSE:Tools.repo  
root # zypper install osc
```

- For other systems, use your preferred package manager.
- As an alternative, you can use the AppImage file . An AppImage file is a packaged application which can run on many distributions. Download the file from <https://download.opensuse.org/repositories/openSUSE:/Tools/AppImage/>, save it in your `~/bin` directory, and make the file executable.

Hardware Requirements

Make sure you have a minimum of 1 GB of free disk space. The **osc** command builds all packages locally under `/var/tmp/oscbuild` and caches downloaded packages under `/var/tmp/osbuild-packagecache`.

1.4 Covered Scenarios

This guide is based on the following assumptions.

- Since Git is used throughout this guide, and many OBS concepts are modeled after their Subversion (SVN) equivalents, you have a working knowledge of version control systems such as Git and Subversion (SVN).
- You are using the openSUSE Build Service at <https://build.opensuse.org>. If you are using another OBS instance, some commands may differ.

- You have an account on an Open Build Service instance.
- You are running an RPM-based operating system like openSUSE or SUSE Linux Enterprise.
- You are using a customized system as shown in *Section 1.5, “Configuring Your System for OBS”*.

All examples use the following elements.

- A user on a local machine (you) called `geeko`. This user builds packages on their own machine.
- An OBS user called `obsgeeko` with home `home:obsgeeko` on the Open Build Service. This user is the same as the system user `geeko`, that is, you.
- An OBS user `obstux` and their home `home:obstux` on Open Build Service. This user acts as a collaborator.
- An example upstream open source project available at <https://github.com/obs-example/my-first-obs-package>. This project contains source code in the C++ programming language.

This guide describes the following common tasks:

Section 1.6, “Setting Up Your Home Project for the First Time”

Setting up a home project using the OBS Web UI.

Section 1.7, “Creating a New Project”

Creating packages from a basic project hosted on GitHub.

Section 1.8, “Patching Source Code”

Patching source code without touching the original source.

Section 1.9, “Branching a Package”

Branching a project, making changes, and submitting back the changes to the original project.

Section 1.10, “Installing Packages from OBS”

Integrating the download repository into your system and installing your built package.

1.5 Configuring Your System for OBS

While it is possible to use the `osc` tool without any configuration, it is recommended to set up your system as described below.

After all dependencies are downloaded and before the actual build process can start, you need to enter the `root` password. This can be inconvenient when you rebuild packages frequently. The configuration below modifies the `sudo` configuration to allow building packages without entering the `root` password. To maximize security, only specific users can have root privileges. Follow the steps below to customize `sudo`.

PROCEDURE 1.1: CONFIGURING `sudo`

To allow all users in the `osc` group to build packages without entering the `root` password, do as follows.

1. Log in as `root` and create a new group `osc`. This group will contain all users which are allowed to build packages:

```
root # groupadd osc
```

2. Add users to your newly created group `osc` which are allowed to build packages:

```
root # usermod -a -G osc geeko
```

Repeat this step to add other users, if necessary.

3. Run `visudo` to create the sudoers file `/etc/sudoers.d/osc`:

```
root # visudo -f /etc/sudoers.d/osc
```

4. Add the following lines to create a command alias that can be executed only by the `osc` group:

```
# sudoers file "/etc/sudoers.d/osc" for the osc group
Cmd_Alias OSC_CMD = /usr/bin/osc, /usr/bin/build
%osc ALL = (ALL) NOPASSWD:OSC_CMD
```

5. Log out of your system and log in again to apply the changes.
6. Create a new OBS configuration file:

```
geeko > osc ls home:obsgeeko
```

If you run the command for the first time, you will be prompted to enter your OBS user name and OBS password.



Note: Alternative Directory Structure

If you prefer to separate projects and subprojects in directories and subdirectories, change the following line in your configuration file `~/.osrcrc`:

```
checkout_no_colon = 1
```

This will use an alternate layout when checking out a project. For example, setting the option above and checking out the home project will generate the `home/obsgeeko` directory structure instead of the single `home:obsgeeko` directory.

However, this guide uses the default configuration with colons.

1.6 Setting Up Your Home Project for the First Time

This section shows how to set up your home project after creating an openSUSE account.

When you log in to your home project for the first time, it will be empty. To build packages, you need to select build targets (operating systems and architectures) first. Build targets are defined project-wide and every package in a project is built for each build target. However, you can disable build targets for a specific package.

Setting up a home project is done as shown below.

PROCEDURE 1.2: ADDING GLOBAL BUILD TARGETS TO YOUR HOME PROJECT

1. Log in to the Open Build Service instance.
2. Click the *Your Home Project* link in the Places menu on the left.
3. Click the *Repositories* tab, then the *Add from a Distribution* link.
4. Select the distributions you want to build for.
OBS shows several Linux distributions. For SUSE distributions, it is recommended to activate at least openSUSE Tumbleweed and the latest openSUSE Leap release. To enable package builds for SUSE Linux Enterprise, mark one of the *openSUSE Backports for SLE 12*. When you select a distribution, OBS shows a message that the distribution has been successfully added it to your home project.
5. Click the *Overview* tab to see the available build targets on the right side.

To add more build targets, repeat the procedure above.

To fine tune your build targets, click the *Repositories* tab, find the respective build target, and click the *Edit repository* link. This shows the available architectures and additional package repositories you can add.

1.7 Creating a New Project

This section demonstrates how to create packages from a simple C++ project hosted on GitHub (the “upstream project”). We assume that this project contains source code which you want to package for different SUSE distributions.



Note: Check the License First

Before building, go to the homepage of the upstream project and check the license.

For example, in the openSUSE Build Service, you must only redistribute packages which are available under an open source license. If you are submitting a package to openSUSE, its source code will be checked for license compliance. If your package is not released under an open source license, it will be rejected.

You can find a list of already used licenses in OBS at <https://license.opensuse.org>. For more details and a comparison of open source licenses in general, see https://en.wikipedia.org/wiki/Comparison_of_free_and_open-source_software_licenses.

To create a package from the upstream project, follow the steps below.

1. Set up your project as shown in [Section 1.6, “Setting Up Your Home Project for the First Time”](#).
2. In the terminal, choose or create a directory on a local partition that has enough space to hold the package sources.
3. Check out your home project:

```
geeko > osc checkout home:obsgeeko
```

This creates an empty `home:obsgeeko` directory in the current directory.

4. Create a new package in your local *working directory*:

```
geeko > cd home:obsgeeko
geeko > osc mkpac my-first-obs-package
```

5. Get the source code of the upstream project and save it in `home:obsgeeko/my-first-obs-package`.

Download a TAR archive of the sources. You do not have to unpack it yet.

In our example, the project is hosted on GitHub and you can use the following URL: <https://github.com/obs-example/my-first-obs-package/releases>. If there is no published release of a project, click the *Clone or download* button and download the latest sources using the *Download ZIP* link.

6. Create the build recipe. This file contains metadata and build instructions.

In this example, we are building an RPM for openSUSE. For RPM-based distributions, we create a *spec file*. The skeleton of such a spec file looks like this:

EXAMPLE 1.1: SKELETON OF A SPEC FILE

```
#
# spec file for package my-first-obs-package
#
# -- Copyright omitted --

Name:          my-first-obs-package ❶
Version:       0.1.0 ❶
Release:       0 ❶
License:       GPL-3.0 ❶
Group:         Documentation ❶
Summary:       Frobnication Tool ❶
Url:           https://github.com/obs-example/my-first-obs-package ❶
Source:        my-first-obs-package-%{version}.tar.gz ❶
BuildRequires: gcc ❷
BuildRequires: cmake ❷
BuildRoot:     %{_tmppath}/%{name}-%{version}-build

%description ❸
This tool frobnicates the bar with the foo when choosing the baz.

%prep ❹
%setup -q -n %{name}-%{version}

%build ❺

%install ❻

%files ❼
%defattr(-,root,root,-)
%doc README LICENSE *.txt
%{_bindir}/*
```

- ❶ **The Header.** Metadata like package name, version, release, license, the RPM group, a brief summary, the upstream URL, and the name of the source file.
- ❷ **Build Requirements.** Lists package dependencies that are required for building. The listed packages are downloaded and installed before building the package.
- ❸ **The Description Section.** Describes the purpose of the package and gives a comprehensive explanation.
- ❹ **The Preparation Section.** Prepares the sources for building. This usually includes unpacking them with the `%setup` macro and patching them using the `%patch` macro. (For more information about patching, see [Section 1.8, "Patching Source Code"](#).)
- ❺ **The Build Section.** Contains commands or RPM macros for building the package.
- ❻ **The Install Section.** Contains commands or RPM macros which create directories or copy files to the installation location.
- ❼ **The Files Section.** Lists all files and directories which belong to the package. Documentation-related files are marked with `%doc`, and they are automatically installed in the default documentation directory.
- ❽ **The Changelog Section.** This section is usually empty. Instead, OBS searches for a file with the extension `.changes`. If such a file exists in the project directory, it will be automatically included as a changelog. The changelog file itself contains a high level overview of the history of the package.

For the complete spec file, see https://build.opensuse.org/package/view_file/home:obs-geeko/my-first-obs-package/my-first-obs-package.spec.

7. Create a changelog file:

```
geeko > osc vc
```

This command opens a text file with the following content in the default editor:

```
-----  
Fri Aug 23 12:31:41 UTC 2017 - geeko@example.com
```

Add a short summary of your changes. Usually, a changelog entry contains a high-level overview such as the version of the software in the package, which patches you applied, and other changes in your project.

Save the file and leave the editor. **osc** then creates the file `my-first-obs-package.changes`.

Your project directory should now look something like this:

```
project directory
├─ my-first-obs-package-0.1.0.tar.gz
├─ my-first-obs-package.changes
└─ my-first-obs-package.spec
```

8. Add all the files to your working directory:

```
geeko > osc add *.spec *.changes *.tar.gz
```

9. Build the package for the default build target:

```
geeko > osc build --local-package
```

The option `--local-package` is used here, because the package is not yet submitted to OBS.

The default build target is set in the osc configuration file `~/.osrc` using the variable `build_repository`. On openSUSE Build Service this is usually openSUSE Tumbleweed. To build the package for another build target, use the following command:

```
geeko > osc build --local-package openSUSE_Tumbleweed x86_64 *.spec
```

10. Check whether your build was successful. If everything was fine, commit the files to your package to your home project on OBS:

```
geeko > osc commit
```

If you encounter build errors, use the `osc buildlog` command to review them, as described below.

To watch the current build of a specific build target, use the `buildlog` (alias `bl`) subcommand inside your working directory:

```
geeko > osc buildlog openSUSE_Tumbleweed x86_64
```

1.8 Patching Source Code

This section describes how to patch an upstream project. We use the same project as shown in [Section 1.7, “Creating a New Project”](#).

There are different reasons for patching a package.

- **You Do Not Have Permission to Commit Upstream.** Often, you cannot commit changes directly to the upstream project. If you send changes to the upstream project, they may be integrated late or even be rejected.
Patch files allow making changes while keeping source code clean and also allow independence from an upstream project's release cycle, coding style, and internal workings.
- **Apply Security and Bug Fixes or Distribution-Specific Fixes.**
- **Change the Source Code, So It Builds on OBS.**
- **Improve Security and Traceability.** Untouched source code in OBS is easier to check for changes than a modified one. The check is usually done with a checksum (MD5 or SHA).
- **Improve Structure and Consistency.** From an organizational point of view, it is better when changes are separated from the source code. With the changes inside the patch file, everybody can see *what* was changed and which files changes were applied to.

We assume that you already have a project as described in [Section 1.7, “Creating a New Project”](#). The project directory should look similar to this:

```
project directory
├─ my-first-obs-package-0.1.0.tar.gz
├─ my-first-obs-package.changes
└─ my-first-obs-package.spec
```

In our case, we want to modify the source code under `src/main.cpp` to change the greeting message.

PROCEDURE 1.3: PATCHING

1. In the terminal, switch to your working directory.
2. Prepare a patch file:
 - a. Unpack the source code:

```
geeko > tar xvf my-first-obs-package-*.tar.gz
```

If you have downloaded the archive from GitHub, the archive contains a directory in the form NAME-VERSION. In our case, unpacking the downloaded archive results in the my-first-obs-package-0.1.0/ directory.

- b. Switch to the directory my-first-obs-package-0.1.0/ and make a copy of the original C++ source file:

```
geeko > cd my-first-obs-package-0.1.0/
geeko > cp src/main.cpp src/main.cpp.orig
```

- c. Make your changes in src/main.cpp.

- d. Create a diff and carefully inspect your changes:

```
geeko > diff -u src/main.cpp.orig src/main.cpp
```

The output should look like this:

```
--- src/main.cpp.orig    2017-08-09 16:28:31.407449707 +0200
+++ src/main.cpp         2017-08-09 16:28:49.131541230 +0200
@@ -2,7 +2,7 @@

    int main()
    {
-       std::cout<<"Hello OBS!\n";
+       std::cout<<"Hello Alice!\n";

        return 0;
    }
```

- e. Redirect the diff into a file:

```
geeko > diff -u src/main.cpp.orig src/main.cpp \
> ../my-first-obs-package_main.diff
```

You can use an arbitrary name for the patch file. However, we recommend giving the file a descriptive name and adding the name of the upstream project. If there is a bug or issue number associated with the patch, add it to the file name as well. You can either use .diff or .patch as the file extension.

- f. You can now remove the directory my-first-obs-package-0.1.0/, as it is not needed anymore.

3. Open your spec file and add the following line in the header under the Source line like this:

```
Source:          my-first-obs-package-%{version}.tar.gz
Patch0:          my-first-obs-package_main.diff
```

4. In the %prep section, add the %patch macro:

```
%prep
%setup -q -n %{name}-%{version}
%patch0
```

5. Add your patch file to the local repository:

```
geeko > osc add my-first-obs-package_main.diff
```

6. Rebuild your package:

```
geeko > osc build
```

7. If everything was successful, commit your changes:

```
geeko > osc commit
```

When prompted, specify and save a commit message.

If you are dealing with a lot of patches, you might find the **quilt** tool useful. For more information about **quilt**, see <https://savannah.nongnu.org/projects/quilt>.

1.9 Branching a Package

This section describes how to collaborate between projects. You can *branch* any package in OBS into any project that you have write permission for. By default, new branches are created as subproject of your home project. These default branches have names beginning with home:obs-geeko:branches.

There are different reasons to branch a package:

- To modify the source code, building it, trying the effect of the changes, and submitting back changes to the original project. Usually, you use this workflow when you do not have write permissions for a project.
- To make changes without affecting the original project.
- To apply temporary changes to try out a different path of development.

Let us assume that there is a user `obsgeeko` who has created a package `home:obsgeeko/my-first-obs-package` on OBS. Now, a second user, `obstux`, would like to submit a code change request to that package.

User `obstux` has to perform the following steps:

PROCEDURE 1.4: BRANCHING FROM A PROJECT

1. In the terminal, choose or create a directory on a local partition with enough free space.
2. Create a branch from geeko's home project:

```
tux > osc branchco home:obsgeeko my-first-obs-package
```

This creates a branched package in OBS at `home:obstux:branches/my-first-obs-package` and checks out a directory `home:obstux:branches:home:obsgeeko:my-first-obs-package`.

3. Change the working directory to your checked-out branch:

```
tux > cd home:obstux/branches/home:obsgeeko/my-first-obs-package
```

4. Make changes as shown in [Section 1.8, "Patching Source Code"](#).
5. Build the package for the default build target:

```
geeko > osc build
```

6. Review the build log:

```
geeko > osc buildlog openSUSE_Tumbleweed x86_64
```

7. Make sure all included and removed files are added to the OBS repository:

```
tux > osc addremove
```

8. If everything was successful, commit your changes:

```
geeko > osc commit
```

When prompted, specify and save a commit message.

9. Create a submit request and finish it by adding a comment:

```
tux > osc submitreq
```

Used without any options, the `submitreq` command submits back to the package where you branched from. Note that with the submit request, you submit a specific version of the source. Later changes do not get automatically fetched by default.

If there are multiple packages in a branch, all packages will be submitted together. To avoid that, specify the names of the source and destination projects and the package name:

```
tux > osc submitreq home:obstux:branches:home:obsgeeko my-first-obs-package  
home:obsgeeko
```

User `obstux` has finished the task now and the submit request is assigned to `obsgeeko`. User `obsgeeko` can now either accept or decline the submit request (“SR”). Also, as long as the SR remains open, `obstux` can also supersede it with a new request.

- **Accept the Submit Request.** The changes from user `obstux` will be integrated into the `home:obsgeeko` project. The accepted submit request will be closed. To make further changes, `obstux` needs to create a new submit request.
- **Decline the Submit Request.** The changes from user `obstux` are not integrated into the `home:obsgeeko` project.

Reasons for declining a submit request can be build errors or style issues. The reviewer usually gives a reason when declining the submit request. User `obstux` can then 1) correct their submission using a new submit request that supersedes the previous one, 2) disagree and reopen the request, or 3) accept the decline and revoke the request.

- **Supersede the Submit Request.** As long as the SR is still open (and this includes the case when it has been declined), `obstux` can continue making changes in their local checkout and, at any time, issue a new submit request using the above workflow. At submission time, OBS will detect the existing (previous) SR and ask the submitter whether they would like to supersede it. If `obstux` answers "yes" here, the new SR will supersede the previous one. Alternatively, by answering "no" to the supersede question, a second SR will be created alongside the previous one.

User obsgeeko is responsible for the following.



Note

If preferred, the below steps can also be performed using the OBS GUI. Requests can be managed under the *Tasks* tab.

PROCEDURE 1.5: DEALING WITH SUBMIT REQUESTS

1. Show all submit requests that belong to your home project

```
geeko > osc request list -s new -P home:obsgeeko
```

2. Find the correct submit request. If you know the correct number you can use:

```
geeko > osc request show 246
```

3. Review the request and decide:

- Accept the submit request:

```
geeko > osc request accept 246 --message="Reviewed OK."
```

- Decline the request and give a reason:

```
geeko > osc request decline 256 --message="Declined, because of missing  
semicolon."
```

If the submit request has been accepted, the changes will be integrated into the home project home:obsgeeko.

If the submit request has been declined, you can fix the issues and resubmit the package. When creating a new submit request, osc will prompt to supersede the previous request.

1.10 Installing Packages from OBS

OBS provides a place containing all the distribution-specific and architecture-specific versions of successfully built packages. When you create a package in your OBS home project, all successfully built packages appear under the <https://download.opensuse.org/repositories/home:/obsgeeko> URL.

However, this is only true for the home project itself and manually created subprojects, but not for subprojects created as a result of branching a package. Branched projects are not published by default. If you need the build results, download the binaries manually with **osc getbinaries**. For example, if you have enabled the openSUSE Tumbleweed distribution, all packages for openSUSE Tumbleweed will be published at https://download.opensuse.org/repositories/home:/obsgeeko/openSUSE_Tumbleweed/. This *download repository* is used as an installation source for Zypper or YaST.

To install the `my-first-obs-package` package from your home project, use the following steps:

1. Inside your working directory, determine the download repository URLs:

```
geeko > osc repourls
https://download.opensuse.org/repositories/home:/obsgeeko/openSUSE_Tumbleweed/
home:obsgeeko.repo
https://download.opensuse.org/repositories/home:/obsgeeko/openSUSE_42.2/
home:obsgeeko.repo
```

2. Copy the desired URL of your preferred distribution. In our case, that is the line containing `openSUSE_Tumbleweed`.
3. Use **zypper** and add the copied URL:

```
root # zypper addrepo https://download.opensuse.org/repositories/home:/obsgeeko/
openSUSE_Tumbleweed/home:obsgeeko.repo
```

When prompted, accept the GPG key of the download repository.

4. Install the package:

```
root # zypper install my-first-obs-package
```

To update the package again, run [Step 4](#). You do not need to execute [Step 1](#), as the repository is already configured in your system.

1.11 Other Useful **osc** Commands

The following list gives you a short overview of frequently used **osc** subcommands that were not mentioned in this guide. For an overview of their syntax, use **osc SUBCOMMAND --help**.

osc diff

Generates a diff, comparing local changes against the remote OBS project.

osc list

Shows source or binaries on an OBS server.

osc prjresults

Shows project-wide build results.

osc status

Shows the status of files in your working directory

II Concepts

2 Supported Build Recipes and Package Formats **22**

2 Supported Build Recipes and Package Formats

2.1 About Formats

OBS differentiates between the format of the build recipes and the format of the installed packages. For example, the spec recipe format is used to build RPM packages by calling `rpmbuild`. In most cases, the build result format is the same as the package format used for setting up the build environment, but sometimes the format is different. An example is the KIWI build recipe format, which can build ISOs, but uses RPM packages to set up the build process.

OBS currently supports the following build recipe formats and packages:

SUPPORTED PACKAGE FORMATS

- RPM package format, used for all RPM-based distributions like openSUSE, SUSE Linux Enterprise, Fedora, and others.
- DEB package format, used in Debian, Ubuntu, and derived distributions
- Arch package format, used by Arch Linux

SUPPORTED BUILD RECIPE FORMATS

- Spec format for RPM packages
- Dsc format for DEB packages
- KIWI format, both product and appliances
- preinstallimage
- SimpleImage format
- Mkosi format to build images

If no build recipe format and binary format are specified in the project configuration, OBS tries to deduce them from the preinstall list, which includes the name of the used package manager. This means that you need to manually configure the `kiwi` build recipe, as an RPM package format will select `spec` builds as default. This configuration is done by adding a `Type` line to the project configuration.

2.2 RPM: Spec

RPM (RPM Package Manager) is used on openSUSE, SUSE Linux Enterprise, Red Hat, Fedora, and other distributions. For building RPMs you need:

.spec

the *spec file* for each package containing metadata and build instructions. OBS parses the spec file's BuildRequires lines to get a list of package dependencies. OBS uses this information to both build the packages in the correct order and also for setting up the build environment. The parser understands most of RPMs macro handling, so it is possible to use architecture specific BuildRequires, conditional builds and other advanced RPM features.

.changes

the file which contains the changelog.

2.3 Debian: Dsc

DEB packages are used on all Debian or Ubuntu based distributions. For building .deb files, you need:

debian.control

The file contains the meta information for the package like the build dependencies or some description.

debian.rules

This file describes the build section of the DEB building process. There are the configure and make compile commands including other DEB building sections.

PACKAGE.dsc

In this file you describe the package names of each subpackage and their dependency level. Unlike RPM, the release numbers are not increased automatically during build unless the keyword DEBTRANSFORM-RELEASE is added to the file.


2.4 Arch: pkg

Pkg files is used on Arch Linux and its derivatives. For building Pkg you need:

PKGBUILD

It contains the build description and the source tarball. The file PKGBUILD does not have macros like %{buildroot}. It contains variables, for example, makedepends=(PACKAGE1, PACKAGE2). These variables are parsed by OBS and uses them as dependencies. On Arch Linux you typically build packages without subpackage. They are no *-dev or *-devel packages.

2.5 KIWI Appliance

KIWI (<https://github.com/OSInside/kiwi>)  is an OS appliance builder that builds images for various formats, starting from hardware images, virtualization systems like QEMU/KVM, Xen and VMware, and more. It supports a wide range of architectures, which are x86, x86_64, s390 and ppc.

For building an image in KIWI you need:

my_image.kiwi

Contains the image configuration in XML format. Full XML schema documentation can be found https://osinside.github.io/kiwi/image_description.html .

config.sh (optional)

configuration script that runs at the end of the installation, but before package scripts have run.


root/

directory that contains files that will be applied to the built image after package installation. This can also be an archived and compressed directory, usually named root.tar.gz.



Note

OBS only accepts KIWI configuration files with a .kiwi suffix. Other naming schemes KIWI supports like config.xml, are ignored in OBS.

For more information about building images with KIWI, see the https://osinside.github.io/kiwi/building_images.html .

2.6 SimpleImage

This format can be used to get simple rootfs tarball or squashfs image. It does not contain a bootloader or a kernel. For advanced features, use KIWI. Use SimpleImage for simple rootfs tarball/squashfs image of any distribution that is supported by OBS but does not have anything fancier than that.

For building a SimpleImage, you need a `simpleimage` file. Be aware of the following points:

- SimpleImage uses a similar syntax than a spec file.
- Supported tags include `Name`, `Version`, `BuildRequires`, and `#!BuildIgnore`.
- Additional customization with `%build` phase is supported.
- RPM macros are not supported, but `$SRCDIR` shell variable is available.

EXAMPLE 2.1: SIMPLEIMAGE FILE (`simpleimage`)


```
Name:          example-image
Version:       1.0
BuildRequire:  emacs
#!BuildIgnore: gcc-c++


%build
# Set root password
passwd << EOF
opensuse
opensuse
EOF

# Enable ssh
systemctl enable sshd
```

2.7 ApplImage

2.8 Flatpak

The Flatpak (<https://flatpak.org/>)  format can be used to generate desktop apps for Linux.

For building an installable Flatpak bundle, you need a `flatpak.yaml` manifest file. See [Flatpak Manifests](https://docs.flatpak.org/en/latest/manifests.html) (<https://docs.flatpak.org/en/latest/manifests.html>)  for the full documentation.

Also some project and package configuration is necessary.

QUICK START

- To avoid having to enter the configuration manually, you can:
- go to the [Image Templates \(https://build.opensuse.org/image_templates\)](https://build.opensuse.org/image_templates) and create a project from there, or
- branch the [Template Package \(https://build.opensuse.org/package/show/OBS:Flatpak:Templates/FlatpakTemplate\)](https://build.opensuse.org/package/show/OBS:Flatpak:Templates/FlatpakTemplate)

MANIFEST FORMAT

- Input format is [YAML \(https://yaml.org\)](https://yaml.org). Although flatpak also accepts JSON, we are using YAML in Open Build Service, because we have a special additional field in form of a YAML comment.
- You can use JSON, as it is a subset of YAML. But be aware that flatpak accepts non-standard `//` comments, while Open Build Service does not.
- `#!BuildVersion` - Use this field to specify the version of your app so the `.flatpak` file will be versioned. Flatpak manifests do not have a version field.
- To learn YAML, have a look at this [YAML Tutorial \(https://www.yaml.info/\)](https://www.yaml.info/).

The base images for Freedesktop, GNOME, KDE are maintained in the [OBS:Flatpak \(https://build.opensuse.org/project/show/OBS:Flatpak\)](https://build.opensuse.org/project/show/OBS:Flatpak) repository in form of rpm packages. They are generated by installing the images from [Flathub \(https://flathub.org/\)](https://flathub.org/) and packing the directories into a tar archive.

EXAMPLE 2.2: FLATPAK MANIFEST (flatpak.yaml)

```
#####
# Flatpak manifest example for Open Build Service
# https://docs.flatpak.org/en/latest/manifests.html
# Input should be YAML, even though the file can have
# a .json suffix (JSON is a subset of YAML).
# Don't use '//' comments!
#####

# Special OBS field because flatpak does not have a version field
# Default will be '0' if the field is missing.
#!BuildVersion: 3.14.15
---
```

```

app-id: org.gnome.Mahjongg
runtime: org.gnome.Platform
sdk: org.gnome.Sdk
runtime-version: '3.36'
command: gnome-mahjongg

finish-args:
- --share=ipc
- --socket=fallback-x11
- --socket=wayland
- --device=dri
- --metadata=X-DConf=migrate-path=/org/gnome/Mahjongg/

cleanup:
- "/share/man"

modules:
- name: gnome-mahjongg
  buildsystem: meson
  sources:
  - type: archive

    # Source archives should be put into the OBS package, but you can
    # keep the original URL from where it was downloaded here.
    url: https://download.gnome.org/sources/gnome-mahjongg/3.36/gnome-
mahjongg-3.36.2.tar.xz

    # You can also just specify a simple filename
    # url: gnome-mahjongg-3.36.2.tar.xz

    # flatpak-builder will do a checksum
    sha256: 'd2e8f1563ee03d112a17042c4e99971295b36f3ba795c7d905d636cc94b8ae97'

```

EXAMPLE 2.3: FLATPAK PROJECT CONFIG (prj.conf)

```

Type: flatpak
Support: kmod-compat kernel-default perl-YAML-LibYAML

```

EXAMPLE 2.4: FLATPAK PROJECT META EXAMPLE





```

<project name="Your:Project:Name">
  <title>Title</title>
  <description>Description</description>
  <repository name="openSUSE_Leap_15.2">
    <path project="OBS:Flatpak" repository="openSUSE_Leap_15.2"/>
    <arch>x86_64</arch>
  </repository>
  <repository name="openSUSE_Leap_15.1">

```

```
<path project="OBS:Flatpak" repository="openSUSE_Leap_15.1"/>
<arch>x86_64</arch>
</repository>
</project>
```

2.9 mkosi

Mkosi (<https://github.com/systemd/mkosi/>)  allows building images for rpm, arch, deb and gentoo based distributions, on any architecture that supports UEFI. Images built with mkosi will follow the [Discoverable Partitions Specification](https://systemd.io/DISCOVERABLE_PARTITIONS/) (https://systemd.io/DISCOVERABLE_PARTITIONS/)  and will be bootable on baremetal (UEFI), virtual machines (UEFI), containers via [systemd-nspawn](https://www.freedesktop.org/software/systemd/man/systemd-nspawn.html), (<https://www.freedesktop.org/software/systemd/man/systemd-nspawn.html>)  or as [Portable Services](https://systemd.io/PORTABLE_SERVICES/) (https://systemd.io/PORTABLE_SERVICES/)  in systemd.

For building an image in mkosi you need the `mkosi.my_image` recipe file. This file contains the image configuration in INI format. All available options can be found in the [Mkosi documentation](https://github.com/systemd/mkosi/blob/main/mkosi.md) (<https://github.com/systemd/mkosi/blob/main/mkosi.md>) .



Note

Ensure to set `Type: mkosi` in the repository's `prjconf` where the image builds are enabled on OBS.

EXAMPLE 2.5: MKOSI MINIMAL BUILD RECIPE (`mkosi.suse`) FOR A TUMBLEWEED IMAGE

```
[Distribution]
Distribution=opensuse
Release=tumbleweed

[Output]
Format=gpt_ext4

[Content]
Password=
Autologin=yes
Packages=
patterns-base-minimal_base
```

EXAMPLE 2.6: MKOSI PROJECT CONFIG (`prjconf`)

```
%if "%_repository" == "suse"
Type: mkosi
```

```
Substitute: integritysetup
Substitute: veritysetup
Prefer: openSUSE-release-appliance-custom python310-cryptography
%endif
```

EXAMPLE 2.7: MKOSI PROJECT META EXAMPLE

```
<project name="Your:Project:Name">
  <title>Title</title>
  <description>Description</description>
  <repository name="suse">
    <path project="openSUSE:Factory" repository="snapshot"/>
    <arch>x86_64</arch>
  </repository>
</project>
```

III Setup

- 3 osc, the Command Line Tool **31**
- 4 Build Configuration **37**

3 osc, the Command Line Tool

3.1 Installing and Configuring

To work with Open Build Service, install the **osc** command line tool from your preferred openSUSE distributions or from the OBS project [openSUSE:Tools](#). The tool runs on any modern Linux system and is available for different distributions, like CentOS, Debian, Fedora, SLE, openSUSE, to name a few.

For SUSE related systems, install it with the **zypper** command (replace *DISTRI* with your distribution):

```
root # zypper addrepo https://download.opensuse.org/repositories/openSUSE:/Tools/DISTRI/
openSUSE:Tools.repo
root # zypper install osc
```

For other systems, use your preferred package manager.

As an alternative, use the AppImage file. An AppImage file is a packaged application and its dependencies which can run on many distributions. Download the file, save it in your [~/bin](#) directory, and make the file executable.

3.2 Configuring osc

Usually, the default configuration is appropriate in most cases. There are some special configuration options which might be helpful if you have special needs.

Some useful options in the [~/.osrc](#) file are described in the following list (all under the [general](#) section):

[apiurl](#) (string)

Used to access the Open Build Service (OBS) API server. This is needed if you work with different OBS servers (for example, a public and a private one). If you have to distinguish different servers, you can also use the [-A](#) option. Usually, it is good practice to create an alias like this:

```
alias iosc="osc -A https://api.YOURSERVER"
```

You use **iosc** the same as with **osc**.

extra-pkgs (list)

Contains a space-separated list of package. These extra packages are installed when you build packages locally. Useful when you need an additional editor inside the build environment, for example **vim**.

build_repository (string)

Sets the default platform when omitted in **osc build**.

exclude_glob (list)

Contains a list of space separated file names to ignore. For example, ***.bak** to ignore all backup files.

checkout_no_colon (bool)

Separates projects and subprojects in directories and subdirectories instead of creating a single directory. For example, setting the option and checking out the home project will lead to a directory structure **home/obsgeeko** instead of the single directory **home:obs-geeko**.

use_keyring (bool)

Use the default keyring instead of saving the password in the OBS configuration file. For KDE the KWallet is used, for GNOME it is Seahorse.

cafile (string)

Provide set of trusted CA certificates for HTTPs requests. Expects CAs in a single file containing a bundle of CA certificates in PEM format. More details can be found in [OpenSSL documentation \(https://www.openssl.org/docs/manmaster/man3/SSL_CTX_load_verify_locations.html\)](https://www.openssl.org/docs/manmaster/man3/SSL_CTX_load_verify_locations.html) ↗.

capath (string)

Provide set of trusted CA certificates for HTTPs requests. Expects a directory containing CA certificates in PEM format. More details can be found in [OpenSSL documentation \(https://www.openssl.org/docs/manmaster/man3/SSL_CTX_load_verify_locations.html\)](https://www.openssl.org/docs/manmaster/man3/SSL_CTX_load_verify_locations.html) ↗.

3.3 Usage

3.3.1 Getting Help

To get a general help about this command, use `osc --help`. For help of specific subcommands, use `osc help SUBCOMMAND`.

Most commands can be called by a long name (like `status`) or by one or more aliases (as `st`).

3.3.2 Using `osc` for the First Time

When you use the `osc` command for the first time, the command will ask you for your credentials of your OBS instance. The credentials are stored in the configuration file `~/.oscrc`.

By default, the password is stored as plain text. In terms of security, that is not ideal. To avoid the issue:

- **Use a Password Manager.** Set the option `use_keyring` to `1` after you have created a configuration file for the first time. Remove your `credentials` sections from your configuration file. The next time `osc` asks for your username and password, it will store it in the password manager instead of the configuration file.
- **Obfuscating the Password.** Set `plaintext_passwd` to `0`. This is not a security feature, but it obfuscates the password in the configuration file.

If you prefer your current password manager, set the option `use_keyring` to `1` after you have authenticated it.

3.3.3 Overview of Brief Examples

The `osc` command is similar to `git`: The main command `osc` has several subcommands. It serves as client and it is used to build packages locally, submit files to a remote OBS instance, edit metadata, or query build results.

List Existing Content on the Server

```
osc ls                #list projects
osc ls Apache         #list packages in a project
```

```
osc ls Apache flood          #list files of package of a project
```

osc ls shows you a list of projects on OBS. Which OBS instance it shows depends on the option `apiurl` in the configuration file. By default, the openSUSE Build Server is used. If you need another server, use the `-A` option as shown in [Section 3.2, “Configuring osc”](#).

Checkout Content

```
osc co Apache                # entire project
osc co Apache flood          # a package
osc co Apache flood flood.spec # single file
```

Update a Working Directory

```
osc up
osc up [directory]
osc up *                # from within a project dir, update all packages
osc up                  # from within a project dir, update all packages AND check out
                        all newly added packages
```

Upload Changed Content

```
osc ci                    # current dir
osc ci [file1] [file2]    # only specific files
osc ci [dir1] [dir2] ...  # multiple packages
osc ci -m "updated foobar" # specify a commit message
```

Check the Commit Log

```
osc log
```

Show the status (which files have been changed locally)

```
osc st
osc st [directory]
```

If an update cannot be merged automatically, a file is in 'C' (conflict) state, and conflicts are marked with special lines. After manually resolving the problem, use **`osc resolved FILE`**.

Mark files to be Added or Removed on the Next Checkin

```
osc add foo
osc rm foo
```

Add all New Files in Local Copy and Removes all Disappeared files

```
osc addremove
```

Generate a diff to view the changes

```
osc diff [file]
```

Show the Build Results of the Package

```
osc results  
osc results [platform]
```

Show the Log File of a Package

(you need to be inside a package directory)

```
osc buildlog [platform] [arch]
```

Show the URLs of .repo Files which are Packages Sources for Package Managers

```
osc repourls [dir]
```

Trigger a Package Rebuild for all Repositories/Architectures of a Package

```
osc rebuildpac [dir]
```

Build a Package on Your Computer

```
osc build [platform] [arch] [specfile] [--clean|--noinit|...]
```

Show Configured Platforms/Build Targets

```
osc platforms [project]
```

Show Possible Build Targets for Your Project

```
osc repos
```

Show Metadata

```
osc meta prj [project]  
osc meta pkg [project] [package]  
osc meta user [username]  
osc meta prjconf [project]
```

Edit Meta Information

Create new package/project if it does not exist. It will open an editor with the raw XML metadata. To avoid need to edit XML, you can use the web UI instead.

```
osc meta prj -e [project]
osc meta pkg -e [project] [package]
osc meta prjconf -e [project]
```

(The project configuration may well be empty. It is needed in special cases only.)

Update Package Metadata on OBS with Metadata Taken from Spec File

```
osc updatepacmetafromspec [dir]
```

4 Build Configuration

4.1 About the Build Configuration

Each project has a *build configuration* which defines the setup of the build system and the publish behaviour. Usually the distribution base is defining it and you do not need to change anything. However, when you change it, it can be used for the following reasons:

- Handle compatibility layers.
- Switch on or off certain features during the build.
- Decide which package is installed during build.
- Resolve dependency problems like when there are multiple providers for a dependency:
- Handle user decisions like macro settings
- Modify publish behaviour, e.g. define metadata or filter binaries.

The build configuration can be stored in multiple places. Inside of OBS it can be stored only once per project. However, it is possible to define exceptions via if-conditions, for example per repository name or architecture. At the project level, the build configuration is sometimes referred to as the "project config", or "prjconf" for short. At build time, the configuration gets merged according to the repository path configuration defined in the project metadata. This resulting configuration can be requested using **`osc buildconfig`**. To view or edit the build configuration in projects, use one of the following methods

- With **`osc`**. Use **`osc meta prjconf`** in your working directory of your project.
- In the OBS Web UI. Via the *Project Config* tab.
- With the OBS API. Reachable via the `/source/PROJECT/_config` path.
- Via git/scmsync. Store a file called '`_config`' in a git repository used via the scmsync mechanism for a project.

4.2 Configuration File Syntax

The syntax is basically the same as in RPM spec files. However, it is independent of the packaging format used. The build configuration (prjconf) is parsed by OBS. This means, you can use RPM features like macros or conditions in the configuration. All lines (except conditionals) have the form:

```
keyword: arguments
```

Use the conditionals (%if or %ifarch) if a line should only be used in some condition.

Many keywords, like ``Require`` or ``BuildFlags``, do not replace existing data, but add to it. This means you can have multiple ``BuildFlags`` lines instead of having one line with all the flags you need.

An exclamation mark ``!`` can be prepended to the argument to remove an existing entry from the data.

In the following list, the placeholder *PACKAGES* indicates a package base name (or names). When specifying multiple packages, separate their base names with spaces. For example, as a package name you need the base name like *gcc* but not the full name as in *gcc-1.2.3.i386.rpm*.

The following list contains a list of allowed keywords in the build configuration (prjconf):

AVAILABLE KEYWORDS IN BUILD CONFIGURATION

BinaryType: *TYPE* (OBS 2.4 or later)

The binary type is the format of the packages that make up the build environment. This is usually set automatically depending on the recipe type and preinstall package list. Currently understood values are: ``rpm``, ``deb``, and ``arch``.

Sets the binary format used to set up the build environment. For example a package with spec build description may use and generate deb packages instead of RPMs. If no binary type is specified, OBS deduces it from the build recipe type. If the recipe type is also not set, OBS looks at the Preinstall package list for a hint.

BuildEngine: *ENGINE*

Use an alternative build engine. Examples are ``mock`` (for Fedora and Red Hat) and ``debootstrap`` (for Debian), ``debbuild`` (to build debian packages with spec files), ``podman`` (container builds). Here is an example config for ``debbuild``:

```
Type: spec
Repotype: debian
Binarytype: deb
```



```
BuildEngine: debbuild
Support: pax debbuild
```

BuildFlags: FLAG:VALUE

The BuildFlags keyword defines flags for the build process. The following values for FLAG are usable.

allowrootforbuild

Allow any package build to use root user for building. This still needs a marker inside of the build description to enable it.

vmfstype:TYPE

Defines a specific file system when building inside of a VM. Possible values are ext2, ext3, ext4, btrfs, xfs, reiserfs (v3).

vmfsoptions:OPTION

Sets options for file system creation. Currently only the ``nodirindex`` option is supported, which disables directory indexing for ext file systems. This makes file ordering inside of directories reproducible but may have a negative performance impact.

vmfsoptions:nodirindex

kiwiprofile:PROFILE

Selects the profiles to build in kiwi appliance builds.

logidlelimit:SECONDS

Build jobs which do not create any output are aborted after some time. This flag can be used to modify the limit.

excludebuild:PACKAGE

Exclude a package from building. If a package builds multiple flavors, the corresponding flavor can be specified via the ``package:flavor`` syntax.

onlybuild:PACKAGE

DANGER: this may remove many build results when introduced the first time! It can be used to maintain a whitelist of packages to be built. All other packages will turn to excluded state and get removed if available.

useccache:PACKAGE

Configure usage of ccache when building the specified package.

ccachetype:TYPE

Defines the ccache implementation, possible values are: ccache, sccache

obsgendiff

OBS can run an external program that has access to the current build and the previously successful result, e.g. to generate a difference or a changelog from the diff.

OBS will run all scripts in `/usr/lib/build/obsgendiff.d/` on the build host (not in the `%buildroot`) when this flag is set. If one of the scripts fails to run or no scripts are found, then the overall build fails. I.e. if `BuildFlags: obsgendiff` is set, then you *must* provide at least one script in `/usr/lib/build/obsgendiff.d/`, otherwise your build will fail.

A common use case for `obsgendiff` is to run [release-compare](https://github.com/openSUSE/release-compare) (<https://github.com/openSUSE/release-compare>) after the build.

setvcs

Add the SCM URL to binary results when the package sources are managed via the `scmsync` mechanic. The url is written into the VCS tag of rpms when enabling this functionality.

nodisturl

Skip the embedding of `DISTURL` tag into binaries. Please note that this might also break other features like binary tracking. So never do this for maintained binaries.

sbom:FORMAT

OBS 2.11 can produce and publish additional SBOM (Software Bill Of Material) meta data by enabling this flag. This is currently supported for container and kiwi images builds and includes only data from installed rpm packages. Supported formats are `spdx` and `cyclonedx`.

slsaverion:VERSION

OBS 2.11 is producing `slsa` provenance files in version 0 by default at build time, when enabled. It is possible to switch to `v1` by specifying `slsaverion:v1` as buildflag.

container-compression-format:FORMAT

Sets a compression format for container layers. Possible values are `gzip`, `zstd`, `zstd:chunked`. Every value other than `gzip` is only supported by `podman` and `buildah`.

container-build-format:FORMAT

For `podman` container builds, it specifies the container config format. Possible values are `'docker'` and `'oci'`. The default is `'docker'`. The `'docker'` format allows a few extensions like `ONBUILD`, `SHELL`, `DOMAINNAME`, `COMMENT`, `HEALTHCHECK` amongst others.

Conflict: *PACKAGE*

Specify that a package must not be installed in the build environment.

Conflict: *PACKAGE_A:PACKAGE_B*

Specify a synthetic conflict between two given packages.

Constraint: *SELECTOR STRING* (OBS 2.4 or later)

Define build constraints for build jobs. The selector is a colon-separated list which gets a string assigned. See the build job constraints page for details.

DistMacro: *NAME VALUE*

Define a macro to be used when parsing the spec files of packages. This is similar to using a `Macros:` section with the difference that the macro will not be written to the .rpmmacros file. It should therefore be used for macros that come from packages of the distributions. Note that the lines of the project config are macro expanded while parsing, so you have to use `%%` for a literal percent sign in the value.

ExpandFlags: *FLAG*

Flags which modify the behaviour during dependency resolution.

unorderedimagerepos (OBS 2.10 or later)

The priority of repositories defined in an image build is usually important. This is to avoid switching repositories when the same package is available in multiple repositories. However, it might be wanted to ignore that and just pick the highest version. This can be achieved by defining this flag

preinstallexpand

Preinstall also all dependencies of a preinstalled package. This may increase the amount of preinstalled packages a lot.

module:NAME-STREAM (OBS 2.10.7 or later)

Enable Red Hat-specific module support in repo md repositories. By default, no module is used, so every module needed needs to be specified in the configuration. To remove a module, add an exclamation mark (!) as prefix.

dorecommends

Try to install all recommended packages. Packages with dependency conflicts are ignored.

dosupplements

Try to install all supplemented packages. Packages with dependency conflicts are ignored. This has the downside that new packages can cause different dependency expansion, so this should only be enabled for special use cases.

ignoreconflicts

Ignore defined conflicts of packages. By default these are reported as unresolvable. This switch may be useful when packages get not installed in the build environment, but getting processed afterwards. That tool, e.g. some image building tool, must be able to handle the situation (e.g. by just using a subset of the packages).

kiwi-nobasepackages

Do not put the require/support/preinstall packages in the repositories offered to the kiwi build tool. This should have been the default.

keepfilerequires

Dependencies on files are only fulfilled if matching FileProvides are specified in the build configuration (prjconf). If those are missing, the dependency results in an "unresolvable" state for directly required files or in silent breaking of the dependency for indirectly required files. With this option, all file requires are honoured by default and lead to "unresolvable" if there are no matching FileProvides defined.

ExportFilter: *REGEX ARCHITECTURES*

The export filter can be used to export build results from one architecture to others. This is required when one architecture needs packages from another architecture for building. The *REGEX* placeholder must match the resulting binary name of the package. It will export it to all listed scheduler architectures. Using a single dot will export it to the architecture which was used to build it. So not using a dot there will filter the package.

FileProvides: *FILE PACKAGES*

OBS ignores dependencies to files (instead of package names) by default. This is mostly done to reduce the amount of memory needed, as the package file lists take up a considerable amount of repository meta data. As a workaround, FileProvides can be used to tell the systems which packages contain a file. The File needs to have the full path.

HostArch: *HOST_ARCH*

This is used for cross builds. It defines the host architecture used for building, while the scheduler architecture remains the target architecture.

Ignore: *PACKAGE_OR_DEPENDENCY*

Ignore can be used to break dependencies. This can be useful to reduce the number of needed packages or to break cyclic dependencies. If a package is specified, all capabilities provided by the package are ignored.

Ignore: *ORIGIN_PACKAGE:PACKAGE_OR_DEPENDENCY*

Ignore a dependency coming from *ORIGIN_PACKAGE*. See the previous section for more details.

Keep: *PACKAGES*

To eliminate build cycles the to-be-built package are not installed by default, even when it is required. Keep can be used to overwrite this behavior. It is usually needed for packages like *make* that are used to build itself. Preinstalled packages are automatically kept, as the package installation program needs to work all the time.

OptFlags: *TARGET_ARCH_FLAGS* (RPM only)

Optflags exports compiler flags to the build by adding lines to rpm's *rpmrc* file. They will only have an effect when the spec file is using *\$RPM_OPT_FLAGS* or *%{optflags}*. The target architecture may be set to *** to affect all architectures.

Order: *PACKAG_A:PACKAGE_B*

The build script takes care about the installation order if they are defined via dependencies inside of the packages. However, there might be dependency loops (reported during setup of the build system) or missing dependencies. The *Order* statement can be used then to give a hint where to break the loop.

The package in *PACKAGE_A* will get installed before the package in *PACKAGE_B*.

Patterntype: *TYPE*

Defines the pattern format. Valid values are: none (default), ymp, comps. Multiple types can be specified.

Prefer: *PACKAGES*

In case multiple packages satisfy a dependency, the OBS system will complain about that situation. This is unlike like most package managing tools, which just pick one of the package. Because one of OBS' goal is to provide reproducible builds, it reports an error in this case instead of choosing a random package. The Prefer: tag lists packages to be preferred in case a choice exists. When the package name is prefixed with a dash, this is treated as a de-prefer.

Prefer: *ORIGIN_PACKAGE:PACKAGE*

It is possible to define the prefer only when the dependency comes from the specified originating package.

Preinstall: *PACKAGE*

This is used to specify packages needed to run the package installation program. These packages are unpacked so that the native installation program can be used to install the build environment. Included scripts are *not* executed during this phase. However, these packages will be re-installed later on including script execution.

PublishFilter: *REGEXP [REGEXP]*

Limits the published binary packages in public repositories. Packages that match any *REG-EXP* will not be put into the exported repository. There can be only one line of PublishFilter for historic reasons. However, multiple *REGEXP* can be defined.

PublishFlags: *FLAG*

Flags which modify the behaviour during repository generation.

artifacthub:TAG (OBS 2.11 or later)

Publish a specific verified publisher identifier (aka repository id) for artifactub.io. This can be used to proof to be the publisher (aka maintainer) for containers from these OBS repositories.

createempty (OBS 2.11 or later)

Create a repository even with no content, but with meta data.

noearlypublish (OBS 2.11 or later)

Only publish build results after entire repository has finished building. This is the default for classic package (rpm/deb) build types, but not for certain image builds (like kiwi or products). Without this, build results get published immediately after the build is finished.

archsnc (OBS 2.11 or later)

Publish all architectures at the same time. This means that the publisher is waiting until the last architecture has finished building.

nofailedpackages (OBS 2.11 or later)

Block publishing if any build result was failed, broken, or unresolvable. That means, packages can be published for an architecture on which it builds, even if a package fails to build on another architecture.

This is by default evaluated individually for each architecture. It can be combined with the `archsnc` flag when publishing should be blocked also when a failure exists on any architecture.

keepobsolete (OBS 2.11 or later)

The default behaviour of OBS is to remove binaries when a package object gets removed, even when the project is publish disabled (but the package may have been enabled).

This flag is changing this behaviour to keep binaries in published repositories even when a package gets removed. As consequence this flag has only an effect on publish disabled projects.

singleexport (OBS 2.11 or later)

If multiple packages contain different versions of a rpm package, only publish the one from the first package. If the project is of the type `maintenance_release`, this will be the package with the highest incident number.

withsbom (OBS 2.11 or later)

Enables publishing of SBOM files (SPDX or CycloneDX format) (`.cdx.json` or `.spdx.json` files). Please note that the building of SBOM files usually needs to get enabled via a BuildFlags switch as well.

withreports (OBS 2.11 or later)

Also publish internal content tracking files (`.report` files).

ympdist:NAME (OBS 2.11 or later)

Defines the distversion to be used in group element of ymp files. This is used by the installer to check if the repository is suitable for the installed distribution.

RegistryURL: URL

Define a url for the downloading of containers.

Reptype: TYPE[:OPTIONS]

Defines the repository format for published repositories. Read on for permissible values (repository types). The syntax of the OPTIONS parameter depends on the repository type, and is also described below.

This is the list of repository types. Multiple values can be combined in the same line, separated by spaces.

rpm-md

rpm-md repository data is generated as invented by YUM originally.

suse

suse tag repository data is generated as used until SUSE Linux 10 generation.

hdlist2

Mandriva repository format

debian

Debian repository format

arch

Arch Linux repository format

vagrant

Vagrant image repository format

staticlinks

Additional links to build results excluding version and build numbers are created.

zyppservice

Generate zypp service files, publishing all used repository pathes to the zypp client.

helm

Helm repository metadata

checksumsfile

Checksums file with signatures of repository content

ymp

YaST single install ymp file generation. Zypp services should be used instead.

comps

Generate comps files, Fedora style patterns

This is the list of repository options to modify the way the repository type is generated:

sha256

rpm-md repository data is generated using SHA-256 checksums. This is the default.

sha512

rpm-md repository data is generated using SHA-512 checksums.

legacy

rpm-md repository data is generated using SHA-1 checksums. Considered to be unsafe and not anymore recommended.

zchunk

rpm-md repository data gets extended with additional zchunk compressed files.

filelists-ext

rpm-md repository provides additional filelist-ext provides

compression-zstd

rpm-md repository data is compressed using zstd instead of gz

deltainfo

rpm-md repository provides additional rpm delta informations for incremental rpm updates.

splitdebug:SUFFIX

A second repository is generated where all debuginfo and debugsource packages get moved to. The specified SUFFIX is added to the repository name. For example:

```
RepoType: rpm-md splitdebug:-debuginfo
```

rsyncable

rpm-md repository gets recompressed using rsyncable format.

rawsig

checksumsfile repository provides binary signatures instead of ascii signatures

RepoURL: [TYPE@]URL

Define a url for the downloading of repository packages. Supported types are currently `arch`, `debian`, `hdlist2`, `rpmmd`, `suse`. If the type is not specified, it is guessed from the build type.

Required: PACKAGE

Specify a package that always is installed for package builds. A change in one of these packages triggers a new build.

Runscripts: PACKAGES

Defines the scripts of preinstalled packages which needs to be executed directly after the preinstall phase, but before installing the remaining packages.

Substitute: OLD_DEPENDENCY NEW_DEPENDENCY

It is possible to replace BuildRequires dependencies with other dependencies. This will have only an effect on directly BuildRequired packages, not on indirectly required packages.

Support: *PACKAGE*

Specify a package that always is installed for package builds. Unlike `Required:`, a change in one of these packages does not trigger an automatic rebuild.

This is useful for packages that most likely do not influence the build result, for example `make` or `coreutils`.

Target: *TARGET_ARCH*

Defines the target architecture. This can be used to build for i686 on i586 schedulers for example. Please note that on rpm based systems just the architecture needs to be specified, but on debian systems the gnu triplet, for example `arm-linux-gnueabi`.

Target: *GNU_TRIPLET*

Defines the target architecture. This can be used to build for i686 on i586 schedulers for example. Please note that on rpm based systems just the architecture needs to be specified, but on debian systems the gnu triplet, for example `arm-linux-gnueabi`.

Type: *TYPE*

Build recipe type. This is the format of the file which provides the build description. This gets usually autodetected, but in some rare cases it can be set here to either one of these: `spec`, `dsc`, `kiwi`, `livebuild`, `arch`, `preinstallimage`, `mkosi`.

Defines the build recipe format. Valid values are currently: `none`, `spec`, `dsc`, `arch`, `kiwi`, `preinstallimage`. If no type is specified, OBS deduces a type from the binary type.

VMInstall: *PACKAGE*

Like `Preinstall`, but these packages get only installed when a virtual machine like Xen or KVM is used for building. Usually packages like `mount` are listed here.

4.3 Building with `ccache` or `sccache`

The usage of `ccache` or `sccache` can be enabled for each package by setting the `useccache:PACKAGE` build flag.

The `ccache` package will automatically be installed and configured. The directory `/.ccache/` will be configured as cache directory. To configure `ccache`, the file `/.ccache/ccache.conf` can be modified as part of the build process by the `$BUILD_USER` environment variable.

In some cases, there is no archive for the current package, such as when the package was newly branched or when binaries were deleted. In these cases, the system will check whether there is a package of the same name built for the same architecture within one of the repositories

configured in the project's meta configuration. If so, the archive of that package will be used. The repositories will be searched in the order they are configured in the meta configuration, starting from the top.

An alternative way to enable caching based on build dependencies is to add "--enable-cache" as dependency, for example via a Substitute rule:

```
Substitute: gcc-c++ gcc-c++ --enable-ccache
```

This will always enable ccache when a direct build dependency to gcc-c++ is required.

It is also possible to set the type, eg:

```
Substitute: cargo cargo --enable-ccache=sccache
```

4.4 Macro Definitions in the Build Configuration

You can use rpm macro definitions in the build configuration (prjconf) to improve configurability. There are two types of macros that can be defined in the build configuration:

- **Macros: Macro Definitions.** Macros defined after a `Macros:` line are exported into the `.rpmmacros` file of the build root. As such, these macro definitions can be used in a spec file. The section may be closed via a `:Macros` line. This is sometimes required to avoid parse errors, for example when the macro definitions are inside an `%if-%endif` statement. The `Macros:` section is always verbatim: any condition to activate it must be outside of the `Macros:` tags. Macros defined in this section are used by Open Build Service for build dependency resolution and are also available at build time. Any definition here is also overwriting definitions provided by any packages.
- **DistMacro.** The `DistMacro` can be used to define a macro for build dependency resolution. It is intended to be used when a distribution defines relevant macros inside of any of their packages. These macros are hidden by default to the Open Build Service dependency resolver. Use this directive to make it known to Open Build Service. Unlike `Macros:`, `DistMacro` won't overwrite any existing and possibly changed definition at build time.
- **%define Macro Definitions.** Macro definitions starting with a `%define` line are used during the build configuration parsing only. These definitions are *not* available inside the build root or during parsing of build recipes. Typical use cases are macros inside of `%if` statements inside of the build configuration or macros in any `Support`, `Require` or `Substitute` line.

4.4.1 Macros for the Build Configuration Only

To specify macros for the building process, use the `%define` keyword.

For example, if you put this line in the `prjconf`

```
%define _with_pulseaudio 1
```

then the macro `%_with_pulseaudio` will expand to `1` only inside the build configuration.

4.4.2 Macros Used in Spec Files Only

To define the values of macros used in spec files, `%define` is **not** used. For this use case, either enclose the macro definitions between `"Macros:"/" :Macros"` lines, or place them at the end of the `prjconf` file. All lines after a line containing the directive `"Macros:"` up to the end of the config, or up to a `:Macros` line, are used when parsing spec files and also made available to the build by copying them to the `.rpmmacros` file in the build root.

The macro definition in the project configuration is located at the end and has the following structure:

EXAMPLE 4.1: STRUCTURE OF A MACRO DEFINITION

```
Macros:
  # add your macro definitions
:Macros
```

Everything that starts with a hash mark (`#`) is considered a comment.

The macro definition itself are defined without a `%define` keyword. Start with `%macroname`, for example:

```
%_hardened_build 0
```

IV Usage

- 5 Basic OBS Workflow **52**
- 6 Local Building **62**
- 7 Using Source Services **65**
- 8 SCM/CI Workflow Integration **71**
- 9 Staging Workflow **107**
- 10 Notifications **112**
- 11 Moderation **120**

5 Basic OBS Workflow

5.1 Setting Up Your Home Project

This section shows how to set up your home project with the command line tool `osc`. For more information about setting up your home project with the Web UI, see [Section 1.6, “Setting Up Your Home Project for the First Time”](#).

This chapter is based on the following assumptions:

- You have an account on an Open Build Service instance. To create an account, use the Web UI.
- You have installed `osc` as described in [Section 3.1, “Installing and Configuring”](#).
- You have configured `osc` as described in [Section 3.3.2, “Using `osc` for the First Time”](#).

PROCEDURE 5.1: SETTING UP YOUR HOME PROJECT

1. Get a list of all available build targets of your OBS instance:

```
geeko > osc ls /
```

For example, on the openSUSE Build Service, build targets will include distributions such as `openSUSE:Tumbleweed`, `openSUSE:Leap:VERSION`, `openSUSE:Tools`, `openSUSE:Templates`.

2. Configure your build targets with:

```
geeko > osc meta prj --edit home:obsgeeko
```

The previous command shows a XML structure like this:

EXAMPLE 5.1: XML STRUCTURE OF BUILD SERVICE METADATA

```
<project name="home:obsgeeko">
  <title>obsgeeko's Home Project</title>
  <description>A description of the project.</description>
  <person userid="obsgeeko" role="bugowner"/>
  <!-- contains other OBS users -->
  <debuginfo>
    <enable repository="openSUSE_Factory"/>
  </debuginfo>
```

```
<!-- add <repository> elements here -->
</project>
```

3. To add build targets, use the `repository` element. For example, on openSUSE Build Service, you can add the build targets openSUSE Tumbleweed for x86 and x86-64 with:

```
<repository name="openSUSE_Tumbleweed">
  <path project="openSUSE:Tumbleweed" repository="standard"/>
  <arch>i586</arch>
  <arch>x86_64</arch>
</repository>
```

4. Add more `repository` elements as needed. Insert the information from [Step 1](#) into the `project` attribute.

On openSUSE Build Service, you can normally use the attribute `repository` with the value `standard`. For example, to add openSUSE Leap as a build target, create an entry like:

```
<repository name="openSUSE_Leap_42.3">
  <path project="openSUSE:Leap:42.3" repository="standard"/>
  <arch>i586</arch>
  <arch>x86_64</arch>
</repository>
```

5. Save the file (or leave it untouched).

`osc` will check if the new configuration is valid XML. If the file is valid, `osc` will save it. Otherwise, it will show an error and prompt you whether to *Try again?*. In this case, press **n**. Your changes will be lost and you will need to start from [Step 2](#) again.

After a while, the defined build targets show up in your home project.

5.2 Creating a New Package

This section covers how to create packages for an arbitrary software project, which we will refer to here as the “upstream project”. We assume that this project contains source code which you want to package for one or more SUSE (openSUSE) distributions. We assume the setup of your home project in your OBS instance is already done. If not, refer to [Section 5.1, “Setting Up Your Home Project”](#).

To create a package from the upstream project, do the following:

PROCEDURE 5.2: GENERAL PROCEDURE TO BUILD A RPM PACKAGE

1. Open a shell. Choose or create a directory on your system in a partition that has enough space to hold the package sources.
2. Prepare your *working directory*. These steps only have to be performed once:


- a. Check out your home project:

```
geeko > osc checkout home:obsgeeko
```

This will create home:obsgeeko in the current directory.

- b. Create a new package inside your local working directory:

```
geeko > cd home:obsgeeko
geeko > osc mkpac YOUR_PROJECT
```

3. Download the source of the upstream project and save it in home:obs-geeko/YOUR_PROJECT.
4. Create a *spec file* which contains metadata and build instructions. For more information about spec files, see <https://rpm-packaging-guide.github.io> .
5. Create a new changelog and add your changes:

- a. To create a new changelog file or to update an existing changelog file with osc, use:

```
geeko > osc vc
```

The command will open an editor with the following content:

```
-----
Fri Aug 23 08:42:42 UTC 2017 - geeko@example.com
```

- b. Enter your changes in the editor.

Usually, changelog entries contain a high-level overview like:

- **Version Updates.** Provide a general overview of new features or changes in behavior of the package.
- **Bug and Security Fixes.** If a bug was fixed, mention the bug number. Most projects have policies or conventions for abbreviating bug numbers, so there is no need to add a long URL.
For example, in openSUSE Build Service, `boo#` is used for bugs on <https://bugzilla.opensuse.org> and `fate#` is used for features on <https://fate.opensuse.org>.
- **Incompatible Changes.** Mention incompatible changes, such as API changes, that affect users or other developers creating extensions of your package.
- **Distribution-Related Changes.** Mention any changes in the package structure, package names, and additions or removals of patch files or “hacks”.

For more information about changelogs, see [https://en.opensuse.org/openSUSE:Creating_a_changes_file_\(RPM\)](https://en.opensuse.org/openSUSE:Creating_a_changes_file_(RPM)).

6. Add all the files to your working directory:

```
geeko > osc add *.spec *.changes *.tar.gz
```

7. Build the package for a specific distribution and architecture, for example, openSUSE Tumbleweed for x86-64:

```
geeko > osc build --local-package openSUSE_Tumbleweed x86_64 *.spec
```

If you encounter problems, see [Section 5.3, “Investigating the Local Build Process”](#).

8. Check if your build was successful. If everything was fine, commit the files to your package to your home project on OBS:

```
geeko > osc commit
```

To delete a file in your working directory, merely deleting it from the local filesystem (`rm FILE`) is not sufficient, since `osc`, like any other Source Code Control System, will just complain that the file is missing. If you really want to delete a file, use the command:

```
geeko > osc delete FILE
```

While there is no dedicated **osc** to "move" (rename) a file, the desired end result can be obtained using the following procedure:

PROCEDURE 5.3: **PROCEDURE FOR MOVING A FILE WITHIN A LOCALLY CHECKED-OUT OBS PACKAGE**

1. `geeko > cp ORIGINAL_FILE NEW_FILE`

2. `geeko > osc delete ORIGINAL_FILE`

3. `geeko > osc add NEW_FILE`

4. And, finally:

```
geeko > osc status
```

to verify that the end result is as intended.

5.3 Investigating the Local Build Process

It is hard to describe a general procedure when you encounter a build error. Most build errors are very specific to the package being built. However, there are generic tools that often help:

- [Section 5.3.1, "Build Log"](#)
- [Section 5.3.2, "Local Build Root Directory"](#)

5.3.1 Build Log

Each build produces a log file on OBS. This log file can be viewed by the **buildlog** (or **bl**) subcommand. It needs a build target which is the distribution and the architecture.

For example, to view the build log of your current project for openSUSE Tumbleweed on a x86-64 architecture, use: use:

```
geeko > osc buildlog openSUSE_Tumbleweed x86_64
```

However, this command will print the complete build log which could be difficult to spot the errors. Use the **buildlogtail** subcommand to show only the end of the log file:

```
geeko > osc buildlogtail openSUSE_Tumbleweed x86_64
```

Additionally, the `osc` creates some build log files in the build directory `/var/tmp/build-root/`:

`.build.log`

Contains the log.

`.build.command`

Contains the command which is used to build the package. For RPM-like systems it is `rpmbuild -ba PACKAGE.spec`.

`.build.packages`

Contains the path to all object files.

5.3.2 Local Build Root Directory

If you build a package locally and you get a build error, investigate the problems in the build root directory directly. This is sometimes easier and more effective than only looking at the build log. By default, the directory `/var/tmp/build-root/` is used as the *build root*. This is defined in the configuration file `~/.oscrc` using the key `build-root`.

Each combination of distribution and architecture has its own build root. To change into the build root for openSUSE Tumbleweed on the x86-64 architecture, use the following command:

```
geeko > osc chroot openSUSE_Tumbleweed x86_64
```

When prompted, enter the `root` password.

Your shell will then change to the directory `/home/abuild` belonging to the user `abuild` in group `abuild`.

The build root contains the following structure:

EXAMPLE 5.2: DIRECTORY STRUCTURE OF A BUILD ROOT (`/var/tmp/build-root/`)

```
/home/abuild/
├─ rpmbuild
│   ├─ BUILD ①
│   ├─ BUILDROOT ②
│   ├─ OTHER ③
│   ├─ RPMS ④
│   │   ├─ i386
│   │   ├─ noarch
│   │   └─ x86_64
│   └─ SOURCES ⑤
```

- ❶ Contains directory named after the package name. In spec files, the name of the package directory is referenced using the `%buildroot` macro.
- ❷ If the build process was unable to create a package, this directory contains all files and directories which are installed in the target system through the `%install` section of the spec file.
If the package has been successfully built, this directory will be emptied.
- ❸ Usually contains the file `rpmlint.log`.
- ❹ If the build was successful, stores binary RPMs into subdirectories of architecture (for example, `noarch` or `x86_64`).
- ❺ All source files from the working copy will be copied here.
- ❻
- ❼ Stores source RPMs into this directory.

5.4 Adding Dependencies to Your Project

Software usually depends on other software: To run an application, you may, for example, need additional libraries. Such dependencies are called *installation requirements*.

Additionally, there are also dependencies that are only necessary for building a package but not when the software it contains is run. Such dependencies are called *build requirements*.

The Open Build Service provides the following methods to handle both dependencies in your projects:

- [Section 5.4.1, “Adding Dependencies to Your Build Recipes”](#)
- [Section 5.4.2, “Associating Other Repositories with Your Repository” \(layering\)](#)
- [Section 5.4.3, “Reusing Packages in Your Project” \(linking and aggregating\)](#)

5.4.1 Adding Dependencies to Your Build Recipes

In a spec file, dependencies are expressed with the keywords `Requires` (installation requirements) and `BuildRequires` (build requirements). Both belong to the header of the spec file.

```

Name:          foo-example
Version:       1.0.0
BuildRequires: bar
Requires:      zoo1 >= 1.5.6

```

5.4.2 Associating Other Repositories with Your Repository

There is no need to duplicate the work of others. If you need a specific package which is available in another repository, you can reference this repository in your project metadata. This is called *layering*.

When a package is needed, it can be installed from another other repository (see the note below). To add another repository that can be used as build or installation requirements, do the following:

1. Open a terminal.
2. Edit the project metadata:

```
geeko > osc meta prj --edit home:obsgeeko
```

3. Search for repository elements. For example, to allow usage packages from devel:languages:python in a openSUSE Tumbleweed project, extend the repository element with:

```

<repository name="openSUSE_Tumbleweed">
  <path project="devel:languages:python" repository="openSUSE_Factory"/>
  <path project="openSUSE:Factory" repository="standard"/>
  <arch>x86_64</arch>
</repository>

```



Note: Order Is Important

The order of the path elements is important: path elements are searched from top to bottom.

If a package cannot be found in the first repository, the second repository is considered. When the first suitable package is found, it is installed and the build preparation can continue.

For practical reasons, additional repositories should be added before the standard repositories of the specified distribution.

4. Add more path elements under the same repository element.
5. If necessary, repeat [Step 3](#) and [Step 4](#) to add path elements to repository elements of other distributions or releases.

5.4.3 Reusing Packages in Your Project

To reuse existing packages in your package repository, OBS offers two methods: “aggregating” and “linking”.

5.4.3.1 Linking a Package

A linked package is a clone of another package with additional modifications. Linking is used in the following situations:

- The source code needs changes, but the source either cannot be changed in the original package or doing so is impractical or inconvenient to change the source.
- To separate the original source from own patches.

The general syntax of the linkpac command is:

```
geeko > osc linkpac SOURCEPRJ SOURCEPAC DESTPRJ
```

For example, to link the package python-lxml from devel:language:python into your home project, use the following command:

```
geeko > osc linkpac devel:language:python python-lxml home:obsgeeko
```

In contrast to aggregating, the checkout contains all the files from the linked repository. To reduce it to a single file (like with aggregating), “unexpand” it in the working directory like this:

```
geeko > osc up --unexpand-link
Unexpanding to rev 1
A  _link
D  pytest-3.2.1.tar.gz
D  python-pytest-doc.changes
```

```
D    python-pytest-doc.spec
D    python-pytest.changes
D    python-pytest.spec
At revision 1.
```

This gives you a `_link` file similar to the `_aggregate` file. You can use the `--expand-link` option in the `up` subcommand to revert to the previous state.

5.5 Manage Group

Users with Maintainer rights can add users to their group and remove users from it. They can also give other users Maintainer rights.

```
osc api -d "<group><title><group-title></title><email><group-email></email><maintainer
  userid=<user-name>/><person><person userid=<user_name>/></person></group>' -X PUT "/
group/<group-title>"
```

6 Local Building

Every build that happens on the server can also be executed locally in the same environment using the `osc` tool. All you need is to check out the source code and run `osc build` to run the build recipe. The following explains it for RPM format, but it works for any. `osc` will download needed binaries and execute the local build.

6.1 Generic Local Build Options

Frequently, local builds are undertaken on local checkouts of source packages that already reside on an OBS server - for example, to test changes before committing them to the server.

It is also possible to trigger a local build in an arbitrary local directory containing sources, without any corresponding source package on an OBS server. (However, `osc` will still need a connection to the server in order to download build dependencies.) The following text describes what the source directory should contain, at a minimum.

Independent of the build format you need at least one source file as build description. The file name and structure is format specific. You can find some supported formats described below. To build your build format, you need:

- the original source archive. Instead of that the package may contain a `_service` file which describes how to create it, for example by downloading it or building it from a SCM repository. It can also be used to create the build description file. Find more details about it in the source service chapter.
- optional patches which changes the original source code to fix problems regarding security, the build process, or other issues
- other files which do not fall into one of the previous categories

In the typical case of source packages locally checked out from an OBS server, this is already the case. To build an existing package, the general procedure is as follows:

1. If you have not done so yet, set up your project as shown in [Section 5.1, "Setting Up Your Home Project"](#).
2. In the terminal, choose or create a directory on a local partition that has enough space to hold the package sources.

3. Check out the project that contains the package:

```
geeko > osc checkout PROJECT PACKAGE
```

This creates a `PROJECT/PACKAGE` directory in the current directory.

4. Change into the directory:

```
geeko > cd PROJECT/PACKAGE
```

5. The simplest way to run a build is just to call the `osc build` command. osc will try to detect your installed OS and build for it if possible.

```
geeko > osc build
```

However, you may also manually specify the build target. For example openSUSE Tumbleweed for x86_64, you want to create the RPM package:

```
geeko > osc build openSUSE_Tumbleweed x86_64
```

6. It will download the required dependencies and execute the build script. Therefore it needs to ask for root permissions in most cases.

Successful Build

```
[ 15s] RPMLINT report:
[ 15s] =====
[ 16s] 2 packages and 0 specfiles checked; 0 errors, 0 warnings.
[ 16s]
[ 16s]
[ 16s] venus finished "build PACKAGE.spec" at Fri Sep 1 11:54:31 UTC 2017.
[ 16s]

/var/tmp/build-root/openSUSE_Tumbleweed-x86_64/home/abuild/rpmbuild/
SRPMS/PACKAGE-VERSION-0.src.rpm

/var/tmp/build-root/openSUSE_Tumbleweed-x86_64/home/abuild/rpmbuild/RPMS/
noarch/PACKAGE-VERSION-0.noarch.rpm
```

Unsuccessful Build

```
[ 8s] venus failed "build PACKAGE.spec" at Fri Sep 1 11:58:55 UTC 2017.
[ 8s]

The buildroot was: /var/tmp/build-root/openSUSE_Tumbleweed-x86_64
```

A successful build of a spec file ends with the creation of the RPM and SRPM files.

7. For a detailed log, see the file [/var/tmp/build-root/openSUSE_Tumbleweed-x86_64/.build.log](#).

6.2 Advanced Local Build Environment Handling

The default build environment for local builds is usually chroot. While this is simplest environment and is therefore easy and fast to handle, it has also a number of shortcomings. For one it is not safe against attacks, so you must not build sources or using build dependencies from a resource which you do not trust. Furthermore the environment is not fully isolated and runs on the kernel the target distribution runs. This means esp image builds and kernel/hardware specific builds may fail or won't produce the same result. The server side is usually set to inside of KVM therefore to avoid this.

- You can also build locally in KVM (if your hardware supports it) by running

```
geeko > osc build --vm-type=kvm --vm-memory=MB
```

- Another important virtualization mode is qemu. This can be used to build for a foreign hardware architecture even when the distribution is not prepared for the qemu user land emulator. However, this qemu system emulator approach will be much slower.

```
geeko > osc build --vm-type=qemu --vm-memory=MB REPOSITORY ARCHITECTURE
```

- You may want to jump inside of a the build environment for debugging purposes. This can be done via the following command:

```
geeko > osc shell --vm-type=VM
```

- To remove the build environment, use:

```
geeko > osc wipe --vm-type=VM
```

7 Using Source Services

7.1 About Source Services

Source Services are tools to validate, generate or modify sources in a trustable way. They are designed as smallest possible tools and can be combined following the powerful idea of the classic UNIX design.

Source services allow:

- Server-side generated files are easy to identify and are not modifiable by the user. This way, other users can trust them to be generated in the documented way without modifications.
- Generated files never create merge conflicts.
- Generated files are a separate commit to the user change.
- Source services are runnable at any time without user commit.
- Source services are runnable on server and client side in the same way.
- Source services are safe. A source checkout and service run never harms the system of a user.
- Source services avoid unnecessary commits. This means there are no time-dependent changes. In case the package already contains the same file, the newly generated file is dropped.
- Source services running local or inside the build environment can get created, added and used by everybody.
- Source services running in default or server side mode must be installed by the administrator of the OBS server.
- The use of a source service can be defined per package or project wide.

For using source services you need (refer to [Example 7.1, "Structure of a `_service` File"](#)):

- An XML file named `_service`.
- A root element `services`.
- A `service` element which uses the specific service with optional parameters.

EXAMPLE 7.1: STRUCTURE OF A `_service` FILE

```
<services> ❶
  <service name="MY_SCRIPT" ❷ mode="MODE" ❸>
    <param name="PARAMETER1">PARAMETER1_VALUE</param> ❹
  </service>
</services>
```

- ❶ The root element of a `_service` file.
- ❷ The service name. The service is a script that is stored in the `/usr/lib/obs/service` directory.
- ❸ Mode of the service, see [Section 7.2, “Modes of Source Services”](#).
- ❹ One or more parameters which are passed to the script defined in ❷.

The example above will execute the script:

```
/usr/lib/obs/service/MY_SCRIPT --PARAMETER1 PARAMETER1_VALUE --outdir DIR
```

7.2 Modes of Source Services

Each source service can be used in a mode defining when it should run and how to use the result. This can be done per package or globally for an entire project.

TABLE 7.1: SOURCE SERVICE MODES

Mode	Runs remotely	Runs locally	Added File Handling
Default	After each commit	Before local build	Generated files are prefixed with <code>_service:</code>
<code>trylocal</code>	Yes	Yes	Changes are merged into commit
<code>localonly</code>	No	Yes	Changes are merged into commit
<code>serveronly</code>	Yes	No	Generated files are prefixed with <code>_service:</code> This can be useful, when the service is not available or can not work on developer workstations.

Mode	Runs remotely	Runs locally	Added File Handling
<u>build-time</u>	During each build before calling the build tool (for example, rpm-build) ^a The service package must be available for building.		
<u>manual</u>	No	Only via explicit CLI call	Exists since OBS 2.11
<u>disabled</u>	No	Only via explicit CLI call	

^a A side effect is that the service package is becoming a build dependency and must be available.

Default Mode

The default mode of a service is to always run after each commit on the server side and locally before every local build.

trylocal Mode

This mode is running the service locally. The result is committed as standard files and not named with a _service: prefix. Additionally, the service runs on the server by default. Usually the service should detect that the result is the same and skip the generated files. In case they differ, they are generated and added on the server.


localonly Mode

This mode is running the service locally. The result gets committed as standard files and not named with _service: prefix. The service is never running on the server side. It is also not possible to trigger it manually.

serveronly Mode

The serveronly mode is running the service on the server only. This can be useful, when the service is not available or can not work on developer workstations.

buildtime Mode

The service is running inside of the build job, both for local and server side builds. A side effect is that the service package is becoming a build dependency and must be available. Every user can provide and use a service this way in their projects. The generated sources are not part of the source repository, but part of the generated source packages. Note that services requiring external network access are likely to fail in this mode, because such access is not available if the build workers are running in secure mode (as is always the case at <https://build.opensuse.org> )

manual Mode,

disabled Mode

The manual mode is neither running the service locally nor on the server side by default. It can be used to temporarily disable the service but keeping the definition as part of the service definition. Or it can be used to define the way how to generate the sources and doing so by manually calling **osc service rundisabled**. The result will get committed as standard files again. NOTE: it did only exist as "disabled" before OBS 2.11, but "manual" is the better matching alias name for its usage. The osc client may do have different behaviour in future between manual and disabled.

7.3 Defining Source Services for Validation

Source services can be used to validate sources. This can be defined at different levels:

- **Per Package.** Useful when the packager wants to validate whether the downloaded sources are really from the original maintainer.
- **Per Project.** Useful for applying project-wide policies which cannot be skipped for any package.

You can validate sources using either of two methods:

- By comparing checksums and metadata of the files in your repository with checksums and metadata as recorded by the maintainer.
- Alternatively, you can download the sources from a trusted location again and verify that they did not change.

7.4 Creating Source Service Definitions

Source services are defined in the `_service` file and are either part of the package sources or used project-wide. Project-wide services are stored under the `_project` package in file `_service`.

The `_service` file contains a list of services which get called in the listed order. Each service can define a list of parameters and a mode. The project wide services get called after the per package defined services.

The `_service` file is in XML format and looks like this:

```
<services>
  <service name="download_files" mode="trylocal" />
  <service name="verify_file">
    <param name="file">krabber-1.0.tar.gz</param>
    <param name="verifier">sha256</param>
    <param
name="checksum">7f535a96a834b31ba2201a90c4d365990785dead92be02d4cf846713be938b78</param>
  </service>
</services>
```

With the example above, the services above are executed in the following order:

1. Downloads the file via the `download_files` service using the URL from the Spec file. When using `osc`, the downloaded file gets committed as part of the commit.
2. Compares the downloaded file (`krabber-1.0.tar.gz`) against the SHA256 checksum.

7.5 Removing a Source Service

Sometimes it is useful to continue working on generated files manually. In this situation the `_service` file needs to be dropped, but all generated files need to be committed as standard files. The OBS provides the `mergeservice` command for this. It can also be used via `osc` by calling `osc service merge`.

7.6 Trigger a service run via a webhook

You may want to update sources in Open Build Service whenever they change in a SCM system. You can create a token which allows to trigger a specific package update and use it via a webhook. It is recommended to create a token for a specific package and not a wildcard token. Read [Chapter 36, Authorization](#) to learn how to create a token.

7.6.1 Creating a webhook on GitLab

Go to your repository settings page in your gitlab instance. Select Integrations there. All what you need to fill is the URL

```
https://YOUR_INSTANCE/trigger/runservice
```

and the Secret Token. Hit the *Add webhook* button and you are good. You may specify project and package via CGI parameters in case you created a wildcard token:

```
https://YOUR_INSTANCE/trigger/runservice?project=PROJECT&package=PACKAGE
```

7.6.2 Creating a webhook on GitHub

Go to your repository settings page of your repository on github.com. Select Webhooks settings and create a hook via *Add Webhook* button. Define the payload URL as

```
https://YOUR_INSTANCE/trigger/webhook?id=$TOKEN_ID
```

and fill the secret box with your token string. Please note that github requires that you must also define the token id as part of the webhook string. All other settings can stay untouched and just hit the *Add webhook* button.

8 SCM/CI Workflow Integration

8.1 SCM/CI Workflow Integration Setup

8.1.1 Introduction

With this integration, you can take advantage of source code management (SCM) systems like GitHub, GitLab or Gitea to manage your packages sources. Then, you can integrate those sources with OBS to run different **workflows**, for instance, to build a package and report back the result to the SCM.

In the following sections, you will find the instructions to set up the integration between SCMs and OBS.

This chapter talks in GitHub jargon to simplify the text. As constantly mentioning all the names for the same things, e.g. *Pull Requests/Merge Requests*, is tiresome and confusing. However, every aspect has its correspondence in GitLab and Gitea. Refer to [Section 8.1.10, “Equivalence Table”](#) for clarification of terminology.

8.1.2 Prerequisites

Before you start, you need to

- have a repository on GitHub.
- have a package on an OBS Instance.

8.1.3 Supported SCMs

We support the GitHub.com and GitLab.com instances.

We also support Self-Hosted instances from GitHub, GitLab and Gitea. As long as the network connectivity works, OBS will be able to interact with that SCM.

8.1.4 Token Authentication

OBS and GitHub need to talk to each other. Tokens are the way to make this happen.

8.1.4.1 How to Authenticate OBS with SCMs

You have to create a GitHub **personal access token**. OBS is going to use it to talk to GitHub on your behalf.

The personal access token needs, at least, the following scopes assigned:

- GitHub Classic Token: repo
- GitHub Fine-Grained Token:
 - Contents: Read only
 - Commit statuses: Read and write
- GitLab: api
- Gitea: repo

Check [GitHub's](https://docs.github.com/en/github/authenticating-to-github/creating-a-personal-access-token) (https://docs.github.com/en/github/authenticating-to-github/creating-a-personal-access-token) [↗](#), [GitLab's](https://docs.gitlab.com/ee/user/profile/personal_access_tokens.html#creating-a-personal-access-token) (https://docs.gitlab.com/ee/user/profile/personal_access_tokens.html#creating-a-personal-access-token) [↗](#) and [Gitea's](https://docs.gitea.io/en-us/api-usage/#generating-and-listing-api-tokens) (https://docs.gitea.io/en-us/api-usage/#generating-and-listing-api-tokens) [↗](#) documentation to learn how.

8.1.4.2 How to Authenticate SCMs with OBS

You have to create an OBS **workflow token**. Github is going to use it to trigger actions on OBS on your behalf.

8.1.4.2.1 Create Token

You can create the OBS token via WebUI in *Profile > Manage Your Tokens*.

You can also use *osc* for this:

```
osc token --create --operation workflow --scm-token long_ascii_salad
```

Example of response:

```
<status code="ok">
  <summary>Ok</summary>
  <data name="token">long_ascii_salad</data>
  <data name="id">12345</data>
```

```
</status>
```

Make sure you replace *long_ascii_salad* with your real GitHub personal access token created in [Section 8.1.4.1, “How to Authenticate OBS with SCMs”](#)



Warning

Don't forget to keep your token secret to prevent someone else from triggering operations in your name!

8.1.4.2.2 Regenerating Secrets and Deleting Tokens

If you suspect your OBS token secret was leaked, you can regenerate the secret or delete the whole token to secure it again:

a) Regenerate the token secret

Through the WebUI in *Profile > Manage Your Tokens > Edit > Regenerate Token*.

b) Delete the token

You can always delete your token via WebUI, in *Profile > Manage Your Tokens*, or with these commands:

```
osc token # list all your tokens
```

```
osc token --delete $token_id # remove the token with the given id
```

Then you can create a new one as explained in [Section 8.1.4.2, “How to Authenticate SCMs with OBS”](#) and replace it wherever you use it.

8.1.5 Webhooks

Once OBS and GitHub are allowed to speak to each other, they can start talking via **webhooks**.

8.1.5.1 SCM Events

On a GitHub repository, events are happening all the time: a pull request is created, somebody pushes a commit, a pull request is merged etc. When you set up a webhook on GitHub, you can specify which events you are interested in. Only when those events happen, the webhook will be sent to OBS.

This is the list of SCM events supported by the existing workflows in OBS:

- Pull requests and Merge requests
- Pushes
- Tag Pushes

In addition, the Pull request events contain a field called action. OBS supports a different subset of Pull request event actions, depending on the SCM. For GitHub and Gitea, the following set of actions is supported:

- closed
- opened
- reopened
- synchronize
- synchronized

For Gitlab's Merge request events, the following actions are supported:

- close
- merge
- open
- reopen
- update

Refer to the [Equivalence Table](#) for more details or read more about GitHub events (<https://docs.github.com/en/developers/webhooks-and-events/webhooks/webhook-events-and-payloads>), GitLab events (https://docs.gitlab.com/ee/user/project/integrations/webhook_events.html) and Gitea events (<https://docs.gitea.io/en-us/webhooks/>).

8.1.5.2 How to Set Up a Webhook on Github

Go to the project you want to set the integration on, then under *Settings > Webhooks*.

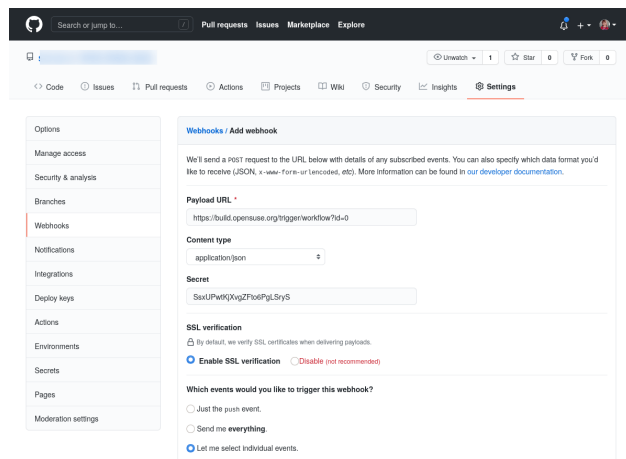


FIGURE 8.1: WEBHOOKS ON GITHUB.

You have to fill in the form with:

- **Payload URL:** `https://build.opensuse.org/trigger/workflow?id=12345`. Replace 12345 with the OBS token numerical ID previously obtained.
- **Content type:** `application/json`.
- **Secret:** `uvwxyz`. Replace uvwxyz with the OBS token secret string previously obtained.
- **Enable SSL verification.**
- **Let me select individual events:** Pull requests, Pushes.

8.1.5.3 How to Set Up a Webhook on GitLab

Go to the project you want to set the integration on, under the *Settings* > *Webhooks*.

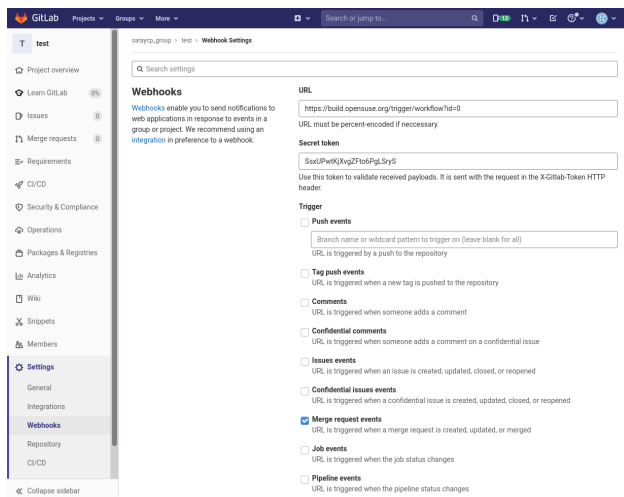


FIGURE 8.2: WEBHOOKS ON GITLAB.

Fill in the following fields:

- **URL:** *https://build.opensuse.org/trigger/workflow?id=12345*. Replace 12345 with the OBS token numerical ID previously obtained.
- **Secret Token:** *uvwxyz* Replace uvwxyz with the OBS token secret string previously obtained.
- **Trigger:** *Merge request events, Push events, Tag push events.*

8.1.5.4 How to Set Up a Webhook on Gitea

Go to the repository you want to set the integration on, then under *Settings > Webhooks*.

The screenshot shows the 'Add Webhook' interface in a GitHub repository. The 'Target URL' field is populated with 'https://build.opensuse.org/trigger/workflow?id='. The 'HTTP Method' is set to 'POST' and the 'POST Content Type' is 'application/json'. The 'Secret' field is masked with asterisks. Under 'Trigger On', 'All Events' is selected. A green 'Add Webhook' button is visible at the bottom of the form.

FIGURE 8.3: WEBHOOKS ON GITEA.

You have to fill in the form with:

- **Target URL:** `https://build.opensuse.org/trigger/workflow?id=12345`. Replace 12345 with the OBS token numerical ID previously obtained.
- **HTTP Method:** POST.
- **POST Content Type:** application/json.
- **Secret:** `uvwxyz`. Replace uvwxyz with the OBS token secret string previously obtained.
- **Custom events...:** Pull Request, Push.

8.1.6 OBS Workflows

A GitHub event occurs and OBS receives the corresponding webhook. Now is when the **OBS workflows** come into play.

A workflow is nothing else than a sequence of **steps** you want to perform in OBS. You can describe the steps to run in a YAML configuration file.

To do so, in the root directory of your GitHub repository, create a directory `.obs` which contains a file called `workflows.yml`. If you don't want to use that directory, you should check [Configuration File Location](#) out.

The content of `.obs/workflows.yml` could look like this:

```
rebuild_master:
```

```

steps:
  - rebuild_package:
      project: home:Admin
      package: ctris
filters:
  event: push

```

You can also define multiple workflows, each one needs an *unique* name. The following example contains two workflows: *main_workflow* and *rebuild_master*.

```

main_workflow:
  steps:
    - branch_package:
        source_project: OBS:Server:Unstable
        source_package: obs-server
        target_project: OBS:Server:Unstable:CI
  filters:
    event: pull_request
rebuild_master:
  steps:
    - rebuild_package:
        project: home:Admin
        package: ctris
  filters:
    event: push
    branches:
      only:
        - master

```

8.1.6.1 Configuration File Location

By default the configuration file is fetched from the repository's target branch under *.obs/workflows.yml*. You can of course adjust that by editing the token configuration in OBS. The following options are available:

- *Path for Workflows Configuration File* allows you to adjust the path to be different than the default *.obs/workflows.yml* in the code repository.
- *URL to Workflows Configuration File* allows you to use a file that is hosted in a different place than the code repository.

Your Profile / Tokens / Edit Token

Edit Token

Id: 1

Operation: Workflow

Description:

Testing token

If your token has been lost, forgotten or compromised, you can regenerate it. Don't forget to replace the string in any scripts or applications using this token.

[Regenerate Token](#)

SCM token:

Please enter your new SCM token

Leaving this text field empty will not update your SCM token.

Path for Workflows Configuration File:

ci/my_workflows.yml

The default path is '.obs/workflows.yml'.

URL to Workflows Configuration File:

Eg: https://example.com/my_subdir/my_workflows_file.yml

When the URL is given, the path will be ignored. The URL should be accessible for the OBS instance.

[Cancel](#) [Update](#)

FIGURE 8.4: CONFIGURATION FILE LOCATION OPTIONS IN EDIT TOKEN

8.1.6.2 OBS Workflow Steps

We support the following steps (the keys used in the configuration file appears surrounded with parenthesis):

- Branch a package in a project (branch_package).
- Submit a request (submit_request).
- Link a package to a project (link_package).
- Configure repositories/architectures for a project (configure_repositories)
- Rebuild a package (rebuild_package)
- Set flags for projects, packages, repositories or architectures (set_flags)
- Trigger services of a package (trigger_services)



Warning

The user the token belongs to needs to have permissions to branch a package, link packages, configure repositories/architectures, rebuild packages and trigger services of a package.

8.1.6.2.1 Branch a Package in a Project

Given we have a source package called *ctris* coming from a source project called *games*, and a target project called *home:jane*, this step will branch that package onto the target project, keeping in mind that:

- With a pull request event, it will go to e.g.: *home:jane:github:jane:ctris:PR-1/ctris*. *PR-1* being the pull request number.
- With a push event for commits, it will go to e.g.: *home:jane/ctris-66f2acfbded89a19935ee6d481b7cf2ab95427f6*. *66f2acfbded89a19935ee6d481b7cf2ab95427f6* being the SHA of the latest commit that triggered the event.
- With a push event for tags, it will go to e.g.: *home:jane/ctris-release_1*. *release_1* being the name of the tag that triggered the event.

This is an example of a configuration file with a branch package step:

```
workflow:
  steps:
    - branch_package:
        source_project: games
        source_package: ctris
        target_project: home:jane
```

Branching a package into a project that did not exist before, for instance for a pull request event, will branch the package and set up the same repositories that the source project has. If you want to skip this and set up repositories yourself, with the `configure_repositories` step, set the `add_repositories` key to anything else than *enabled*.

8.1.6.2.2 Submit a Request

The submit request step is the equivalent of the *osc submitrequest* command.

The requirements to run a submit request step are:

- There has to be a source package.
- There has to be some changes between the source package and the target package.

After the previous requirements are met, keep in mind that:

- With a pull request open event, push event, or tag_push event, it will create the submit request.
- When more commits are added to the pull request, it will supersede the request it previously created with a new request.
- With a pull_request closed event, it will revoke the request.

This is an example of a configuration file with a submit request step:

```
workflow:
  steps:
    - submit_request:
        source_project: games
        source_package: ctris
        target_project: home:jane_doe
        target_package: ctris                    # (optional, uses source_package if
not set)
        description: 'Check out this cool package' # (optional, uses the commit/pull
request message if not set)
```

8.1.6.2.3 Link a Package to a Project

The link package step is the equivalent of *osc linkpac* command.

Given a source project called *devel*, a source package called *gcc*, a target project called *home:jane*, and a GitHub fork called *jane/gcc* the step will link the package *devel/gcc* against:

- *home:jane:github:jane:gcc:PR-1/gcc* for a pull request event. *PR-1* being the pull request number.
- *home:jane/gcc-fae00a0ac0e5687343a60ae02bf60352002ab9aa* with a push event for commits. *fae00a0ac0e5687343a60ae02bf60352002ab9aa* being the SHA of the latest commit that triggered the event.
- *home:jane/gcc-release_1* with a push event for tags. *release_1* being the name of the tag that triggered the event.

This is an example of a configuration file with a link package step:

```
workflow:
```

```
steps:
  - link_package:
      source_project: devel
      source_package: gcc
      target_project: home:jane
```



Note

If you rely on *Using Source Services* to run, for instance to pick up changes from a PR with the *obs_scm* service, you can't make use of this step. Package links do not run the services. Use the *branch_package* step instead.

8.1.6.2.4 Configure Repositories/Architectures for a Project

Given a project called *home:jane*, the step will configure a number of repositories and architectures for:

- *home:jane:jane_github:repo123:PR-1* when the event is a pull request. *jane_github* being the username/organization which owns the SCM repository. *repo123* being the name of the SCM repository. *PR-1* being the pull request number.
- *home:jane* when the event is a commit or tag push.

Each repository needs:

- a name, e.g.: *openSUSE_Tumbleweed*
- a list of paths, each having a target project (e.g: *openSUSE:Factory*) and target repository (e.g: *snapshot*)
- a list of architectures to be defined for each repository. e.g.: *x86_64* and *i586*

This is an example of a configuration file with a configure repositories step:

```
workflow:
  steps:
    - configure_repositories:
        project: home:jane
        repositories:
          - name: openSUSE_Tumbleweed
            paths:
```

```

    - target_project: openSUSE:Factory
      target_repository: snapshot
    - target_project: openSUSE:Tumbleweed
      target_repository: standard
  architectures:
    - x86_64
    - i586
- name: openSUSE_Leap_15.2
  paths:
    - target_project: openSUSE:Leap:15.2
      target_repository: standard
  architectures:
    - x86_64

```

8.1.6.2.5 Rebuild a Package

Given a project called *home:Admin* and a package *ctris*, the step will rebuild the package *home:Admin/ctris*.

This is an example of a configuration file with a rebuild package step.

```

workflow:
  steps:
    - rebuild_package:
        project: home:Admin
        package: ctris

```

8.1.6.2.6 Set Flags for Projects, Packages, Repositories or Architectures

There are OBS-wide defaults for each flag type. This step is only necessary if you want to diverge from the defaults (see [Valid flag types](#)).

Providing the type *build*, the status *enable* and the project *home:Admin*, OBS will enable *all* builds:

- for the *home:Admin:\$MY_SCM_ORG:\$MY_SCM_PROJECT:PR-\$MY_PR_NUMBER* project when the webhook event is a pull request.
- for the *home:Admin* project when the webhook event is a push.

Providing multiple flags is supported as noted in the configuration file below:

```

workflow:
  steps:

```

```
- set_flags:
  flags:
    - type: build
      status: enable
      project: home:Admin
    - type: publish
      status: disable
      project: home:Admin
```

The type, status and project keys are always required. Optional keys are also available to limit the flag to a package, repository or architecture.

The project, package, repository and architecture should exist before a flag is set for them. They can be created in steps preceding a *set_flags* step, although this isn't necessary as long as they exist.

The type has to be one of the following values:

VALID FLAG TYPES

- lock (default status *disable*)
- build (default status *enable*)
- publish (default status *enable*)
- debuginfo (default status *disable*)
- useforbuild (default status *enable*)
- binarydownload (default status *enable*)
- sourceaccess (default status *enable*)
- access (default status *enable*)

The status is either *disable* or *enable*.

Take into consideration, that if the *set_flags* step doesn't define a flag specifically, a flag which had been set previously will preserve its value.

So with the configuration file provided below and a *pull request event*, builds of the *home:Admin:\$MY_SCM_ORG:\$MY_SCM_PROJECT:PR-\$MY_PR_NUMBER/ctris* package will be disabled for the *openSUSE_Tumbleweed* repository and *x86_64* architecture. For a *push event*, it's exactly the same, except for the package which is *home:Admin/ctris-\$MY_COMMIT_SHA_OR_TAG_NAME*.

```
workflow:
```


```

steps:
  - set_flags:
      flags:
        - type: build
          status: disable
          project: home:Admin
          package: ctris
          repository: openSUSE_Tumbleweed
          architecture: x86_64

```

8.1.6.2.7 Trigger Services of a Package

Given a project called *home:Admin* and a package *ctris*, the step will trigger services of the package *home:Admin/ctris*.

Be sure to have a `_service` (https://openbuildservice.org/help/manuals/obs-user-guide/cha.obs.source_service.html)  file in the package *home:Admin/ctris*.

This is an example of a configuration file with a trigger services step:

```

workflow:
  steps:
    - trigger_services:
        project: home:Admin
        package: ctris

```

8.1.6.3 Filters

You can customize when workflows run by declaring **branch** or **Event** filters. They will make workflows run or not for specific branches/events.

You can define them in the configuration file `.obs/workflows.yml`. Here's an example:

```

workflow:
  steps:
    - branch_package:
        source_project: home:jane_doe
        source_package: ctris
        target_project: games
  filters:
    event: pull_request
    branches:
      only:
        - master

```

- staging

8.1.6.3.1 Filters Delimiters: **only** and **ignore**

Some steps can affect a group of elements (branches) You can use filter delimiters like **only** and **ignore** to specify which elements should be affected, or not, by the step.

The available filters delimiters are:

- **only**: the step only affects the elements in the list.
- **ignore**: the step affects all the elements except those in the list.



Note

only has precedence over **ignore**, so if both are defined, **ignore** is not considered.

This is an example to run a workflow *only* for the target branches *master*:

```
workflow:
  steps:
    - rebuild_package:
        project: games
        package: ctris
  filters:
    branches:
      only:
        - master
```

This is an example to run a workflow for all the target branches *except* for the branch *staging*:

```
workflow:
  steps:
    - rebuild_package:
        project: games
        package: ctris
  filters:
    branches:
      ignore:
        - staging
```


8.1.6.3.2 Event Filter

Setting an event filter will run the workflow only for this event. The event filter doesn't accept multiple events. Documentation on the SCM events can be found here: [Section 8.1.5.1, "SCM Events"](#).

The available event filters are:

- **pull_request** is for pull request events.
- **merge_request** is an alias for the 'pull_request' event. Introduced with workflow version 1.1 (also see [Section 8.3.1, "Workflow Version Table"](#)). .
- **push** is for push events related to commits.
- **tag_push** is for push events related to tags.

The following is an example to run a workflow only for a pull request event:

```
workflow:
  steps:
    - branch_package:
        source_project: games
        source_package: ctris
        target_project: home:jane_doe
  filters:
    event: pull_request
```

8.1.6.3.3 Branches Filter

Matches target branches based on their names and runs a workflow only for those branches.

This is an example to run a workflow for all target branches, except *master* and *final*:

```
workflow:
  steps:
    - branch_package:
        source_project: home:jane_doe
        source_package: ctris
        target_project: games
  filters:
    branches:
      ignore:
        - master
        - final
```

Learn more about [Filters Delimiters: only and ignore](#).



Note

tag_push events are not supported by the **branches** filter.

8.1.6.4 Placeholder Variables

With placeholder variables, workflows are now dynamic. Whenever a webhook event comes in, OBS downloads the workflows file and parses it. This is when the placeholder variables are replaced by the data they refer to in the webhook event payload.

Here's a list of supported placeholder variables and their mapping:

- `%{SCM_ORGANIZATION_NAME}`: The name of the SCM organization, like *openSUSE* for the GitHub repository *openSUSE/open-build-service*.
- `%{SCM_REPOSITORY_NAME}`: The name of the SCM repository, like *open-build-service* for the GitHub repository *openSUSE/open-build-service*.
- `%{SCM_PR_NUMBER}`: The number of the pull/merge request from which the webhook event originates. This placeholder variable should be defined in workflows running only for pull request webhook events.
- `%{SCM_COMMIT_SHA}`: The SHA of the commit from which the webhook event originates.

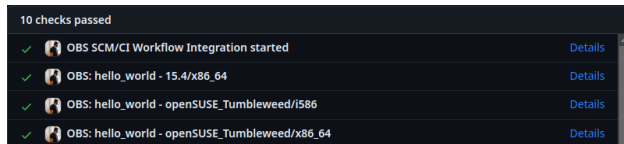
Below is an example of a workflow with a placeholder variable:

```
# The test_build workflow will branch a package based on the SCM repository name from
which the webhook event came from.
test_build:
  steps:
    - branch_package:
        source_project: games
        source_package: %{SCM_REPOSITORY_NAME}
        target_project: games:CI
  filters:
    event: pull_request
```

For a more in-depth example in combination with configuration file location, refer to [Section 8.4.6, "Using a Custom Configuration File URL in Combination with Placeholder Variables"](#).

8.1.7 Status Reporting

Once all the steps in the workflow are done, OBS will report the build results back to GitHub. OBS will show detailed package build status for each distribution and architecture you have set up in the configuration file.



10 checks passed	
✓ OBS SCM/CI Workflow Integration started	Details
✓ OBS: hello_world - 15.4/x86_64	Details
✓ OBS: hello_world - openSUSE_Tumbleweed/i586	Details
✓ OBS: hello_world - openSUSE_Tumbleweed/x86_64	Details

FIGURE 8.5: BUILD STATUS

Moreover, if your package builds several multibuild flavors, the status will have the [flavor](https://openbuildservice.org/help/manuals/obs-user-guide/cha.obs.multibuild.html) (<https://openbuildservice.org/help/manuals/obs-user-guide/cha.obs.multibuild.html>) [↗](#) name appended to the package name:



✓ OBS SCM/CI Workflow Integration started	Details
✓ OBS: hello_world:bar - 15.4/x86_64	Details
✓ OBS: hello_world:bar - openSUSE_Tumbleweed/i586	Details
✓ OBS: hello_world:bar - openSUSE_Tumbleweed/x86_64	Details
✓ OBS: hello_world:foo - 15.4/x86_64	Details
✓ OBS: hello_world:foo - openSUSE_Tumbleweed/i586	Details
✓ OBS: hello_world:foo - openSUSE_Tumbleweed/x86_64	Details

FIGURE 8.6: BUILD STATUS FOR SEVERAL MULTIBUILD FLAVORS



Note

Due to a limitation, the **initial** "pending" build status of packages with multibuild flavors is **not** reported. The build status for those flavors will however still be reported when the build finishes.

8.1.8 Workflow Runs

For every SCM event, OBS compiles relevant information about the workflows running on the system. You can find the so-called "Workflow Runs" under the list of tokens.

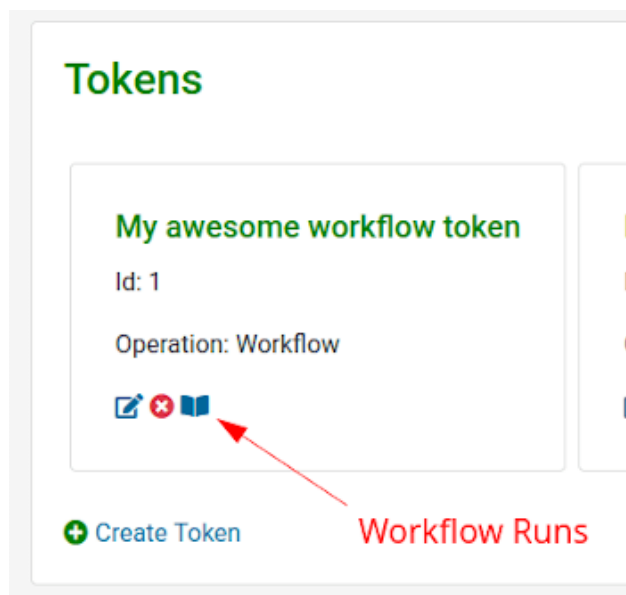


FIGURE 8.7: TOKENS LIST

From the list of workflow runs, you can get information like:

- the status (running/fail/success) represented by icons,
- the type of event and action,
- links to the SCM repository, pull/merge request or commit involved,
- the time when the workflow was triggered

Your Profile / Tokens / Workflow Runs

Workflow Runs

Displaying workflow runs 41 - 47 of 47

✓	Push event	saraycp/hello_world e33abcb9e29f1e1540f6e88e9e2ce2553460c	2022-02-02 20:48:26 UTC
✱	Pull request event opened	saraycp/hello_world #61	2022-02-02 20:45:00 UTC
✱	Pull request event opened	saraycp/hello_world #61	2022-02-02 20:43:24 UTC
✓	Pull request event opened	saraycp/hello_world #61	2022-02-01 14:19:46 UTC
⚠	Create event	saraycp/hello_world Unknown source	2022-02-01 14:17:20 UTC
✓	Push event	saraycp/hello_world f4c677c70f9789251b08968aecd6b2937b3085	2022-02-01 14:17:19 UTC
✓	Push event	saraycp/hello_world acae19496ad502f53d8824feca77b1cdc7afe67	2022-02-01 14:14:08 UTC

First Previous 1 2 3

FIGURE 8.8: WORKFLOW RUNS

Click on each workflow run to get detailed information about it. OBS records the request received from the SCM,

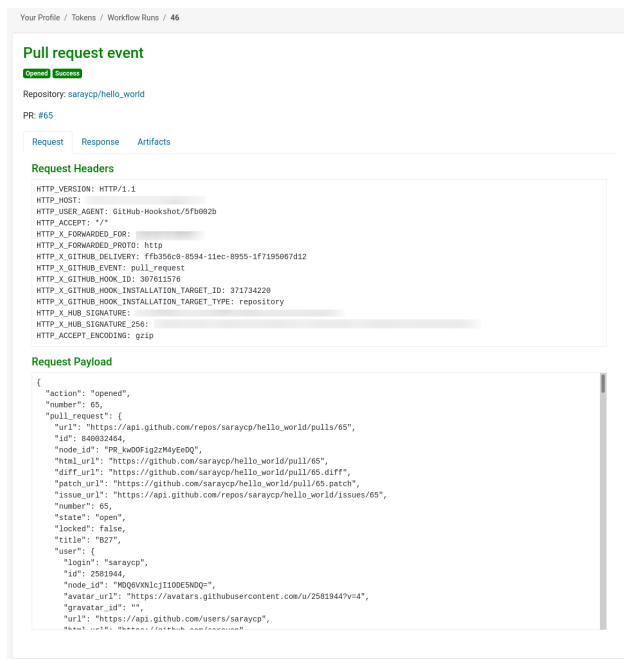


FIGURE 8.9: WORKFLOW RUNS - REQUEST

the response sent back to the SCM,

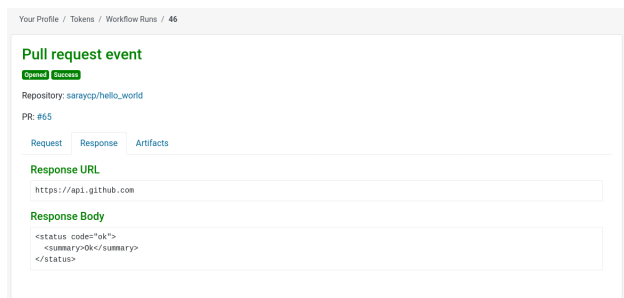


FIGURE 8.10: WORKFLOW RUNS - RESPONSE

and the artifacts used or generated during the run of each workflow.




FIGURE 8.11: WORKFLOW RUNS - ARTIFACTS

These records will help with debugging workflows. If an error occurs in any of the workflow steps, the workflow run will record error messages. And reading the artifacts will help to understand what happened behind the scenes.

8.1.9 Errors

TABLE 8.1: COMMON ERRORS

Error	Reason
<i>"Project not found"</i>	Make sure the projects you declared in the <code>.obs/workflows.yml</code> file exist in your OBS instance.
<i>"Package not found"</i>	Make sure the packages you declared in the <code>.obs/workflows.yml</code> file exist in your OBS instance.
No build result updates are displayed in your PR/MR	Make sure there are repositories defined on your source project. Another reason can be that the build did not start because your package is "unresolvable" or "broken".
The project in OBS doesn't get updated with the latest changes in the SCM.	For certain steps you need to set up a <code>_service</code> file. Follow the obs-service-tar_scm (https://github.com/openSUSE/obs-service-tar_scm#user-documentation)  documentation.

8.1.10 Equivalence Table

TABLE 8.2: EQUIVALENCE TABLE

GitHub	GitLab	Gitea
Repository	Project	Repository
Pull request	Merge request	Pull request
PR	MR	PR

GitHub	GitLab	Gitea
Push	Push Hook	Push
Pull requests (in webhook configuration)	Merge request events (in webhook configuration)	Pull Request (in webhook configuration)
Pushes (in webhook configuration)	Push events (in webhook configuration)	Push (in webhook configuration)

8.2 SCM/CI Workflow Steps Reference Table

For each step, this table shows which event on the SCM will trigger which operations on the OBS.

TABLE 8.3: WORKFLOW STEPS REFERENCE TABLE

SCM event and action	branch_package step	submit_request step	link_package step	re-build_package step	trigger_services step	configure_repositories step	set_flags step
Pull Request opened	The <code>%{source_package}</code> will be branched to <code>%{tar-get_project}:PR-%{SCM_PR_NUM-BER}/%{source_package}</code> . The <code>%{SCM_COMMIT_SHA}</code> will be updated in the file <code>_branch_report-</code>	Create a <code>subject</code> from <code>%{source_package}</code> to <code>%{tar-get_project}:PR-%{SCM_PR_NUM-BER}/%{source_package}</code> . The request status changes will be report-	The <code>%{source_package}</code> will be linked to <code>%{tar-get_project}:PR-%{SCM_PR_NUM-BER}/%{source_package}</code> . The <code>%{SCM_COMMIT_SHA}</code> will be updated in the <code>scmsync</code> attribute	The <code>%{source_package}</code> will be re-build. The build results will be reported back to the <code>%{SCM_COMMIT_SHA}</code> as commit status.	The services of <code>%{project}</code> will be triggered. The build results will be reported back to the <code>%{SCM_COMMIT_SHA}</code> as com-	The repository will be configured for the <code>%{tar-get_project}:PR-%{SCM_PR_NUM-BER}</code> project. Nothing will be reported to the SCM.	The flags will be configured for the <code>%{tar-get_project}:PR-%{SCM_PR_NUM-BER}/%{source_package}</code> . Nothing will be reported to the SCM.

SCM event and action	branch_package step	submit_request step	link_package step	re-build_package step	trigger_services step	configure_repositories step	set_flags step
	quest or in the scmsync attribute of the package. This will trigger a run of the services, which will trig- ger a re- build. The build results of the branched package will be report- ed to the %{SCM_COM- MIT_SHA} as com- mit sta- tus.	ed to the %{SCM_COM- MIT_SHA} as com- mit sta- tus.	of the package. This will trigger a run of the services, which will trig- ger a re- build. The build results of the linked package will will be report- ed to the %{SCM_COM- MIT_SHA} as com- mit sta- tus.		mit sta- tus.		
Pull Re- quest up- dated	The %{SCM_COM- MIT_SHA} will be	Super- sede the request it previous-	The %{SCM_COM- MIT_SHA} will be	The %{project}- %{pack- age} will	The ser- vices of %{project}- %{pack- age}	The reposito- ries will be con-	The flags will be config- ured

SCM event and action	branch_package step	submit_request step	link_package step	re-build_package step	trigger_services step	configure_repositories step	set_flags step
	updated in the file <u>branch_request</u> or in the <u>scmsync</u> attribute of the package <u>{tar-get_project}</u> BER}/{source_package} to <u>{tar-get_project}</u> BER}/{source_package} This will trigger a run of the services, which will trigger a re-build. The build results will be reported back to the <u>{SCM_COMMIT_SHA}</u> as com-	ly created with a new request from <u>{source_package}</u> to <u>{tar-get_project}</u> BER}/{source_package} The request status changes will be reported to the <u>{SCM_COMMIT_SHA}</u> as com-	updated in the <u>scmsync</u> attribute of the package <u>{tar-get_project}</u> BER}/{source_package} trigger a run of the services, which will trigger a re-build. The build results will be reported back to the <u>{SCM_COMMIT_SHA}</u> as com-	be re-build. The build results will be reported back to the <u>{SCM_COMMIT_SHA}</u> as com-	age} will be triggered. The build results will be reported to the <u>{SCM_COMMIT_SHA}</u> as com-	figured for the <u>{tar-get_project}</u> BER} project. Nothing will be reported to the SCM.	for the <u>{tar-get_project}</u> :PR-{SCM_PR_NUMBER} age}. Nothing will be reported to the SCM.

SCM event and action	branch_package step	submit_request step	link_package step	re-build_package step	trigger_services step	configure_repositories step	set_flags step
	mit status.						
Pull Request closed	The project <code>%{tar-get_project}</code> will be deleted.	The submit request will be voked.	The project <code>%{tar-get_project}</code> will be deleted.	This event is ignored.	This event is ignored.	This event is ignored.	This event is ignored.
Pull Request reopened	The project <code>%{tar-get_project}</code> will be re-stored. This will trigger a rebuild of the contained packages. The build results of the branched package will be reported to the	Create a submit request <code>%{source_package}</code> to <code>%{tar-get_project}</code> . The request status changes will be reported to the <code>%{SCM_COMMIT_SHA}</code> as com-	The project <code>%{tar-get_project}</code> will be re-stored. This will trigger a rebuild of the contained packages. The build results of the branched package will be reported to the	This event is ignored.	This event is ignored.	This event is ignored.	This event is ignored.

SCM event and action	branch_package step	submit_request step	link_package step	re-build_package step	trigger_services step	configure_repositories step	set_flags step
	<u>%{SCM_COMMIT_SHA}</u> as commit status.	mit status.	<u>%{SCM_COMMIT_SHA}</u> as commit status.				
Push	The <u>%{source_package}</u> will be branched to <u>%{tar-get_project_dir}</u> - <u>%{SCM_COMMIT_SHA}</u> . The build results of the branched package will be reported to the <u>%{SCM_COMMIT_SHA}</u> as commit status.	Create a <u>%{source_package}</u> request from <u>%{source_package}</u> to <u>%{tar-get_project_dir}</u> - <u>%{SCM_COMMIT_SHA}</u> . The request status changes will be reported to the <u>%{SCM_COMMIT_SHA}</u> as commit status.	The <u>%{source_package}</u> will be linked to <u>%{tar-get_project_dir}</u> - <u>%{SCM_COMMIT_SHA}</u> . The build results of the linked package will be reported to the <u>%{SCM_COMMIT_SHA}</u> as commit status.	The <u>%{project_package}</u> will be re-build. The build results of the <u>%{SCM_COMMIT_SHA}</u> reported back to the <u>%{SCM_COMMIT_SHA}</u> as commit status.	The services of <u>%{project_package}</u> will be triggered. The build results will be reported back to the <u>%{SCM_COMMIT_SHA}</u> as commit status.	The repositories will be configured for the <u>%{project_package}</u> . Nothing will be reported to the SCM.	The flags will be configured for the <u>%{project_package}</u> . Nothing will be reported to the SCM.

SCM event and action	branch_package step	submit_request step	link_package step	re-build_package step	trigger_services step	configure_repositories step	set_flags step
Tag Push	The <code>%{source_project}/%{source_package}</code> will be branched to <code>%{tar-get_project}-%{TAG_NAME}</code> . Nothing will be reported to the SCM.	Create a subject from <code>%{source_package}</code> to <code>%{tar-get_project}-%{TAG_NAME}</code> .	The <code>%{source_package}</code> will be linked to <code>%{tar-get_project}-%{TAG_NAME}</code> . Nothing will be reported to the SCM.	The <code>%{source_package}</code> will be re-build. Nothing will be reported to the SCM.	The services of <code>%{project}</code> will be triggered. Nothing will be reported to the SCM.	The repositories will be configured for the <code>%{project}-%{package}</code> . Nothing will be reported to the SCM.	The flags will be configured for the <code>%{project}-%{package}</code> . Nothing will be reported to the SCM.

8.3 SCM/CI Workflow Versions

To secure the compatibility of SCM/CI workflows with new features and changes, we are introducing those through new versions. We specify them with a **MAJOR.MINOR** versioning scheme. We introduce breaking, non-backward compatible features and changes with major version updates. Minor updates only include backward compatible updates, but might require adjustments to the workflows in order to benefit from new features. The workflow version is specified on the toplevel of your workflow configuration yaml file. Right now we don't enforce to specify a version in the workflow yaml configuration and default to the latest minor version.

```
version: '1.0'
workflow:
  steps:
    - link_package:
        source_project: GNOME:Factory
        source_package: gnome-shell
```

```
target_project: home:jane:playground
```

8.3.1 Workflow Version Table

Current available workflow versions and the introduced changes can be found in the versions table below.


TABLE 8.4: WORKFLOW VERSIONS

Version	Changes
1.1	Add alias for 'merge_request' to event filters. Previously we only supported the term 'pull_request'.

8.4 SCM/CI Workflow Integration Use-Cases

8.4.1 OBS SCM Service

For some of the use cases, you might want the OBS package to get the sources from the pull request in GitHub.

For this, you can make use of the existing `obs-service-tar_scm` (https://github.com/openSUSE/obs-service-tar_scm#user-documentation)  service. Your package should include a properly defined `_service` file. `obs-service-tar_scm` will automatically use the sources of the pull request that triggered it.

8.4.2 Test Build a Package For Every Pull Request on GitHub

You decide to manage your package sources from a GitHub repository. However, every time someone tries to add changes to your sources by opening a pull request, you need to verify that your package still builds for certain repositories and architectures in OBS. You can have the best of both worlds with the *SCM/CI Workflow Integration Setup*.

You will need:

- A project in OBS that you own, it will be the *target project*. Let's say: *home:jane:playground*.
- A package in OBS that you want to test build, it will be the *source package* inside the source project. E.g.: *GNOME:Factory/gnome-shell*.
- A repository in GitHub with the source code that will receive the pull requests, e.g.: *https://github.com/GNOME/gnome-shell*.
- The required tokens to allow OBS and GitHub talk each other as explained in [Section 8.1.4, "Token Authentication"](#)
- The required webhooks so GitHub notifies OBS of any event as explained in [Section 8.1.5, "Webhooks"](#)

This is obviously a good candidate to use the [OBS SCM Service](#).

There are two different strategies to do this: **branching** the package or **linking** to it.

8.4.2.1 Branch

If you decide to branch the package for the test build, the configuration file should be something like this:

```
workflow:
  steps:
    - branch_package:
        source_project: GNOME:Factory
        source_package: gnome-shell
        target_project: home:jane:playground
  filters:
    event: pull_request
```

Whenever someone opens a new pull request in the repository, OBS will branch the *source package* onto the *target project*, trigger the build, and report the results in the pull request's status checks.

Keep in mind that, when OBS branches a package, it copies the repositories from the *source project* to the *target project*, so everything is ready to start building.

Once the pull request is accepted or closed, the branched package will be deleted.

Read [SCM/CI Workflow Integration Setup](#) and, specifically, the workflow steps ([Section 8.1.6.2, "OBS Workflow Steps"](#)).

8.4.2.2 Link and Configure Repositories

If you prefer to link the package for the test build, the configuration file should be something like this:

```
workflow:
  steps:
    - link_package:
        source_project: GNOME:Factory
        source_package: gnome-shell
        target_project: home:jane:playground
    - configure_repositories:
        project: home:jane:playground
        repositories:
          - name: openSUSE_Tumbleweed
            paths:
              - target_project: openSUSE:Factory
                target_repository: snapshot
            architectures:
              - x86_64
              - i586
          - name: openSUSE_Leap_15.2
            paths:
              - target_project: openSUSE:Leap:15.2
                target_repository: standard
            architectures:
              - x86_64
  filters:
    event: pull_request
```

Whenever someone opens a new pull request in the repository, OBS will create a *target package* linked to the *source package*.

Unlike the branching, in this case the repositories are not copied to the *target project*. That is why you need to set up the *configure_repositories* step giving you the flexibility to decide which repositories are you interested in.

Read [SCM/CI Workflow Integration Setup](#) and, specifically, the workflow steps ([Section 8.1.6.2, “OBS Workflow Steps”](#)).

8.4.3 Rebuild a Package for Every Change in a Branch

You have a test build set up and you want it to keep up to date with the new changes you add to the PR. One way to do it, is to configure a rebuild package step with a push event filter.

You need:

- A project and package to test build. E.g.: *home:jane/rust*
- A repository in GitHub with an opened PR. E.g.: *https://github.com/jane/rust/pull/1*
- The required tokens to allow OBS and GitHub talk each other as explained in [Section 8.1.4, “Token Authentication”](#)
- The required webhooks so GitHub notifies OBS of any event as explained in [Section 8.1.5, “Webhooks”](#)
- The source code synchronization setup with the [OBS SCM Service](#).

The workflow configuration should be like this one:

```
workflow:
  steps:
    - rebuild_package:
        project: home:jane
        package: rust
  filters:
    event: push
```

8.4.4 Set Flags on a Package to Disable Builds for an Architecture

When you branch a package, all its repositories and their architectures will be copied over. For this package, you might want to disable builds for a certain repository or architecture. This is possible with the *set_flags* step.

The workflow configuration should be like this one:

```
workflow:
  steps:
    - branch_package:
        source_project: home:jane_doe
        source_package: rust
        target_project: home:jane_doe:CI
    - set_flags:
        flags:
          - type: build
            status: disable
            project: home:jane_doe:CI
            package: rust
```


8.4.5 Create Package on OBS for Every Software Release With Git Tags

You have a software project for which you mark releases with Git tags. For every release, you want to create a package on OBS. This can be automated in a workflow with the *branch_package* step and the *tag_push* event filter. Once the workflow is in place, every tag you push to your SCM repository will branch a package on OBS and create, then build a package for the source code associated to the tag's commit. This way, your users can always install a versioned release of your software project. You can also link one of those versioned releases to another project on OBS if you need it as a dependency.

After the usual setup for OBS workflows with tokens and webhooks (see [Section 8.1, “SCM/CI Workflow Integration Setup”](#)), you will need:

- A package in OBS that you own, and for which you want to create releases. It will be the *source package* (e.g.: *home:jane/my_package*) and it will contain a *_service* file. When this package is branched by the *branch_package* step, the branched package name will end with the name of the tag which was pushed (e.g.: *my_package-1.0*).
- A project in OBS that you own, and which will contain all packages created by the *branch_package* step. It will be the *target project* (e.g.: *home:jane:releases*).
- A SCM repository for your software project with the source code and spec file in which you will create Git tags to mark releases, e.g.: https://github.com/jane/my_package.

The workflow configuration should be like this one:

```
workflow:
  steps:
    - branch_package:
        source_project: home:jane
        source_package: my_package
        target_project: home:jane:releases
  filters:
    event: tag_push
```

The *_service* file in your source package should be like:

```
<?xml version="1.0"?>
<services>
```

```

<service name="obs_scm">
  <param name="versionformat">@PARENT_TAG@</param>
  <param name="url">https://github.com/jane/my_package.git</param>
  <param name="scm">git</param>
  <param name="revision">@PARENT_TAG@</param>
  <param name="extract">my_package.spec</param>
</service>
<service name="set_version"/>
<service name="tar" mode="buildtime"/>
</services>

```

Here's an explanation of the services involved:

- The `obs_scm` (https://github.com/openSUSE/obs-service-tar_scm#obs_scm) service fetches the source code from your SCM repository for a specific revision, which in this case, is for the latest tag (`@PARENT_TAG@`). Don't forget to extract the spec file with a `extract` element.
Set the `versionformat` to match how you want to name your releases. This is typically the Git tag name, so `@PARENT_TAG@` is what you should use. For other options, refer to [git log](https://git-scm.com/docs/git-log) (<https://git-scm.com/docs/git-log>) and its `format: <format-string>` documentation.
- The `set_version` (https://github.com/openSUSE/obs-service-set_version) service updates the `Version` directive in the spec file downloaded by the `obs_scm` service. The version format comes from the `versionformat` element under the `obs_scm` service.
- The `tar` (https://github.com/openSUSE/obs-service-tar_scm#tar) service creates a tarball out of the source code fetched by the `obs_scm` service.

For the spec file in your SCM repository, pay attention to this:

- The `Source0` directive is based on values you provided to the `obs_scm` service in the `_service` file and it should be like this: `my_package-%{version}.tar`.
The first part is the SCM repository name (e.g: `my_package`). The second part is the version macro which will be expanded to match what you defined in the `versionformat` of the `obs_scm` service in the `_service` file. The third part is the archive extension (`.tar`) since a tarball was created by the `tar` service.
- Under the `%prep` directive, you might have to update the `%setup` directive if your source package name doesn't match the name of your SCM repository. Here's how, with `my_package` being the SCM repository name:

```
%prep
```

```
%setup -q -n my_package-%version
```

8.4.6 Using a Custom Configuration File URL in Combination with Placeholder Variables

It may happen that you have multiple repositories following the same set of workflow steps, and you would rather have one copy of the configuration file stored in a single place and applied to multiple workflows. This can be done with the combination of placeholder variables and the configuration file url setting

Let's say you have the following configuration file that works with your SCM repository called *gnome-shell*

```
workflow:
  steps:
    - branch_package:
        source_project: "test-project"
        source_package: "gnome-shell"
        target_project: "test-target-project"
  filters:
    event: pull_request
```

If you replace *gnome-shell* with `%{SCM_REPOSITORY_NAME}` like so:

```
workflow:
  steps:
    - branch_package:
        source_project: "test-project"
        source_package: "%{SCM_REPOSITORY_NAME}"
        target_project: "test-target-project"
  filters:
    event: pull_request
```

It will perform just as well as it did before, however now this configuration can be applied to any other OBS package in the *test-project* and SCM repository combination assuming they have the same name.

From here the only thing left to do would be to host this file somewhere where OBS can access it, creating a workflow token and the corresponding webhooks (following the setup instructions at [Section 8.1.4, "Token Authentication"](#)) for every SCM repository you want this configuration file to apply to, making sure you set the correct configuration url (see [Section 8.1.6.1, "Configuration File Location"](#)).

There are many other ways to use these two features in parallel, make sure to read [Section 8.1.6.4, “Placeholder Variables”](#) and [Section 8.1.6.1, “Configuration File Location”](#) to get some inspiration on how you can use them in your project.

9 Staging Workflow

9.1 Working with Staging Projects

This API provides an easy way to get information about a single or all staging projects like state, requests and checks. Note: To use this API, you first need to setup a staging workflow for a project.

9.1.1 Overview of All Staging Projects

This endpoint provides an overview of all staging projects for a certain project.

```
geeko > osc api '/staging/openSUSE:Factory/staging_projects/'
```

Which will return a simple list of staging projects:

```
<staging_projects>
  <staging_project name="openSUSE:Factory:Staging:A"/>
  <staging_project name="openSUSE:Factory:Staging:B"/>
</staging_projects>
```

The returned XML can include more information by adding any combination of this three parameters: requests, status and history. This example combines requests and status:

```
geeko > osc api '/staging/openSUSE:Factory/staging_projects/?requests=1&status=1'
```

```
<staging_projects>
  <staging_project name="openSUSE:Factory:Staging:A" state="unacceptable">
    <staged_requests count="6">
      <request id="368" type="submit" creator="scp" state="review" package="amet"
superseded_by="" updated="2020-04-29T17:39:36Z"/>
      <request id="369" type="submit" creator="scp" state="declined" package="aut_0"
superseded_by="" updated="2020-04-29T17:41:45Z"/>
      <request id="371" type="submit" creator="scp" state="review" package="dolor"
superseded_by="" updated="2020-04-29T18:07:51Z"/>
    </staged_requests>
    <untracked_requests count="0"/>
    <obsolete_requests count="2">
      <request id="369" type="submit" creator="scp" state="declined" package="aut_0"
superseded_by="" updated="2020-04-29T17:41:45Z"/>
    </obsolete_requests>
    <missing_reviews count="4">
```

```

    <review request="369" state="new" package="aut_0" creator="" by_user="Requestor"/>
  </missing_reviews>
  <building_repositories count="0"/>
  <broken_packages count="0"/>
  <checks count="0"/>
  <missing_checks count="0"/>
</staging_project>
<staging_project name="openSUSE:Factory:Staging:B" state="empty">
  <staged_requests count="0"/>
  <untracked_requests count="0"/>
  <obsolete_requests count="0"/>
  <missing_reviews count="0"/>
  <building_repositories count="0"/>
  <broken_packages count="0"/>
  <checks count="0"/>
  <missing_checks count="0"/>
</staging_project>
</staging_projects>

```

9.1.2 Overview of a Single Staging Project

This endpoint provides an overview of a single staging project.

```
geeko > osc api '/staging/openSUSE:Factory/staging_projects/openSUSE:Factory:Staging:A'
```

Which will return the following XML:

```
<staging_project name="openSUSE:Factory:Staging:A" />
```

The returned XML can include more information by adding any combination of this three parameters: requests, status and history. This example combines status and history:

```
geeko > osc api '/staging/openSUSE:Factory/staging_projects//openSUSE:Factory:Staging:A?
status=1&history=1'
```

```

<staging_project name="openSUSE:Factory:Staging:A" state="unacceptable">
  <building_repositories count="0"/>
  <broken_packages count="0"/>
  <checks count="0"/>
  <missing_checks count="0"/>
  <history count="8">
    <entry event_type="Staged request" request="368" package="amet" author="Admin"/>
    <entry event_type="Staged request" request="369" package="aut_0" author="Admin"/>
    <entry event_type="Staged request" request="371" package="dolor" author="Admin"/>
    <entry event_type="Staged request" request="374" package="harum" author="Admin"/>
  </history>
</staging_project>

```

```
<entry event_type="Unstaged request" request="374" package="harum" author="Admin"/>
</history>
</staging_project>
```

9.1.3 Copy a Staging Project

This endpoint creates a copy of a staging project. It will queue a job which is going to copy the project configuration, repositories, groups and users.

```
geeko > osc api -X POST '/staging/openSUSE:Factory/staging_projects/
openSUSE:Factory:Staging:A/copy/openSUSE:Factory:Staging:A-copy'
```

9.2 Working with Requests

One of the main features of the staging workflow is assigning incoming requests to different staging projects.

9.2.1 Assign Requests into a Staging Project

Our main project openSUSE:Factory received requests with id 1 and 2. We would like to group these two requests together and move them into the staging project openSUSE:Factory:Staging:A. This can be done with the following command which will create a link to the package in openSUSE:Factory:Staging:A.

```
geeko > osc api -X POST '/staging/openSUSE:Factory/staging_projects/
openSUSE:Factory:Staging:A/staged_requests' -d '<requests><request id="1"/><request
id="2"/></requests>'
```

9.2.2 Remove Requests from a Staging Project

When we are done with testing the staging project openSUSE:Factory:Staging:A, we need to remove the requests 1 and 2 again. The following command will remove the package links from openSUSE:Factory:Staging:A.

```
geeko > osc api -X DELETE '/staging/openSUSE:Factory/staging_projects/
openSUSE:Factory:Staging:A/staged_requests' -d '<requests><request id="1"/><request
id="2"/></requests>'
```

9.2.3 List Requests of a Staging Project

Listing all requests which are currently assigned to openSUSE:Factory:Staging:A can be done with the following command.

```
geeko > osc api '/staging/openSUSE:Factory/staging_projects/openSUSE:Factory:Staging:A/staged_requests'
```

Which will return the following XML:

```
<staged_requests>
  <request id="368" type="submit" creator="scp" state="review" package="amet"
superseded_by="" updated="2020-04-29T17:39:36Z"/>
  <request id="369" type="submit" creator="scp" state="declined" package="aut_0"
superseded_by="" updated="2020-04-29T17:41:45Z"/>
  <request id="371" type="submit" creator="scp" state="review" package="dolor"
superseded_by="" updated="2020-04-29T18:07:51Z"/>
</staged_requests>
```

9.2.4 Exclude Requests for a Staging Workflow

Our main project openSUSE:Factory received requests with id 3 and 4. We would like to exclude these two requests for the staging workflow project openSUSE:Factory.

```
geeko > osc api -X POST '/staging/openSUSE:Factory/excluded_requests' -d
'<excluded_requests><request id="3" description="Reason description for request id
3."></request><request id="4" description="Reason description for request id 4."></
request></excluded_requests>'
```

9.2.5 Bring Back Excluded Requests from a Staging Workflow

The following command will stop excluding requests with id 3 and 4 for the staging workflow project openSUSE:Factory.

```
geeko > osc api -X DELETE '/staging/openSUSE:Factory/excluded_requests' -d
'<excluded_requests><request id ="3"/><request id="4"/></excluded_requests>'
```

9.2.6 Accept Staging Project

Once all the requests are ready and the staging project has an acceptable state, the requests can be merged. In other words, the staging project can be accepted.


```
geeko > osc api -X POST '/staging/openSUSE:Factory/staging_projects/  
openSUSE:Factory:Staging:A/accept'
```

10 Notifications



Note: The feature described in this chapter is not included yet in any official OBS release.

That means this feature is only available through our Unstable project which we deploy on [public OBS instance \(https://build.opensuse.org/\)](https://build.opensuse.org/), but is still not part of an OBS release.

The "Notifications" page is the place that keeps you up-to-date with your daily OBS work. There, you can see notifications of events that happen in OBS and are important for you.

You can configure them in detail to receive just what is of your interest. Not only that, with the web UI interface, you can easily filter your notifications and mark them as read, improving your OBS workflow experience.

10.1 Notifications Configuration

Click on the "Manage Your Notifications" link to define which notifications you want to receive. You can see this link in the "Actions" menu on your "Profile" page and on the "Notifications" page.

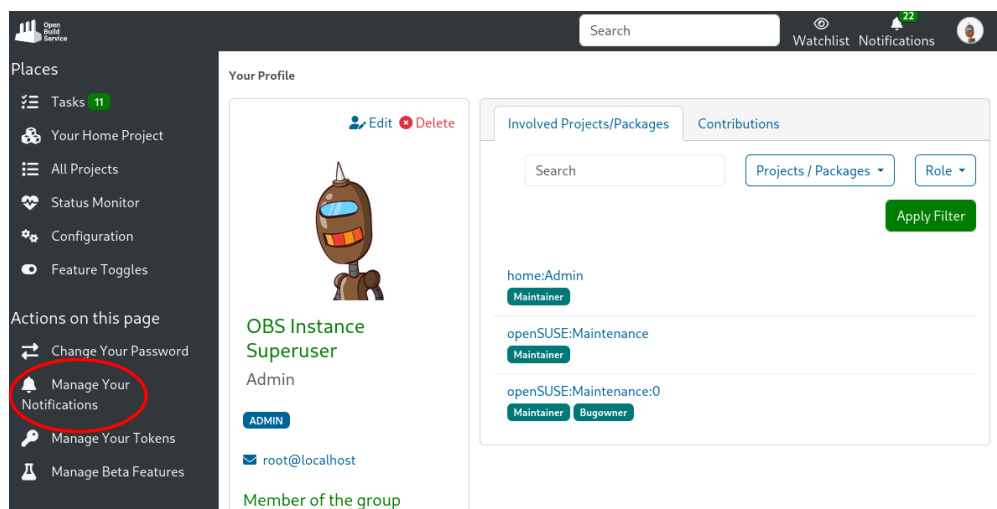


FIGURE 10.1: "MANAGE YOUR NOTIFICATIONS" LINK.

On this page, you can configure under which conditions you receive each type of notification (email, RSS, web). Mark the "web" checkboxes to configure what notifications you will receive in the web UI.

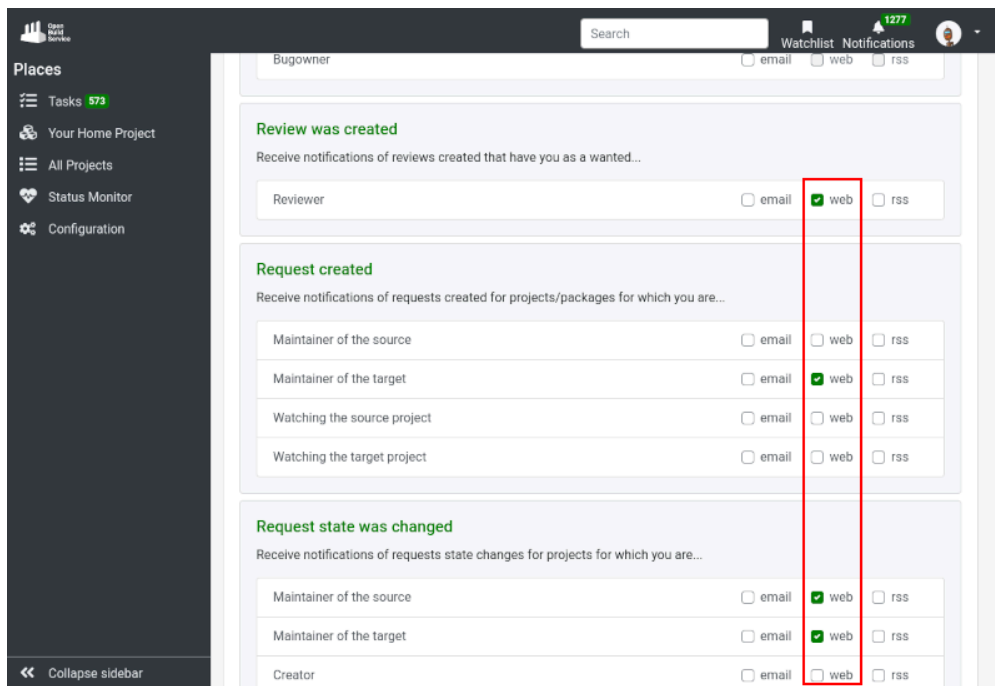


FIGURE 10.2: MARK WEB NOTIFICATIONS.

You can get web notifications from events happening to projects and packages where the group you belong to is an owner. Just mark the "web" checkbox in the groups you are interested in.

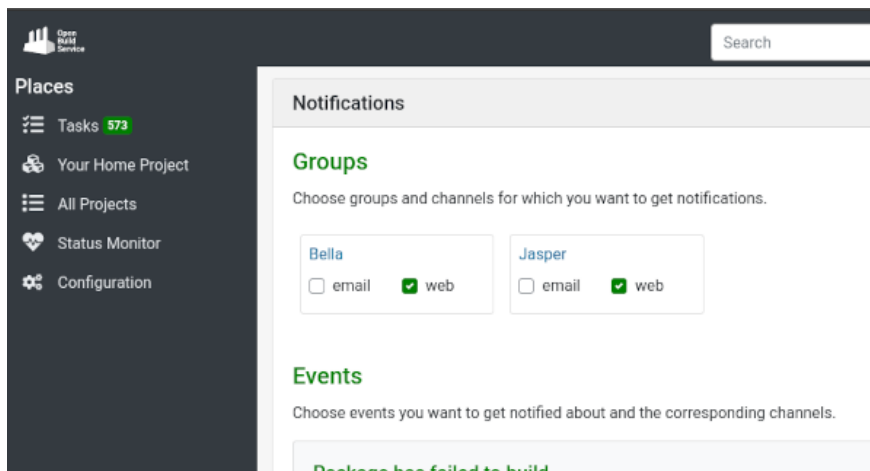


FIGURE 10.3: MARK WEB NOTIFICATIONS FOR GROUPS.

10.2 Where Can We Find the Notifications?

At the top right corner of every page, you will find a link to your notifications. It includes a counter of your unread notifications. Just click on it to access your notifications page.

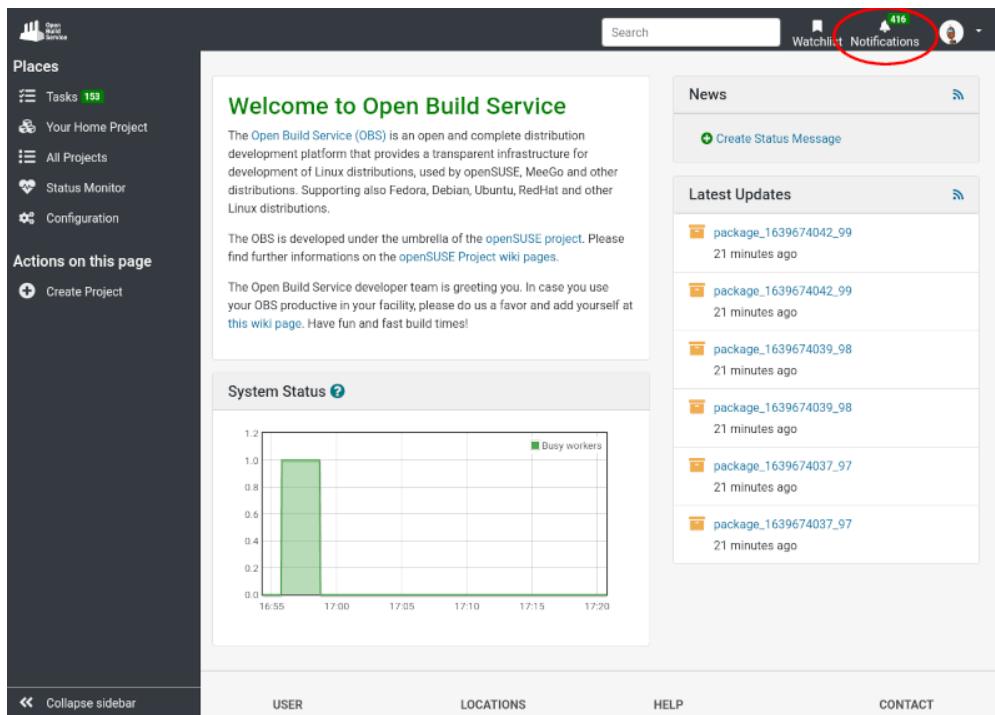


FIGURE 10.4: NOTIFICATIONS LINK.

10.3 Notifications Content

After clicking on the "Notifications" link, you will see a list of your most recent unread notifications.

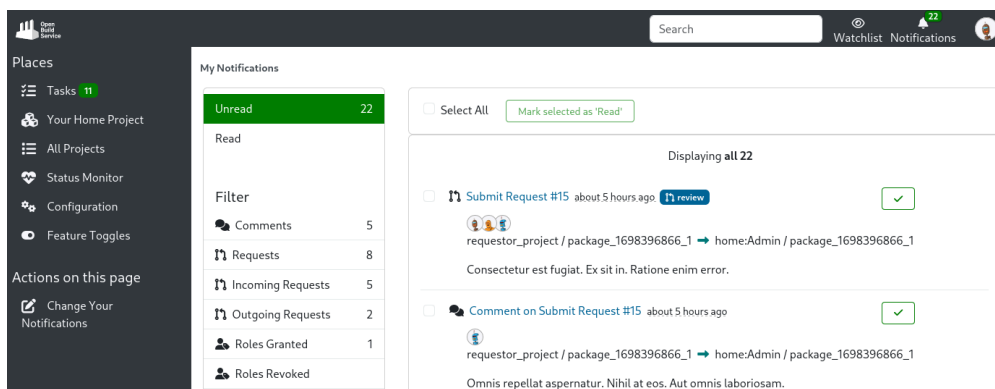


FIGURE 10.5: NOTIFICATIONS PAGE.

A notification can have different origins. Each notification item includes a couple of lines to help recognize them among the others. You always have the option to click on a notification, which will lead you to the specific section of OBS (request, comment, project, package, ...) with the full details on what caused the notification.

10.4 Mark Notification as Read or Unread

Once you read a notification and possibly take action, you can mark it as read. This will sort the notification in the "read" category and it's not shown anymore next to the one that might still need some attention. Also worth mentioning that we perform a periodical cleanup that deletes all the notifications older than 365 days.

There are different ways and places to mark your notifications as read. Marking a single notification can be done by simply clicking the button with the green checkmark shown on the notification itself.



FIGURE 10.6: MARK SINGLE NOTIFICATION AS READ.

Another way is to mark it directly on the related element (for example the associated request) through the little toolbar that appears on top. You will be redirected back to your notifications page afterwards.

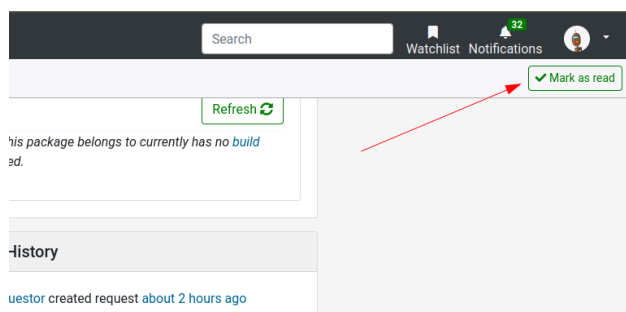


FIGURE 10.7: MARK NOTIFICATION AS READ THROUGH TOOLBAR.

If you received a lot of notifications and don't want to mark them one by one, there are two ways to speed up the process. You can use the checkboxes on the left to select individual notifications and click on the "Mark selected as 'Read'" button.

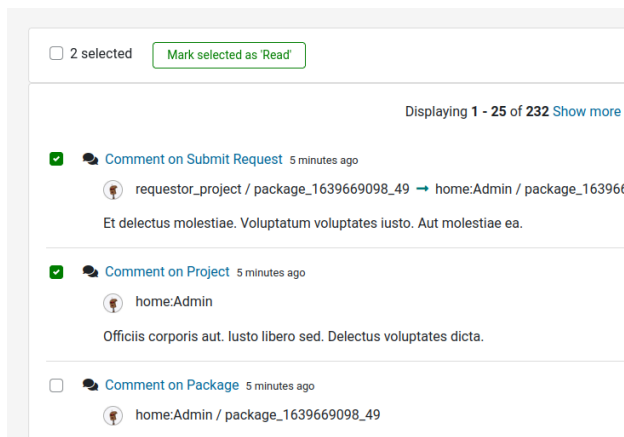


FIGURE 10.8: CHECKBOXES TO MARK MULTIPLE NOTIFICATIONS.

In case you exceed 300 notifications you have the possibility to use the button "Mark all as 'Read'" that will appear on the right side to mark all notifications of the chosen filter as read.

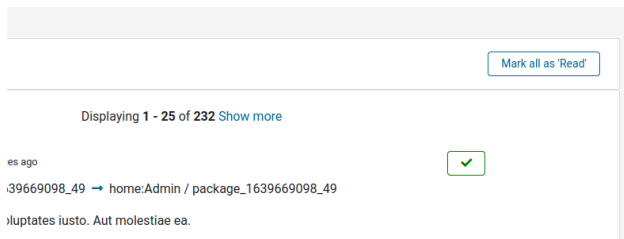


FIGURE 10.9: MARK ALL AS READ.

Should you ever mark a notification as read by accident, you can simply revert this by selecting the "read" filter on the right side and using the equivalent elements, used to mark them as read, in order to bring them back to your unread notifications.

10.5 Notifications Filters

Filters help you in focusing your attention on a subset of your notifications. They are on the left side of your screen.

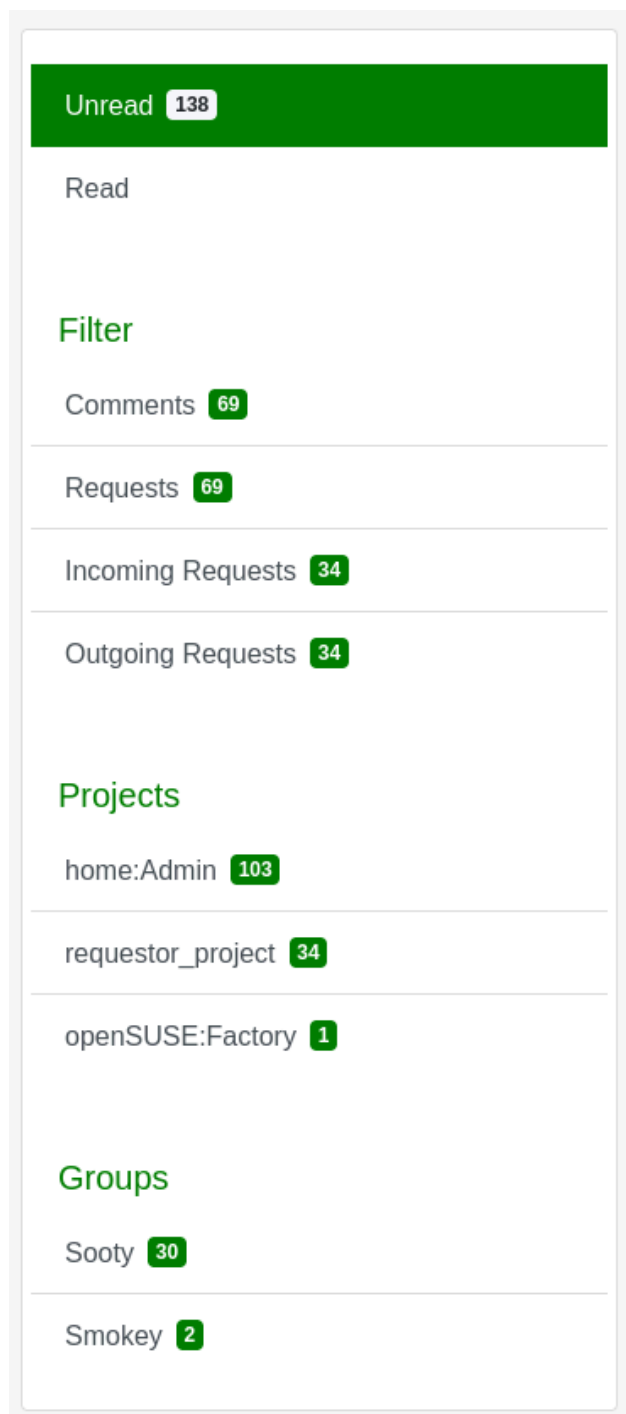


FIGURE 10.10: EXAMPLE OF NOTIFICATION FILTERS.

Here's a description of all available filters:

- *Unread*: All unread notifications, so notifications which you haven't marked as read, will be displayed when selecting this filter.
- *Read*: All notifications which you have marked as read will be displayed when selecting this filter.
- *Comments*: All unread notifications related to comments will be displayed when selecting this filter.
- *Requests*: All unread notifications related to requests will be displayed when selecting this filter.
- *Incoming Requests*: All unread notifications related to incoming requests, so requests which someone submitted to a project/package you are involved in, will be displayed when selecting this filter.
- *Outgoing Requests*: All unread notifications related to outgoing requests, so requests which you submitted to a project/package, will be displayed when selecting this filter.
- *Projects*: All unread notifications related to a project will be displayed when selecting a project filter. Each project with at least one unread notification has its own project filter.
- *Groups*: All unread notifications related to a group will be displayed when selecting a group filter. Each group with at least one unread notification has its own group filter.

10.6 API

Every user can check their unread notifications by querying:

```
osc api -X GET "/my/notifications"
```

It will return a list with your unread notifications:

```
<?xml version="1.0" encoding="UTF-8"?>
<notifications count="2">
  <total_pages>1</total_pages>
  <current_page>0</current_page>
  <notification id="3">
    <title>test 1</title>
    <who>Administrator</who>
```



```
<state>new</state>
<when/>
<event_type>review_wanted</event_type>
</notification>
<notification id="25">
  <title>test 2</title>
  <who>User 2</who>
  <event_type>comment_for_package</event_type>
  <when/>
</notification>
</notifications>
```

This one will toggle a notification read or unread.

```
osc api -X PUT "/my/notifications/3"
```

It will say if the operation went well or not.

```
<?xml version="1.0" encoding="UTF-8"?>
<status code="ok">
  <summary>Ok</summary>
</status>
```

For more information regarding the notifications API endpoints, check out our [New API Documentation \(https://api.opensuse.org/apidocs/#!/Notifications/get_my_notifications\)](https://api.opensuse.org/apidocs/#!/Notifications/get_my_notifications) ↗.

11 Moderation

As a platform where a lot of social interaction happens, OBS is not free of spam, harmful content or other outcomes of user misconduct. Therefore, OBS provides features which help any user report problematic content and help moderators to act accordingly. Those features are described below, and they are helpful for admins, staff members, moderators and users.

11.1 Code of Conduct

The Code of Conduct is the starting point of the moderation process. What is considered problematic content in your Build Service instance? Maybe spam, scams, or projects with a forbidden license? Both moderators and users should read the document and understand what is right or wrong to behave properly. The document is usually displayed on the footer of the Build Service instance.

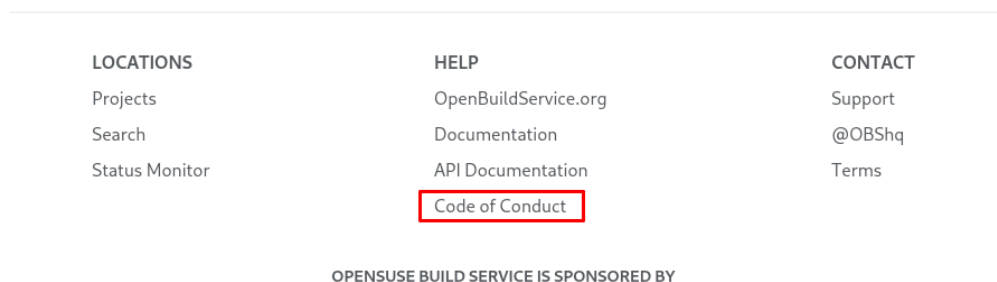


FIGURE 11.1: CODE OF CONDUCT

Admins are responsible for making the Code of Conduct visible to everyone. They should add the text on *Configuration > Code of Conduct* to make it appear on the footer. It allows markdown.

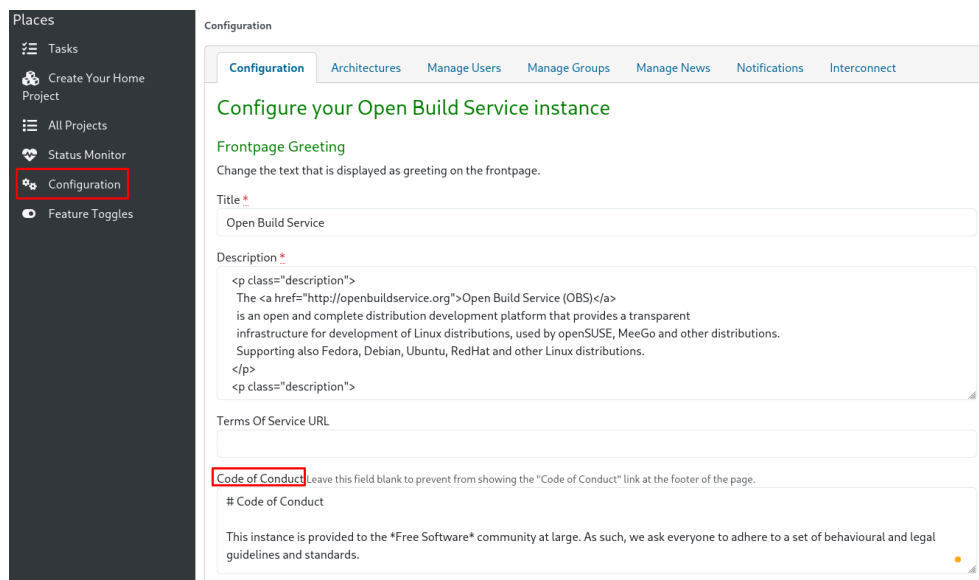


FIGURE 11.2: CONFIGURATION CODE OF CONDUCT

11.2 Reporting Problematic Content

11.2.1 Who Can Report?

Any kind of user can report problematic content in OBS.

11.2.2 What Can Be Reported?

This is the list of elements that can be reported:

- Comments
- Projects
- Packages
- Requests
- Users

11.2.3 How To Report?

You can find a *Report* action next to the comment, request or any of the elements mentioned above. It is displayed as a button, a link or an item in the actions menu. Click on it and fill in the form. You can simply choose one of the provided categories (Spam, Illegal Content, etc.) or write your reason.

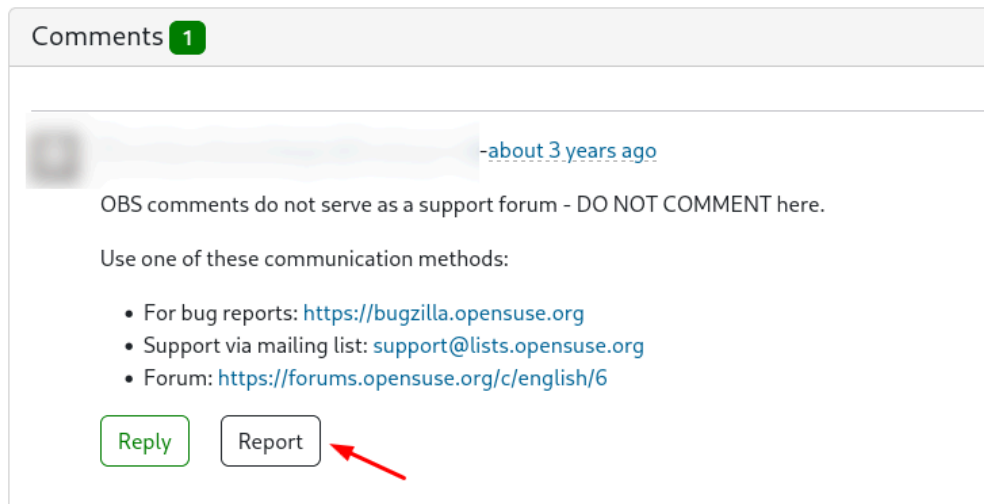


FIGURE 11.3: REPORT A COMMENT

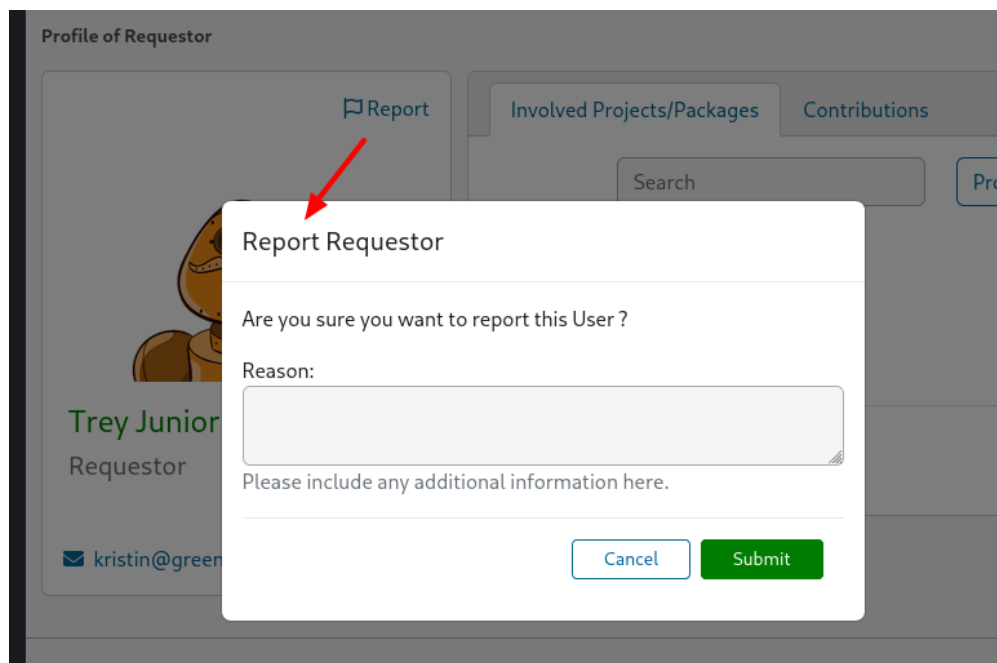


FIGURE 11.4: REPORT A USER

When reporting a comment, the form allows you to report the comment's author as well. Which saves you from having to go to the user's profile page afterwards.

Are you sure you want to report this Comment ?

You may want to read the [Code of Conduct](#) page first.

☒ Spam
☐ Scam
☐ Forbidden license
☐ Illegal content
☐ Other

Reason:

Please include any additional information here.

☐ Report the author of the comment

[Cancel](#) [Submit](#)

FIGURE 11.5: REPORT COMMENT AND AUTHOR

Once the element is reported, moderators will act accordingly. The following sections describe how they should proceed.

11.3 Acting as a Moderator

Moderators should inspect the elements reported by the users to decide whether the reports are fair or not, and act accordingly. They can dismiss the report, hide comments or remove the problematic elements, including users.

11.3.1 Who Is a Moderator?

Admins and staff members are the default moderators in any OBS instance unless they delegate the role to another user. Going through *Configuration > Manage Users*, admins can assign the *Moderator* role to someone else. From that moment on, only the users with the *Moderator* role will be involved in the moderation process.

11.3.2 How Do Moderators Know About the Reports?

Next to any problematic element, they can see a yellow text warning about the reports.

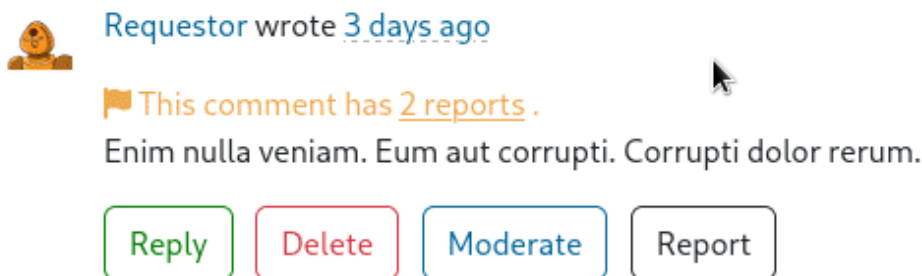


FIGURE 11.6: REPORTS WARNING

However, it is convenient that moderators subscribe to moderation-related events, so they can receive notifications of all the reports and other actions related to the moderation process.

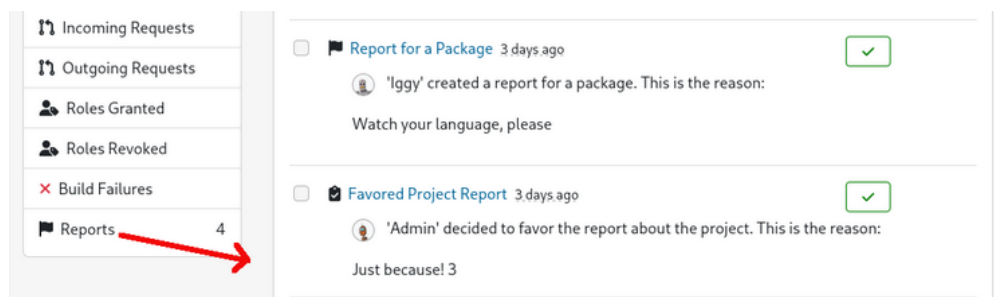


FIGURE 11.7: MODERATION-RELATED NOTIFICATIONS

11.3.3 How To Moderate?

As a moderator you can click on the yellow warning about reports, usually displayed close to the element. There, you can read all the reports and make a *Decision*. Write the reason why you agree (Favor) or disagree (Cleared).

Reports

userd reported 4 minutes ago
This might be spam

lggy reported less than a minute ago
Another spammer!

Decision

Reason

I agree, this is spam. I'm going to delete the user.

favor

Submit

FIGURE 11.8: DECISION FORM

You can read all the editions of a comment to better judge if the user misbehaved at some point. Use the arrow next to the comment date for that.

bar wrote about 3 hours ago (edited about 3 hours ago ▾)

Friendly comment!

Reply **Delete** **Moderate**

bar created about 3 hours ago
bar updated about 3 hours ago

Write **Preview**

FIGURE 11.9: COMMENT EDITION HISTORY

Once you have taken a favored decision, you or an admin can:

- hide the comment: click on the **Moderate** button next to it to replace the original text with a standard message;

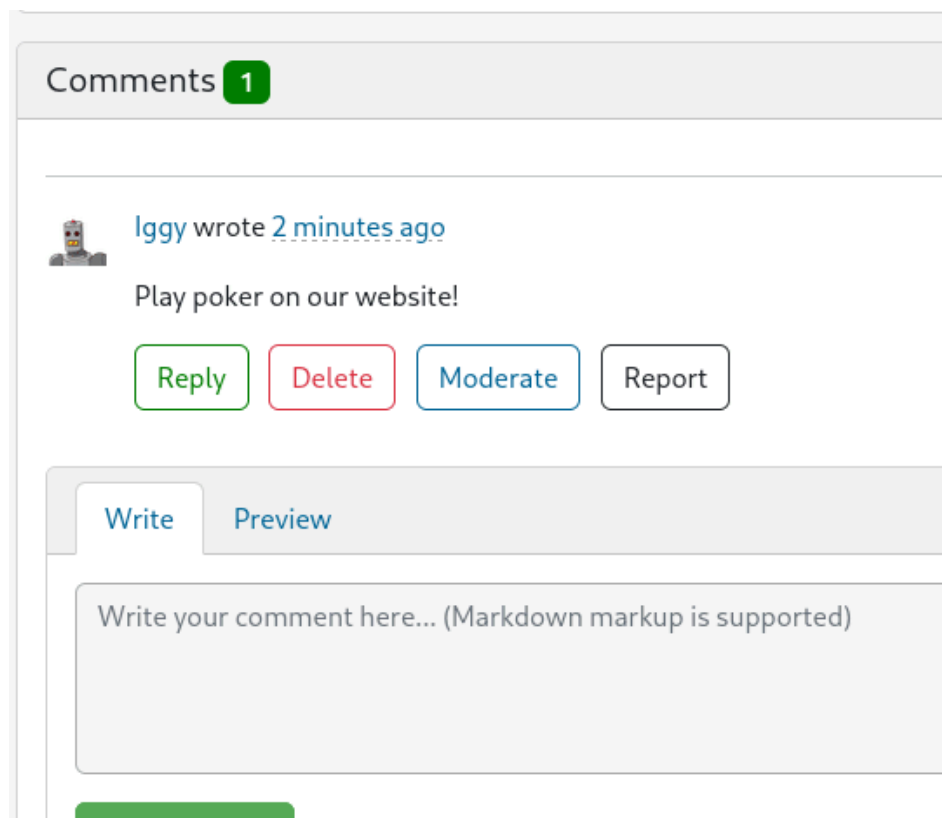


FIGURE 11.10: MODERATE A COMMENT

- revoke the request;
- remove the comment, project, package or user.

Most of these actions are reversible. Read section [Section 11.4, "Reverting Moderator's Actions"](#).

11.4 Reverting Moderator's Actions

Most of these actions are reversible except for removing a comment. You can recover a user, project and package. You can click on the Permit button to show the hidden comment again. However, the comments will be permanently removed.

11.5 User Appeal

After a moderator makes a decision based on a report, the affected user is notified. The user can appeal and justify why their content is not harmful. There is an *Appeal* action in the notification.

In case the moderator changes their mind, they can revert the actions they made. Read section [Section 11.4, “Reverting Moderator's Actions”](#).

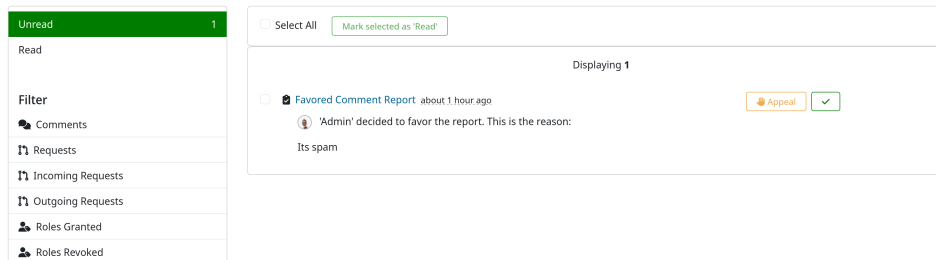


FIGURE 11.11: APPEAL TO DECISION

11.6 Canned Responses For Moderators

Handling reports can be redundant, therefore moderators can customize their own set of canned responses to reuse them for their decisions.

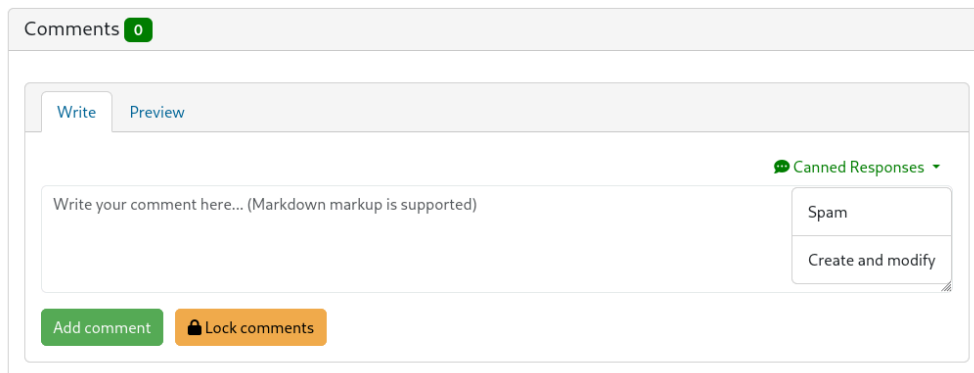


FIGURE 11.12: CANNED RESPONSES

V Best Practices

- 12 Using the OBS Web UI 129
- 13 Basic Concepts and Work Styles 176
- 14 How to integrate external SCM sources 177
- 15 Publishing Upstream Binaries 180
- 16 Bootstrapping 186
- 17 **osc** Example Commands 190
- 18 Advanced Project Setups 191
- 19 Building Kernel Modules 192
- 20 Common Questions and Solutions 193

12 Using the OBS Web UI

This chapter explains and shows how you could use OBS Web UI. We will show and use OBS Web UI based on <http://build.opensuse.org>. You need to make an account first to follow this chapter contents.

12.1 Homepage and Login

Open a browser and navigate to <https://build.opensuse.org>

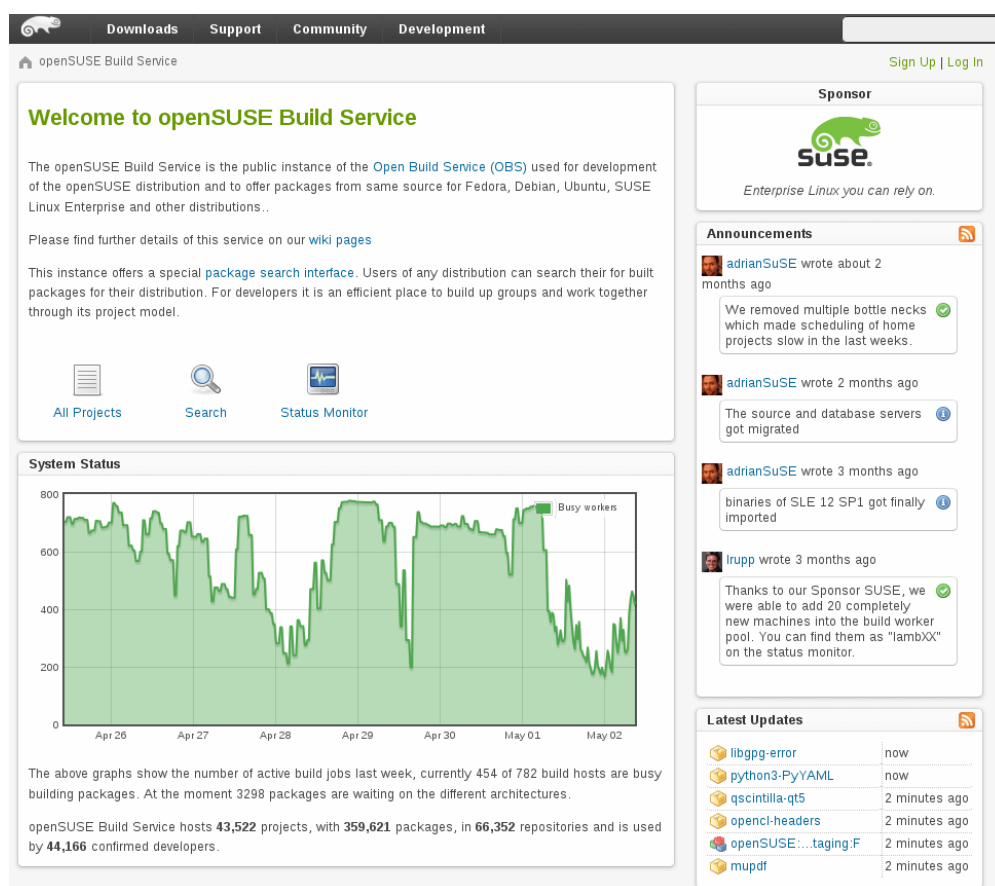


FIGURE 12.1: START PAGE

To proceed, you'll need to log in and authenticate with your username and your password. Click on Login and enter the data in the upper right corner.

The screenshot shows the openSUSE Build Service homepage. At the top is a navigation bar with links: Downloads, Support, Community, and Development. Below this is the 'openSUSE Build Service' header. The main content area is titled 'Welcome to openSUSE Build Service' and contains introductory text about the service. Below the text are three icons: 'All Projects', 'Search', and 'Status Monitor'. A 'System Status' section features a line graph showing 'Busy workers' over time from April 26 to May 02. To the right, there is a 'Log In' overlay with a username field (containing 'obswebui'), a password field (masked with dots), a 'Log In' button, and a 'Cancel' button. Below the login overlay is an 'Announcements' section with several entries from users like 'adrianSuSE' and 'lrupp'. At the bottom right is a 'Latest Updates' section listing recent package updates.

Welcome to openSUSE Build Service

The openSUSE Build Service is the public instance of the [Open Build Service \(OBS\)](#) used for development of the openSUSE distribution and to offer packages from same source for Fedora, Debian, Ubuntu, SUSE Linux Enterprise and other distributions..

Please find further details of this service on our [wiki pages](#)

This instance offers a special [package search interface](#). Users of any distribution can search their for built packages for their distribution. For developers it is an efficient place to build up groups and work together through its project model.

[All Projects](#) [Search](#) [Status Monitor](#)

System Status

The above graphs show the number of active build jobs last week, currently 454 of 782 build hosts are busy building packages. At the moment 3298 packages are waiting on the different architectures.

openSUSE Build Service hosts **43,522** projects, with **359,621** packages, in **66,352** repositories and is used by **44,166** confirmed developers.

Log In

obswebui

Log In

Cancel

Announcements

adrianSuSE wrote about 2 months ago

We removed multiple bottle necks which made scheduling of home projects slow in the last weeks.

adrianSuSE wrote 2 months ago

The source and database servers got migrated

adrianSuSE wrote 3 months ago

binaries of SLE 12 SP1 got finally imported

lrupp wrote 3 months ago

Thanks to our Sponsor SUSE, we were able to add 20 completely new machines into the build worker pool. You can find them as "lambXX" on the status monitor.

Latest Updates

libpgp-error	now
python3-PyYAML	now
qscintilla-qt5	2 minutes ago
opencl-headers	2 minutes ago
openSUSE...taging:F	2 minutes ago
mupdf	2 minutes ago

FIGURE 12.2: LOGIN

After successful authentication, you'll end up on the start page again - with new options visible. We'll go through most of them in detail, but first lets create your home: in the next step.

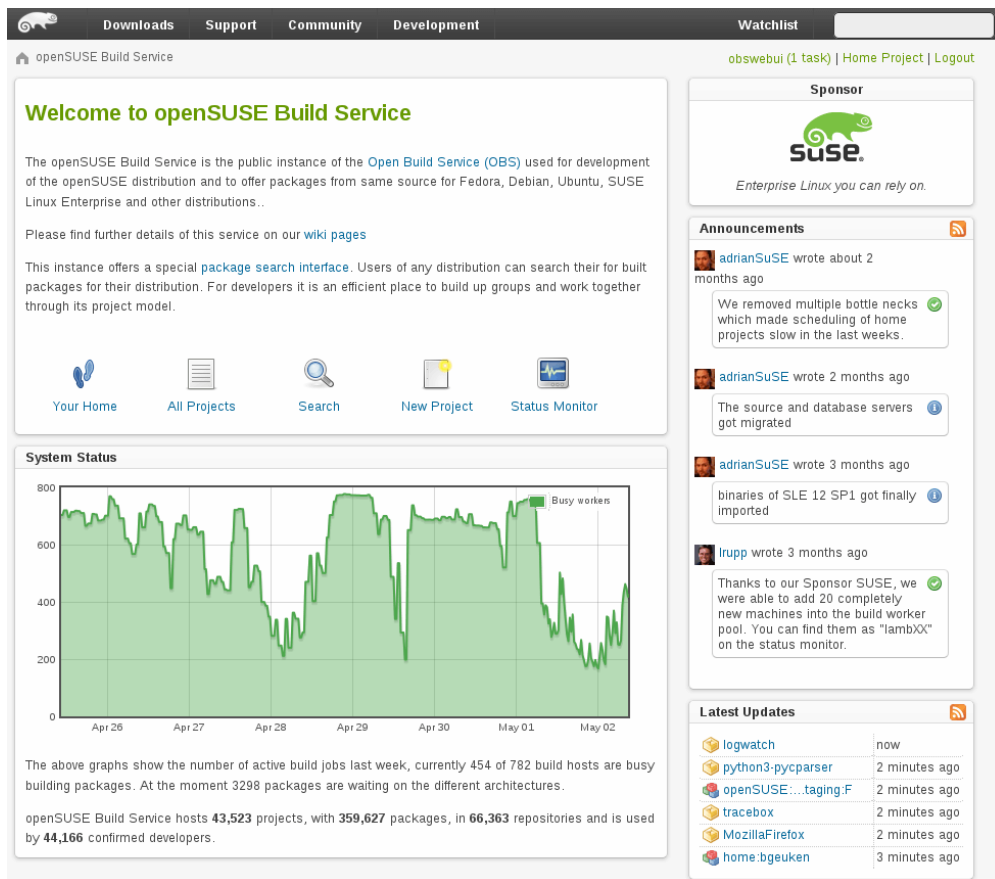


FIGURE 12.3: LOGGED IN

12.2 Home Project

Every user has a home project (home:[userid]) where they have write access by default. This is a personal workspace where you can experiment and play. Click on the link "Home Project" at the upper right to get to your home project.

12.2.1 The Project Page

Your home project will be empty for now, but you can add packages containing sources/build recipes and projects which are containers for the build targets. As you can see, you're the default maintainer which grants you full write access to this project. You're also the bug owner of your project.

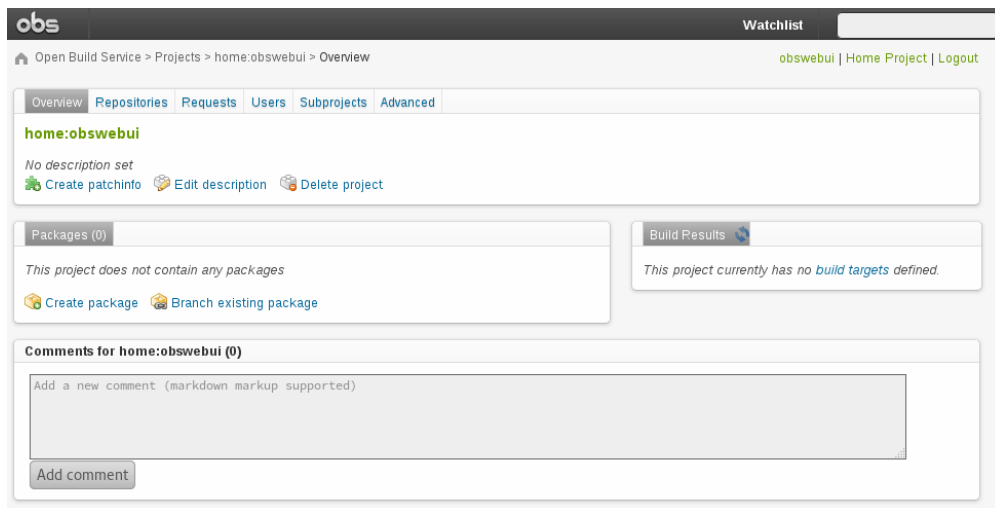


FIGURE 12.4: PROJECT PAGE

12.2.2 Changing a project's title and description

On every project page you will find a "Edit description" link. This link will lead you to a place where you can review and change your project's title and description. Click on the "Update project" button to save.

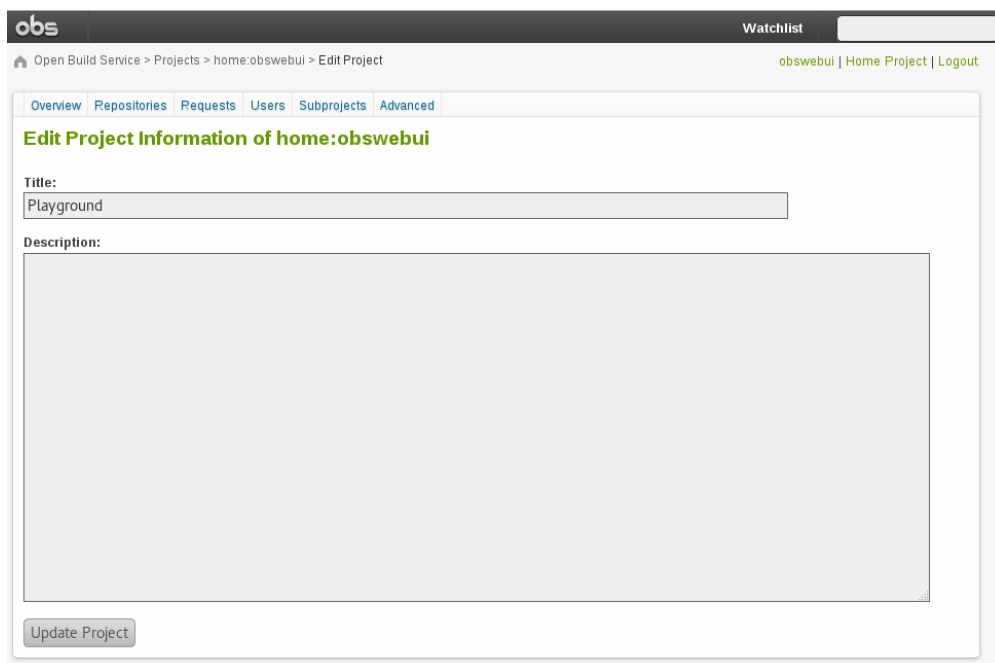


FIGURE 12.5: UPDATING PROJECT DESCRIPTION

12.2.3 Creating Subprojects to a Project

Subprojects are projects that are part of another projects namespace. Subprojects are an easy way to organize multiple projects. On the "Subprojects" tab you can find a list subprojects that belong to a project. To create a new subproject click on the "New subproject" link, fill in the form and press the "Create project" button.



Note

Maintainers of upper projects can always modify the subprojects. Apart from that all projects are separated and have no influence on each other.

FIGURE 12.6: CREATING SUBPROJECTS

12.3 My Projects, Server Status

For now, let's leave your home for a bit and explore the build service. Click on "My Projects" on the left at the bottom. This opens a page listing your watched projects and your involvements in projects or packages.

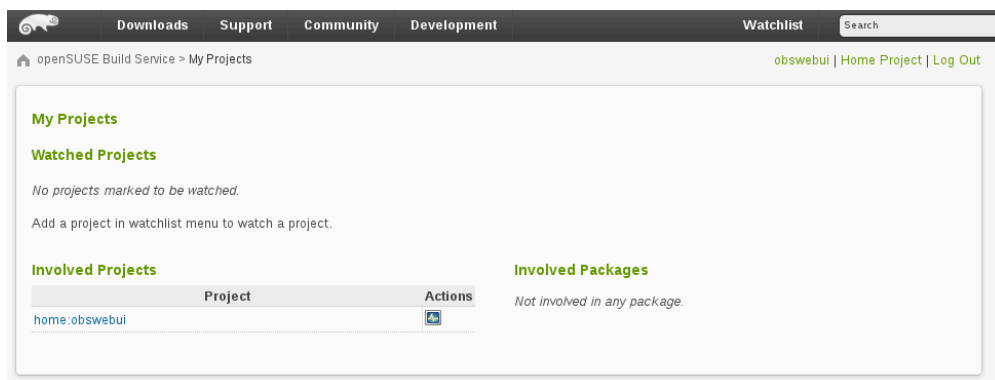


FIGURE 12.7: MY PROJECTS

Now, let's visit the main monitor page by clicking on "Status Monitor". You see here the status of the services, some graphs and graphics are showing the currently running and completed jobs and the overall load.

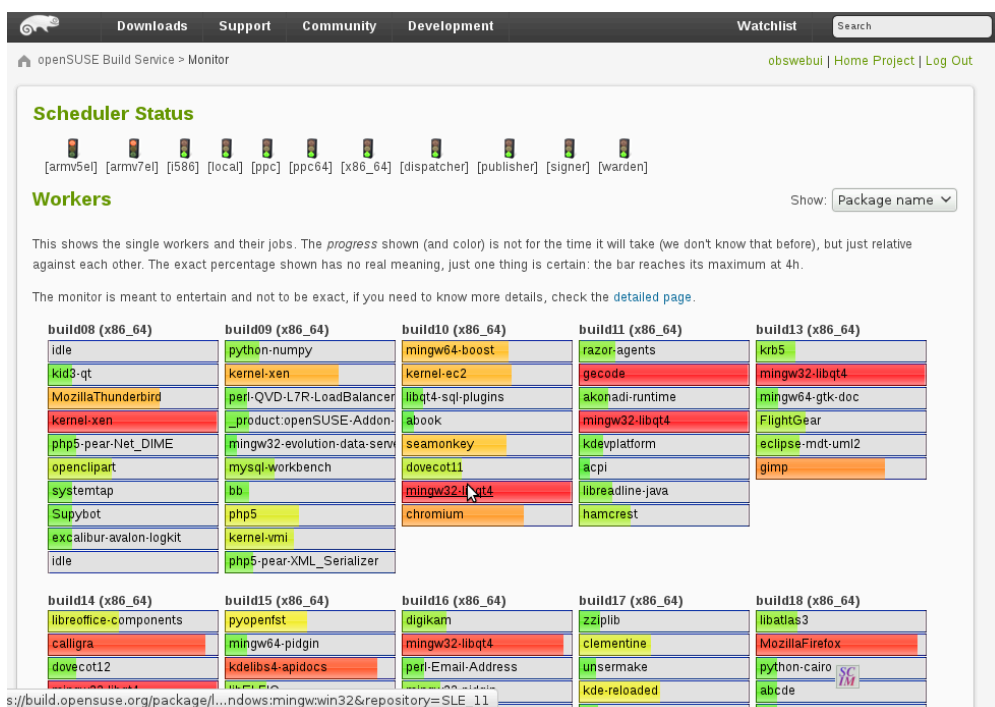


FIGURE 12.8: STATUS MONITOR

12.4 Create a link to a package in your home

We'll show you how you can log in and use the web interface hosted at build.opensuse.org. This includes login, adding a link to a package in your personal workspace (home:~) and how to build that package by adding a repository. First, let's enter "My Projects" by clicking on the link at the bottom left.

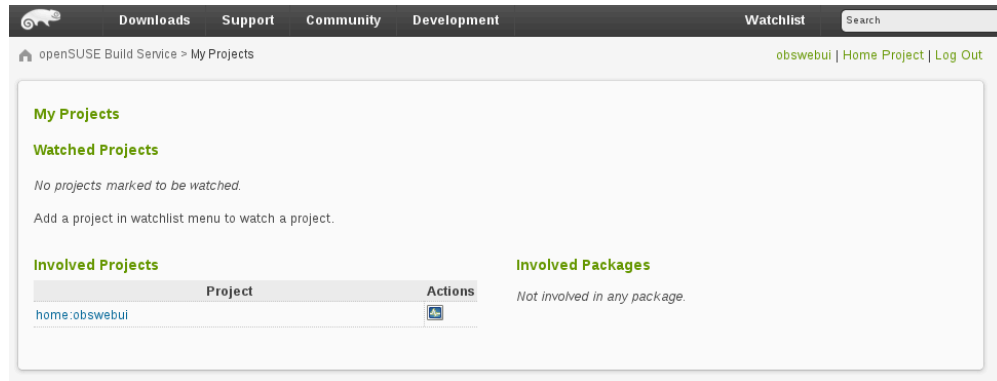


FIGURE 12.9: MY PROJECTS

Now let's create a link to a package and add a repository to build against. A link is basically a pointer to sources of an already existing package. By "repository" we mean container of built binary packages like Debian_8 or openSUSE_13.2. Let's follow these steps:

1. Add link to the existing package.
2. Add repository.
3. Observe the build on the monitor page.
4. Look at package's page.

12.4.1 Add Link to Existing Package

Right below packages, there's "Branch Package from other Project" .

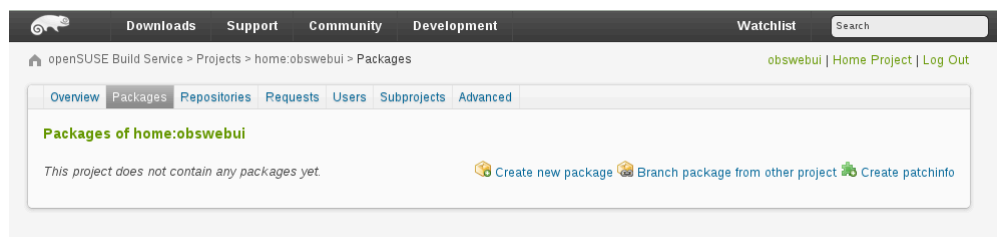


FIGURE 12.10: BRANCH PACKAGE

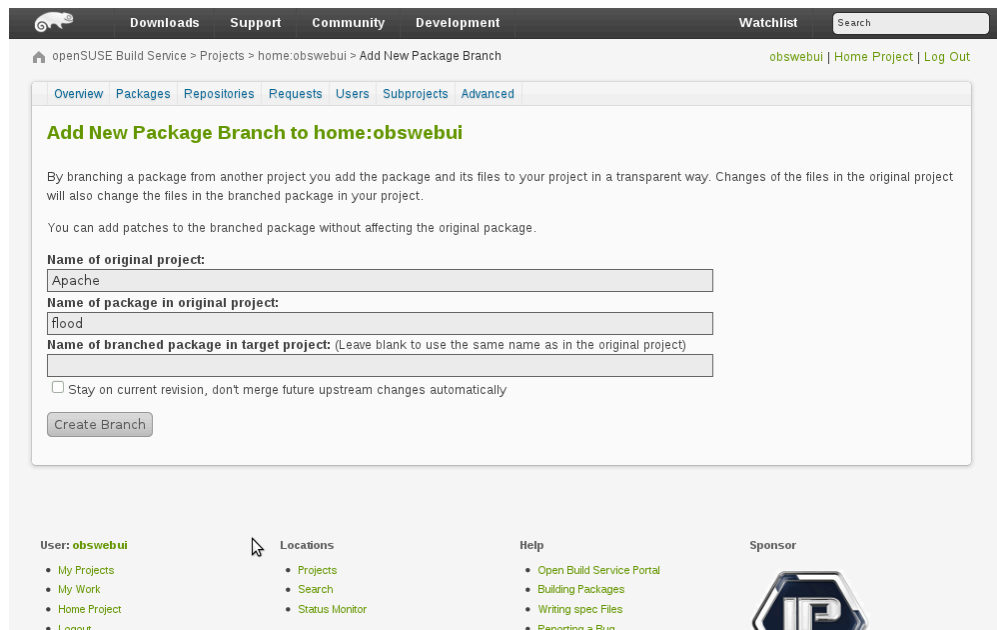
Open that page and enter for

Name of original project:
Apache

and for

Name of package in original project:
flood

- we'll leave "Name of linked package in target project" empty. This is shown on the next picture:



The screenshot shows the 'Add New Package Branch' page in the openSUSE Build Service. The page has a dark header with navigation links: Downloads, Support, Community, Development, Watchlist, and a search bar. Below the header, the breadcrumb trail is 'openSUSE Build Service > Projects > home:obswebui > Add New Package Branch'. The main content area has tabs for Overview, Packages, Repositories, Requests, Users, Subprojects, and Advanced. The 'Overview' tab is active, showing the title 'Add New Package Branch to home:obswebui'. Below the title, there is explanatory text about branching and a note about adding patches. The form contains three input fields: 'Name of original project:' with 'Apache' entered, 'Name of package in original project:' with 'flood' entered, and 'Name of branched package in target project:' which is empty. A checkbox labeled 'Stay on current revision, don't merge future upstream changes automatically' is unchecked. A 'Create Branch' button is at the bottom of the form. The footer contains user information for 'obswebui', navigation links for Locations (Projects, Search, Status Monitor), Help (Open Build Service Portal, Building Packages, Writing spec Files, Reporting a Bug), and a Sponsor logo.

FIGURE 12.11: APACHE FLOOD BRANCH

Proceed with "Create Branch" and you'll be redirected to your home again. You'll see a new package "flood" and a notice about the branch being added.

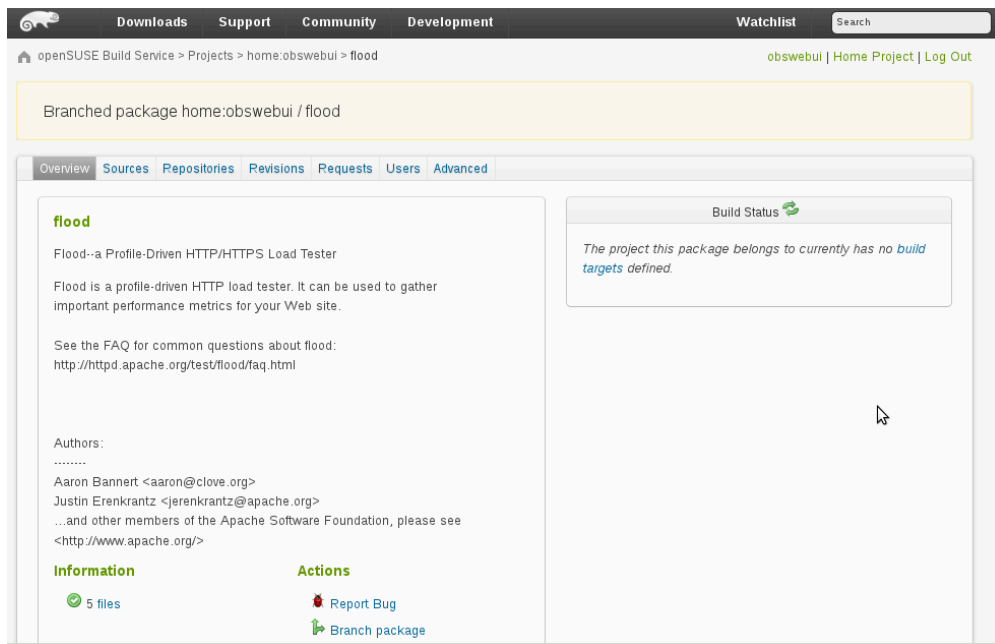


FIGURE 12.12: BRANCHED PACKAGE

Wonderful, we've added a pointer to the sources! Now we need to add a repository, so the builder knows the target-distribution to build packages for. How to add a repository to a project is documented at [Section 12.6.1, "Adding a repository"](#).

12.4.2 Package Page, Build Log and Project Monitor Page

Next, it is time to explore the Monitor page, the package detail page and the build log. Just click on the links and explore the web interface. I recommend starting with your home project's top level 'overview' page - click on the Overview tab and you may see something like this:

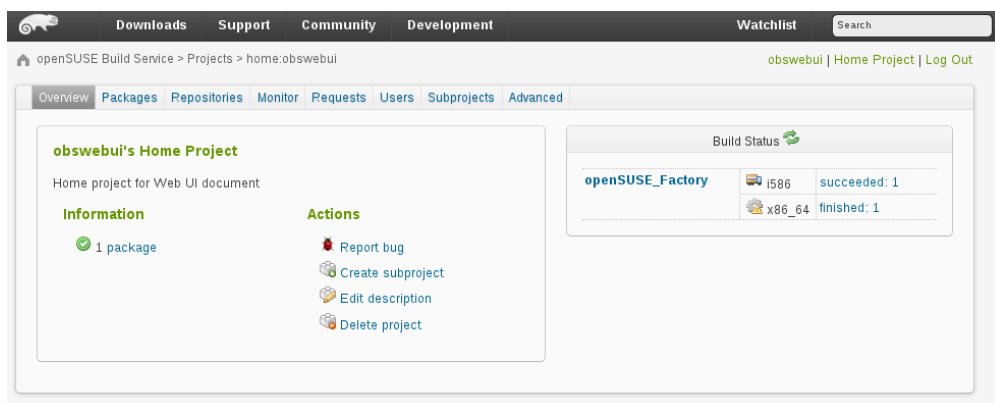
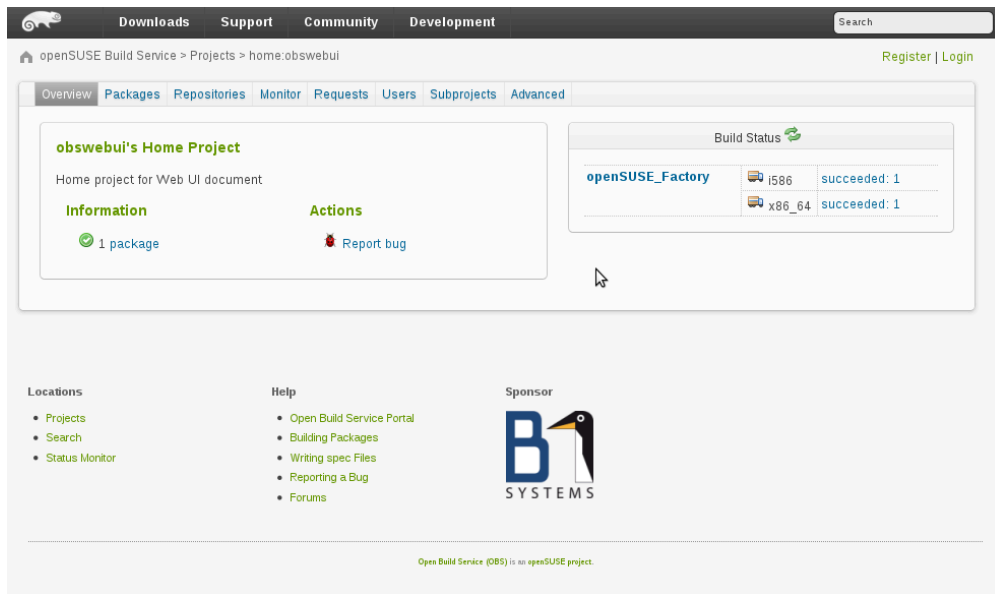


FIGURE 12.13: FLOOD_SUCCEEDED_FINISHED

If you wait a bit, you would see the below building success screen



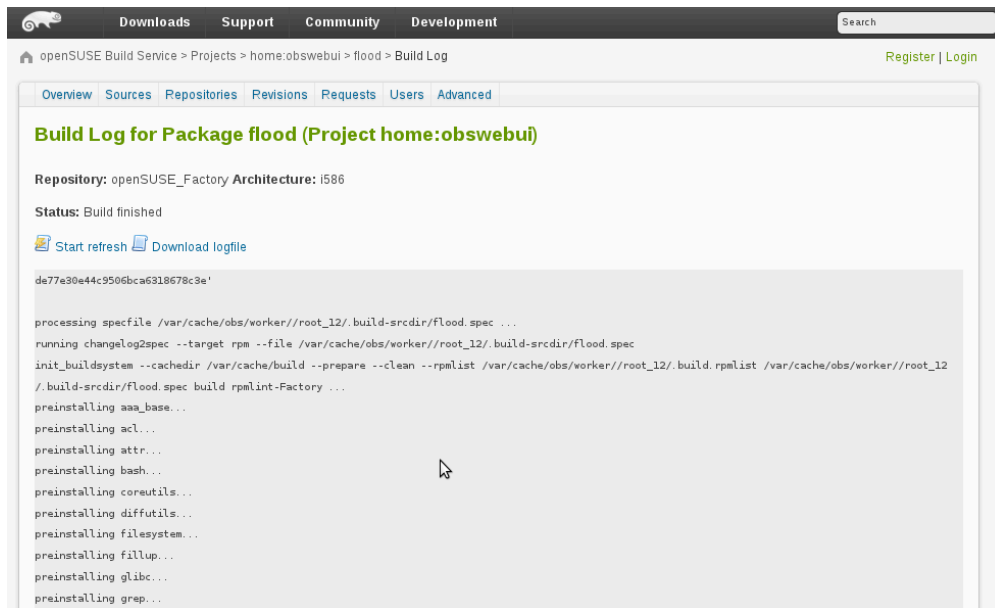
The screenshot shows the OBS project page for 'obswebui's Home Project'. The page has a navigation bar with links: Downloads, Support, Community, Development, and a search bar. The main content area is divided into sections: Overview, Packages, Repositories, Monitor, Requests, Users, Subprojects, and Advanced. The 'Overview' section shows the project name, a description, and a list of packages (1 package). The 'Build Status' section shows the build status for 'openSUSE_Factory' with a table of results:

Architecture	Status
i586	succeeded: 1
x86_64	succeeded: 1

The bottom of the page features a 'Locations' section with links to Projects, Search, and Status Monitor. A 'Help' section lists links to Open Build Service Portal, Building Packages, Writing spec Files, Reporting a Bug, and Forums. A 'Sponsor' section displays the logo for 'B SYSTEMS'.

FIGURE 12.14: FLOOD_BUILD_SUCCESS

Click the “succeeded” message, then you will see the build log as below.



The screenshot shows the OBS build log for the 'flood' package. The page has a navigation bar with links: Downloads, Support, Community, Development, and a search bar. The main content area is divided into sections: Overview, Sources, Repositories, Revisions, Requests, Users, and Advanced. The 'Overview' section shows the build log for the 'flood' package. The build log content is as follows:

```
de77e30e44c9506bca6318678c3e'

processing specfile /var/cache/obs/worker//root_12/.build-srcdir/flood.spec ...
running changelog2spec --target rpm --file /var/cache/obs/worker//root_12/.build-srcdir/flood.spec
init_buildsystem --cachedir /var/cache/build --prepare --clean --rpmList /var/cache/obs/worker//root_12/.build-rpmlist /var/cache/obs/worker//root_12/.build-srcdir/flood.spec build rpmlint-Factory ...
preinstalling aaa_base...
preinstalling acl...
preinstalling attr...
preinstalling bash...
preinstalling coreutils...
preinstalling diffutils...
preinstalling filesystem...
preinstalling fillup...
preinstalling glibc...
preinstalling grep...
```

FIGURE 12.15: FLOOD_BUILD_LOG

12.5 Repository Output: Built Packages

To find the RPMs you built, go to your home project page and click Repositories. From there click on the blue repository name. For example, openSUSE_Factory:



FIGURE 12.16: MY_REPOSITORY



Note

Published repositories are marked with the OBS truck

Now click *Go to download repository*. Note that publishing the repository might take a while. Before the binary repository is published, you will receive a 404 error. When the binaries are available, you will see something like this:

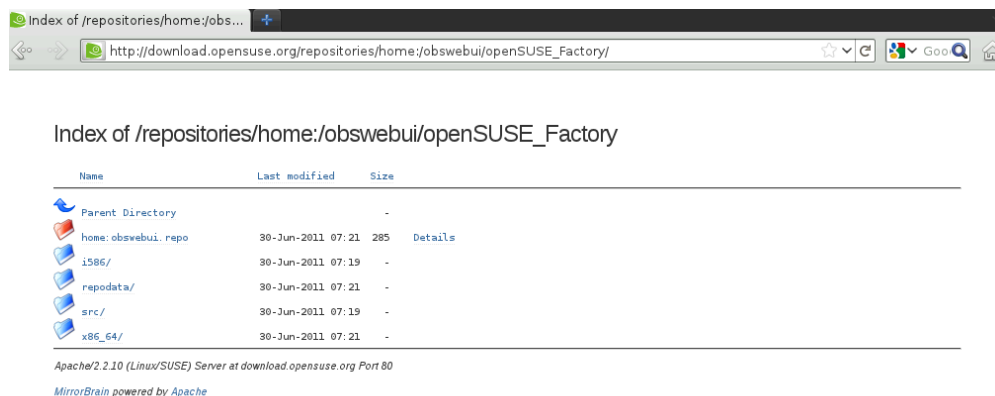


FIGURE 12.17: REPOSITORY STRUCTURE

Your RPMs can be found in the subdirectories, and the .repo file is suitable for use with zypper, yum or other repository-friendly package management tools.

12.6 Managing Repositories

This section will show how you can manage your project's repositories.

12.6.1 Adding a repository

To add a repository to your project, click on "Add Repositories" on the project's repository tab. This will direct you to a list of possible distributions you can build packages for, see *Figure 12.18*, "Adding a Repository to a Project".

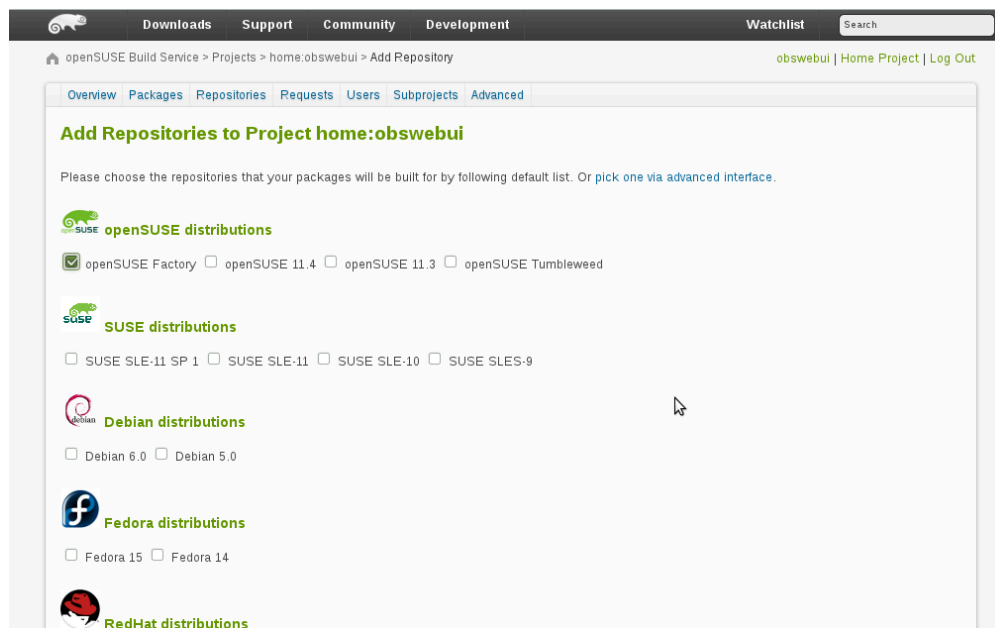


FIGURE 12.18: ADDING A REPOSITORY TO A PROJECT



Note

If you could not find a repository that fits your needs, you might want to switch to the expert mode. Click on the "Expert mode" link right to the button. This page allows you to search and select a repository of any project available in OBS and add it to your projects repository list.

This will take you back to your home: project. The build repository might be disabled: if so, click on the cogwheel to enable it. Congratulations, it is configured. On a heavily loaded server, it can sometimes take a few minutes for your changes to become effective, but your linked package will automatically begin building.

12.6.2 Add Download on Demand repositories to a project

When you have administrator rights you will be able to add Download on Demand repositories to your project. To do so, click on the "Add DoD repository" link and enter your DoD repository data into the form.

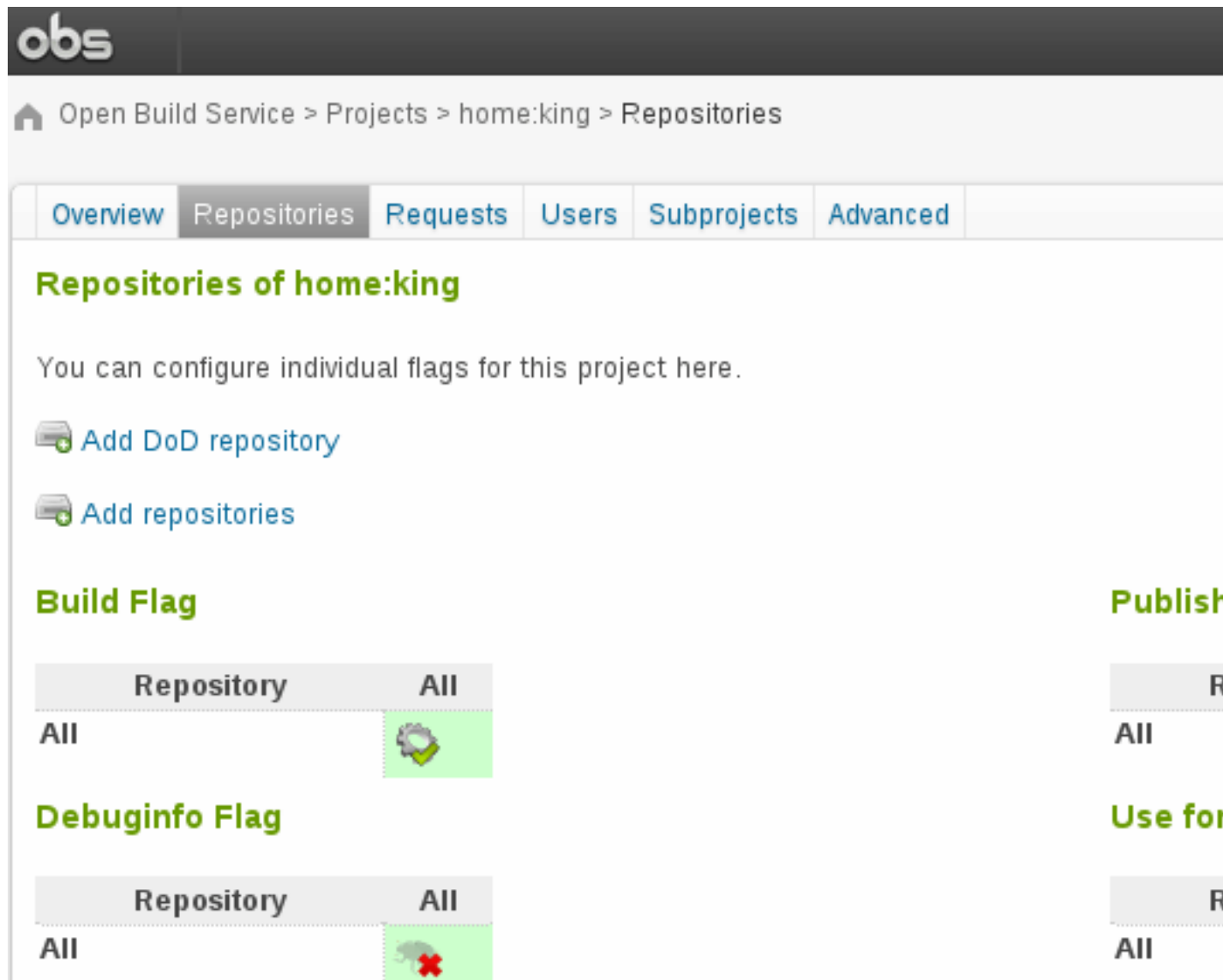


FIGURE 12.19: ADDING A DOWNLOAD ON DEMAND REPOSITORY

The minimal set of fields you have to enter are architecture, repository type and the URL that provides the binary packages. Detailed information about the data you can enter here can be found at [Section 22.3, "Download on Demand Repositories \(DoD\)"](#). Press "Save" to create the repository.

Repositories of home:king

You can configure individual flags for this project here.

Add DoD repository

Repository name

DoD test repository

Download on Demand Source

Architecture

x86_64 ▾

Type

rpmmd ▾

Url

http://opensuse.org/repo

Arch. Filter

Master Url

SSL Fingerprint

Public Key

Save

 [Add repositories](#)

FIGURE 12.20: DOWNLOAD ON DEMAND REPOSITORY FORM

When the repository got added you are able to edit, delete or add additional DoD repository sources.

12.6.3 Adding DoD Repository Sources to a Repository

The screenshot shows the Open Build Service (obs) web interface. The breadcrumb navigation is "Open Build Service > Projects > home:king > Repositories". The "Repositories" tab is selected. The page title is "Repositories of home:king". A message states: "You can configure individual flags for this project here." Below this, the "DoD test repository (x86_64)" is listed. There are links to "Delete repository" and "Go to download repository". Under "Download on demand sources", there is an "Add" link. The URL for x86_64 is "http://opensuse.org/repo (rpmmd)", with "Edit" and "Delete" links. Below these are links to "Add DoD repository" and "Add repositories".

Build Flag

Repository	All	x86_64
All		
DoD test...ository		

Debuginfo Flag

Repository	All	x86_64
All		
DoD test...ository		

FIGURE 12.21: ADDING DOWNLOAD ON DEMAND REPOSITORY SOURCES

Open the DoD repository sources form by clicking the "Add" link. Here you can enter your additional DoD repository source. Click the "Add Download on Demand" button.

Repositories of home:king

You can configure individual flags for this project here.

DoD test repository (i586, x86_64)

 Delete repository  Go to download repository

Download on demand sources

Add Download on Demand for DoD test repository

Architecture	<input type="text" value="i586"/>
Type	<input type="text" value="rpmmd"/>
Url	<input type="text" value="http://opensuse.org/repo/i586"/>
Arch. Filter	<input type="text"/>
Master Url	<input type="text"/>
SSL Fingerprint	<input type="text"/>
Public Key	<input type="text"/>

Add Download on Demand

x86_64: http://opensuse.org/repo/x86_64 (rpmmd)  Edit  Delete

FIGURE 12.22: FORM FOR ADDING DOD REPOSITORY SOURCES

12.6.4 Editing DoD Repository Sources

To edit DoD repository sources after they got added click on the "Edit" link that you find right to each DoD repository source.

Repositories of home:king

You can configure individual flags for this project here.

DoD test repository (*i586*, *x86_64*)

 Delete repository  Go to download repository

Download on demand sources  Add

x86_64: http://opensuse.org/repo/x86_64 (rpmmd)

Edit Download on Demand for DoD test repository / x86_64

Architecture	<input type="text" value="x86_64"/>
Type	<input type="text" value="rpmmd"/>
Url	<input type="text" value="http://opensuse.org/repo/x86_"/>
Arch. Filter	<input type="text"/>
Master Url	<input type="text" value="http://master.opensuse.org/fo"/>
SSL Fingerprint	<input type="text" value="sha256:0a64...0303"/>
Public Key	<input type="text"/>
<input type="button" value="Update Download on Demand"/>	

FIGURE 12.23: FORM FOR EDITING DOD REPOSITORY SOURCES

i586: <http://opensuse.org/repo/i586> (rpmmd)  Edit  Delete

12.6.5 Editing DoD Repository Sources

To delete a repository or repository source, click on the "Delete" link and accept the confirmation dialog.

12.7 Image Templates

Image templates are pre-configured image configurations. The [image templates page \(https://build.opensuse.org/image_templates\)](https://build.opensuse.org/image_templates) provides a list of these templates. Users can clone these templates and further configure them as they like.

How you can create your own image templates will be shown here.

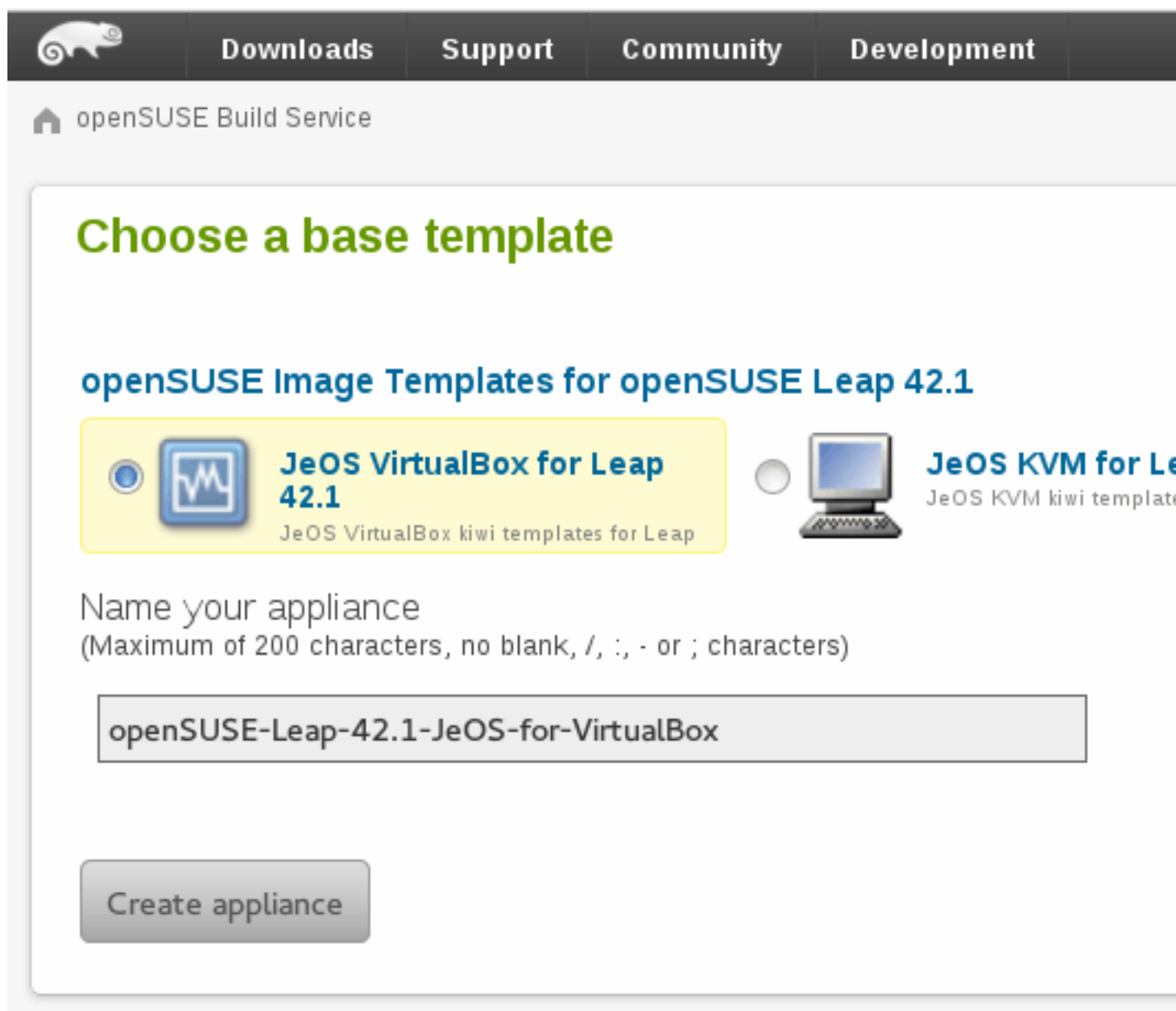


FIGURE 12.24: OBS TEMPLATES PAGE

12.7.1 Creating Own Image Templates

Create a subproject of your home project.

[Overview](#) [Repositories](#) [Monitor](#) [Requests](#) [Users](#) [Subprojects](#) [Advanced](#)

Subprojects of home:bgeuken

Subproject	
branches:devel:languages:ruby:extensions	Branch project
branches:home:justlest:prometheus	Branch project
branches:OBS:Server:Unstable	Branch project

Subproject Name:
home:bgeuken:

Title:

Description:

☐ Disable build results publishing.

[Create Project](#)

FIGURE 12.25: FORM FOR CREATING IMAGE TEMPLATE SUBPROJECT



Note

Published image templates are fetched via a project's attribute. Any package container living in a published project will be visible on the image templates page.

Within that project create a new package. That will be your actual image template.

The screenshot shows a web interface for creating a new package. At the top, there is a navigation bar with tabs: Overview, Repositories, Requests, Users, Subprojects, and Advanced. Below the navigation bar, the main heading is "Create New Package for home:bgeuken:my_image_temp". The form contains three input fields: "Name:" with the value "minimal_apache_server", "Title:" which is empty, and "Description:" which is a large empty text area. Below the description field, there is a checkbox labeled "Disable build results publishing." which is currently unchecked. At the bottom of the form, there is a "Save changes" button.

FIGURE 12.26: NEW IMAGE TEMPLATE

Add the 'KIWI image build' repository to your project. This repository is needed to build KIWI images in your project. Go to the 'Repositories' tab, click on 'Add repositories' and click on the 'KIWI image build' check box.

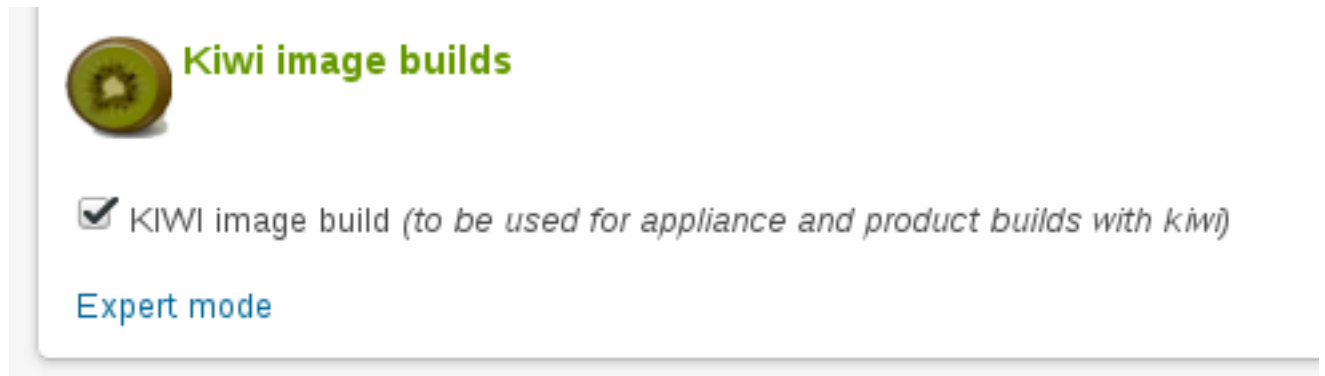


FIGURE 12.27: ENABLING THE KIWI IMAGE BUILD REPOSITORY

Add sources for your image configuration.

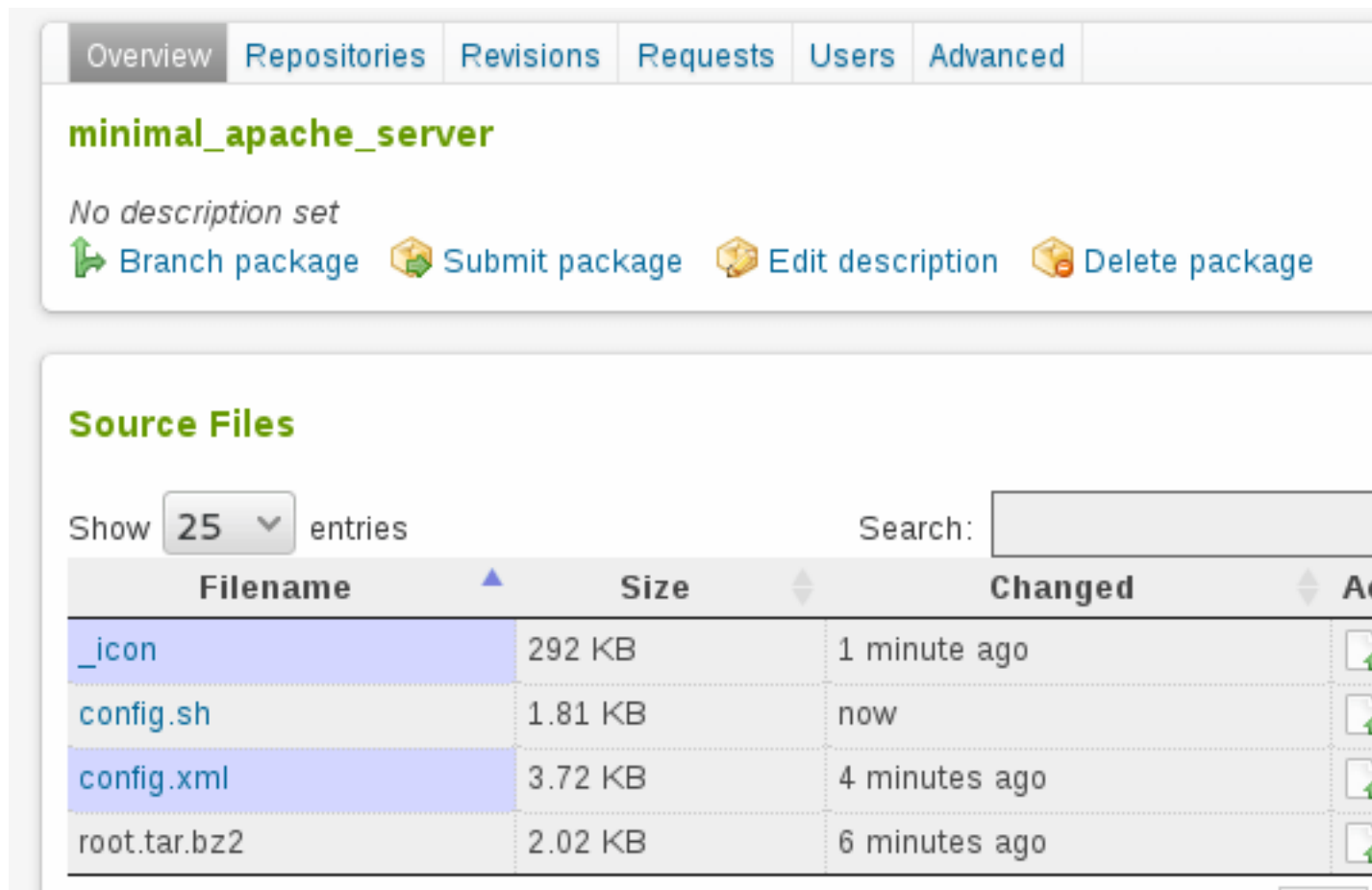


FIGURE 12.28: OVERVIEW OF SOURCES OF A CUSTOM IMAGE TEMPLATE

KIWI configurations usually consists of an xml configuration root tarball.

In addition, you can define an icon for your image templates by adding graphical image (for example, PNG, JPG) to your template sources and name it `_icon`. If that file exists, it will be used as icon for your image on the image templates page.

For a full list of image descriptions and general information about building images with KIWI, see the [KIWI project page \(https://github.com/OSInside/kiwi\)](https://github.com/OSInside/kiwi) and the [KIWI cookbook \(https://osinside.github.io/kiwi/index.html\)](https://osinside.github.io/kiwi/index.html).

12.7.2 Publishing Image Templates on the Official Image Templates Page

Once everything is set up and your templates are building, you might want to publish them. In that case contact the admin of the OBS instance you are using and ask them kindly to do so. If you happen to use the [official OBS \(https://build.opensuse.org/\)](https://build.opensuse.org/), that would be `admin@opensuse.org`.

12.8 KIWI Editor

You can edit the KIWI file associated to your project. It is only possible, at the moment, to edit the repository list and packages with type image. If you are running your own instance of OBS be sure you have the `kiwi_image_editor` feature enabled in your `config/feature.yml` file.






12.8.1 Accessing the KIWI Editor

Go to your package, and upload a file with the `.kiwi` extension (for example, `test.kiwi`), with valid KIWI content.

[Overview](#)
[Repositories](#)
[Revisions](#)
[Requests](#)
[Users](#)
[Advanced](#)


kiwi_package

No description set


 [Branch package](#)
 [Submit package](#)
 [Edit description](#)
 [Delete package](#)


Source Files


Show entries Search:



Filename	Size	Changed	Act
kiwi_file.kiwi	1.85 KB	about 3 hours ago	

Showing 1 to 1 of 1 entries Previous

 [Add file](#)

Latest Revision

 [David Kang \(david_kang\)](#) committed [about 3 hours ago](#) (revision 6)

 [Files changed](#)  [Browse Source](#)

Comments for home:david_kang (0)

Add a new comment (markdown markup supported)

FIGURE 12.29: EXAMPLE OF A PACKAGE WITH A KIWI XML FILE



Note

You should see now a "Edit KIWI" link (next to "Delete package" link).

Click on the "Edit KIWI" link and you will be redirected to the Editor.

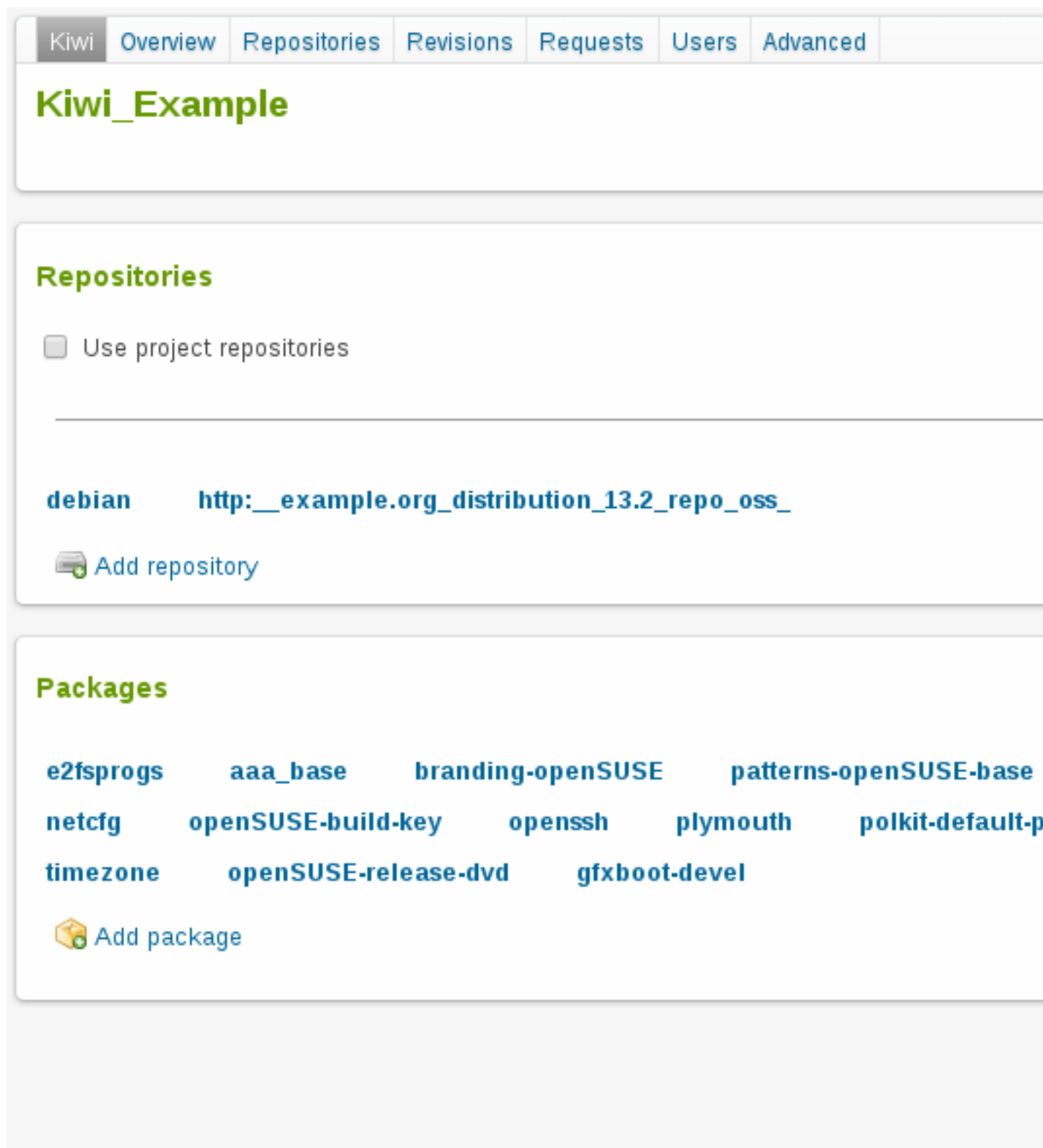


FIGURE 12.30: KIWI EDITOR. SHOW SCREEN

- *Repositories*: Displays the repositories set in the Kiwi file.
- *Packages*: Displays the packages of the package group with type *image*.

12.8.2 Adding Repositories in the KIWI Editor

To add a new repository click *Add repository* link and fill in the dialog. There are two ways to create it:

- *Basic Mode*: Adding the name of a project will provide a list with the repositories from that project.



FIGURE 12.31: KIWI ADDING A NEW REPOSITORY - BASIC MODE

- *Expert Mode*: This mode provides you with a set of customizable parameters for creating a repository.
 - *Type*: Valid options are *rpm-md* and *apt-deb*.
 - *Priority*: Repository priority for the given repository.
 - *Alias*: Alternative name for the configured repository.

- *Source Path*: Define the repository path.
- *User*: Specifies a user name for the given repository.
- *Password*: Specifies a password for the given repository.
- *Prefer License*: The repository providing this attribute will be used primarily to install the license tarball if found on that repository.
- *Image Include*: Specifies whether the given repository should be configured as a repository in the image.
- *Replaceable*: Defines a repository name which may be replaced by the repositories specified in the image description. This attribute should only be applied in the context of a boot image description.

FIGURE 12.32: KIWI ADDING A NEW REPOSITORY - EXPERT MODE

To use the configuration of the current project check the *Use project repositories* checkbox.

Repositories

☒ Use project repositories



This option will use the repositories from the current project. Other repositories

FIGURE 12.33: KIWI USE PROJECT CONFIGURATION



Note

This option will remove the other repositories from your kiwi file.

12.8.3 Adding Packages in the KIWI Editor

Adding a package is practically the same as adding a repository. We offer an autocomplete for the package name that will show you the package available in the repositories added previously.



Edit package

Name:
e2fsprogs

Arch:

Replaces:

☐ Bootinclude ☐ Bootdelete

Cancel Continue

FIGURE 12.34: KIWI ADDING A NEW PACKAGE



Note

The package groups shown in the editor are only those with type *image* and the packages will be added in this kind of package group. If it didn't exist previously, the KIWI Editor creates a package group with type *image* for you.

12.9 Manage Group

Only administrators and users with Maintainer rights can manage groups. They can add and remove other users from the group, as well as give them Maintainer rights.

On the Group Members tab, there is a link to Add Member, then enter the name of an existing user. You can click on the Maintainer checkbox to give Maintainer rights to a user.

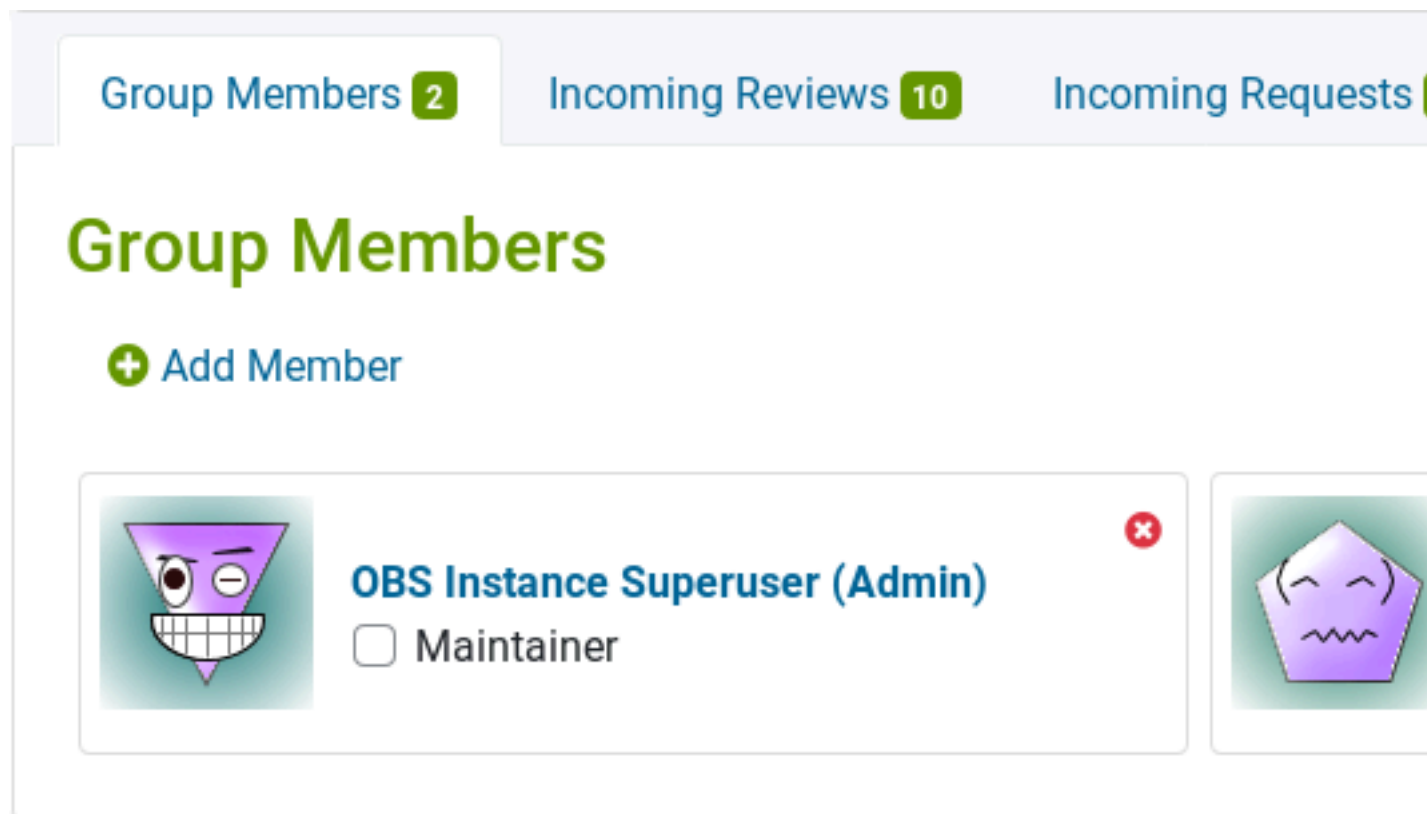


FIGURE 12.35: **MANAGE A GROUP**

12.10 Staging Workflow

The Build Service is well known for providing an easy way to build and distribute binary packages from source code. The packages, grouped together in what we call a project, are built every time they are updated. The maintainers of the package can choose among a wide range of operating systems and hardware architectures to build the packages on. Those continuous building processes ensure that the packages are always working for the different setups.

The maintenance of those packages can be made on a collaborative way via Build Service. As shown in the following diagram, the maintainers can create a package and then they or any other developer can branch it (make a copy of it), can do some changes on its code and can request those changes to be applied on the original package. After that, the maintainers usually review the request, chat with the developer in case it needs some fixes and end up accepting the request. Doing so, the new changes to the code become part of the package's source code.

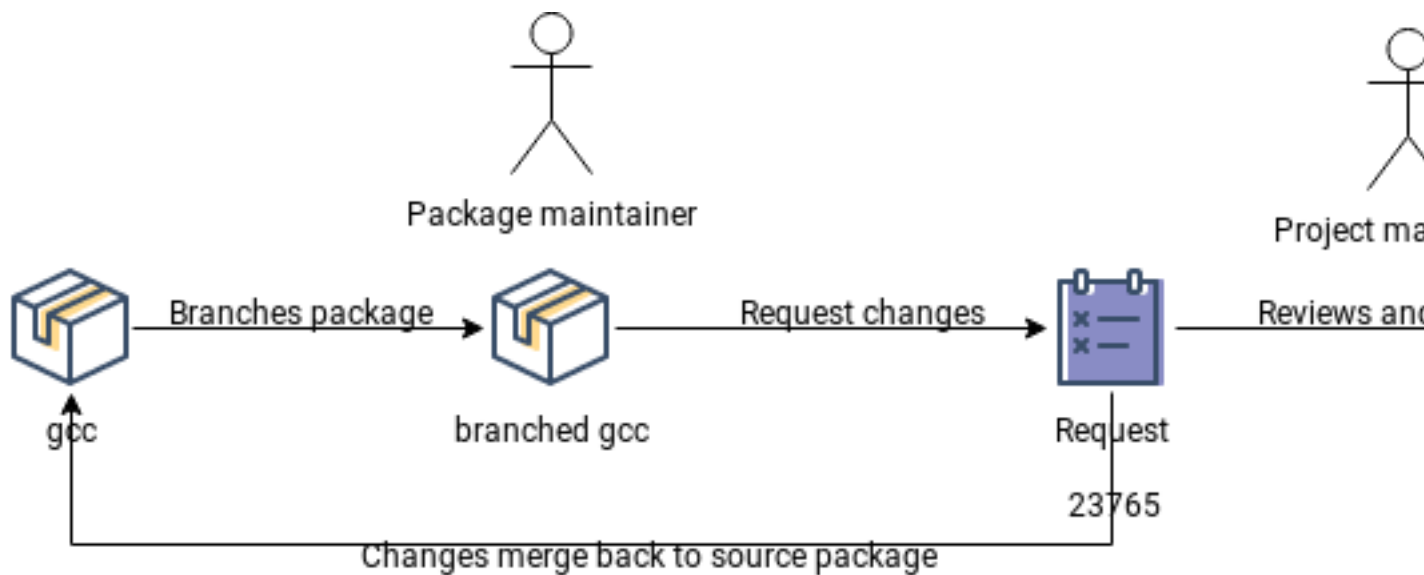


FIGURE 12.36: STAGING WORKFLOW BASIC SCHEMA

However, the workflow is not always that easy. Apart from managing individual packages, Build Service provides many other functionalities and it even allows us to release entire distributions. In a very simplistic way, we can say a distribution is just a Build Service project with thousands of packages inside. Packages that have been selected to be installed together as part of the distribution.

When dealing with such a big project, many people request changes in many different packages all the time. They have to be reviewed, adjusted and tested (built) before being accepted. As you can imagine, it becomes nonviable to review the packages one by one. Even if the maintainers check that a package is not broken and merge it, it can break everything else for conflicts with other packages. To deal with these situations, Build Service provides what we call Staging Workflow.

The idea behind the Staging Workflow is testing the requests incrementally by batches. First, a copy of the original project is created, it is called Staging Project and is going to act as a playground. The Staging Managers select some of the requests they consider to be belonging together and assign the corresponding packages to the Staging Project. This way, the groups of packages are going to be tested (built) in one go. Once the Staging Project gets built, the changes can be merged to the original project.

The Staging Managers can create as many Staging Projects as they require and can assign different selections of requests to each of them. It is still tedious solving the conflicts that appear between them, but being able to test a lot of packages in parallel is much more efficient than doing the same package by package.

Let's make it clearer with a real example. Imagine we are working on the project openSUSE Factory and we start working on its Staging Workflow.

Many contributors and maintainers really want some improvements to be applied on their packages, so the openSUSE:Factory project receives new requests all the time. Among all of them, there are a few that are related to Gnome packages, so the Staging Managers decide to stage them together in openSUSE:Factory:Staging:A. The Staging Project is an exact copy of the main project openSUSE:Factory.

The building process begins and, if something gets broken, the Staging Managers ask the requester to fix it. This can add even more requests to the scene but the goal is always getting a working version of openSUSE:Factory:Staging:A by fixing or even discarding some of the requests. When the building process finishes successfully, the requested changes are merged in the source code of openSUSE:Factory and some other batches of requests are staged again and again until we come up with an stable version of openSUSE:Factory ready to be released.

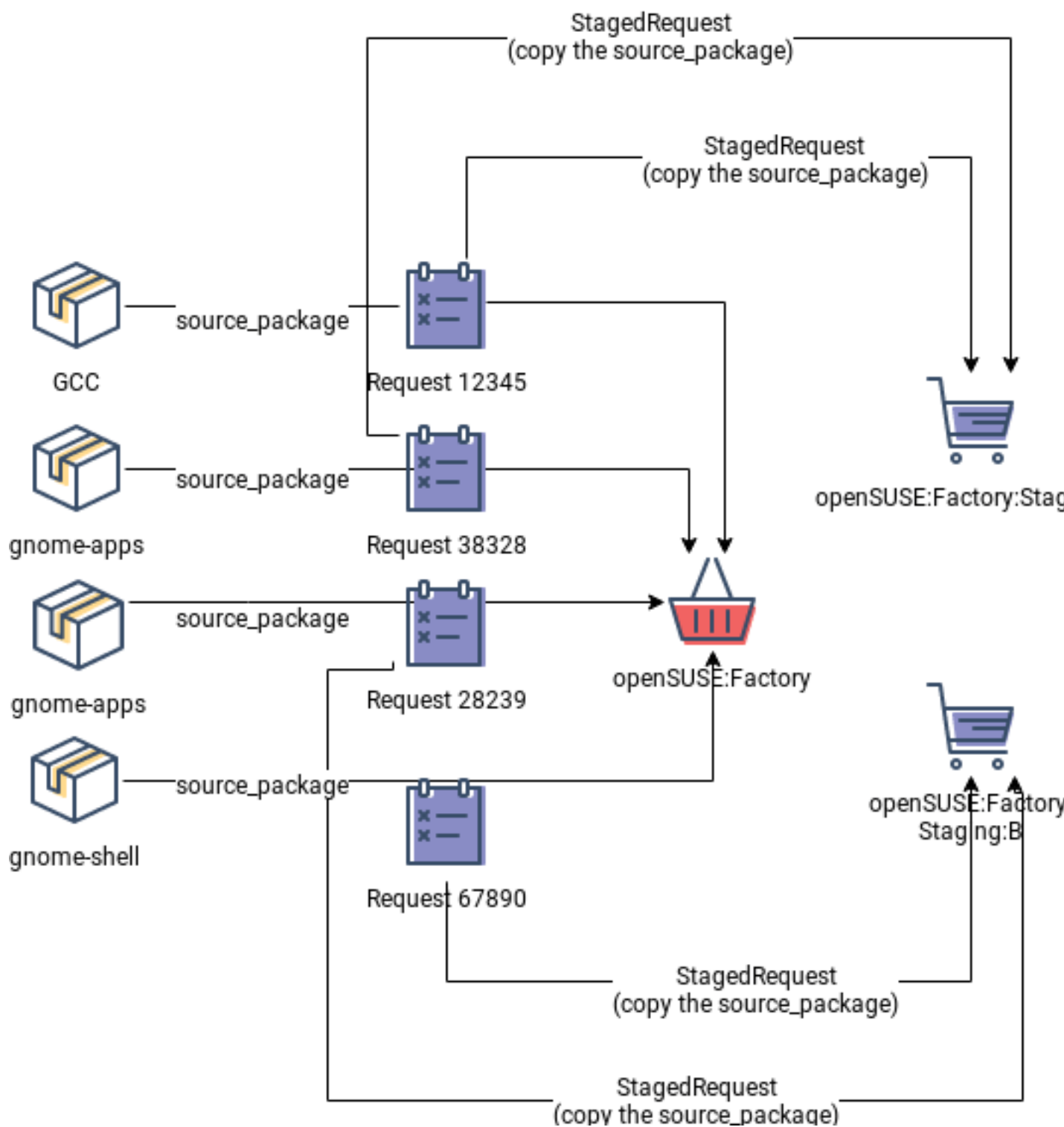


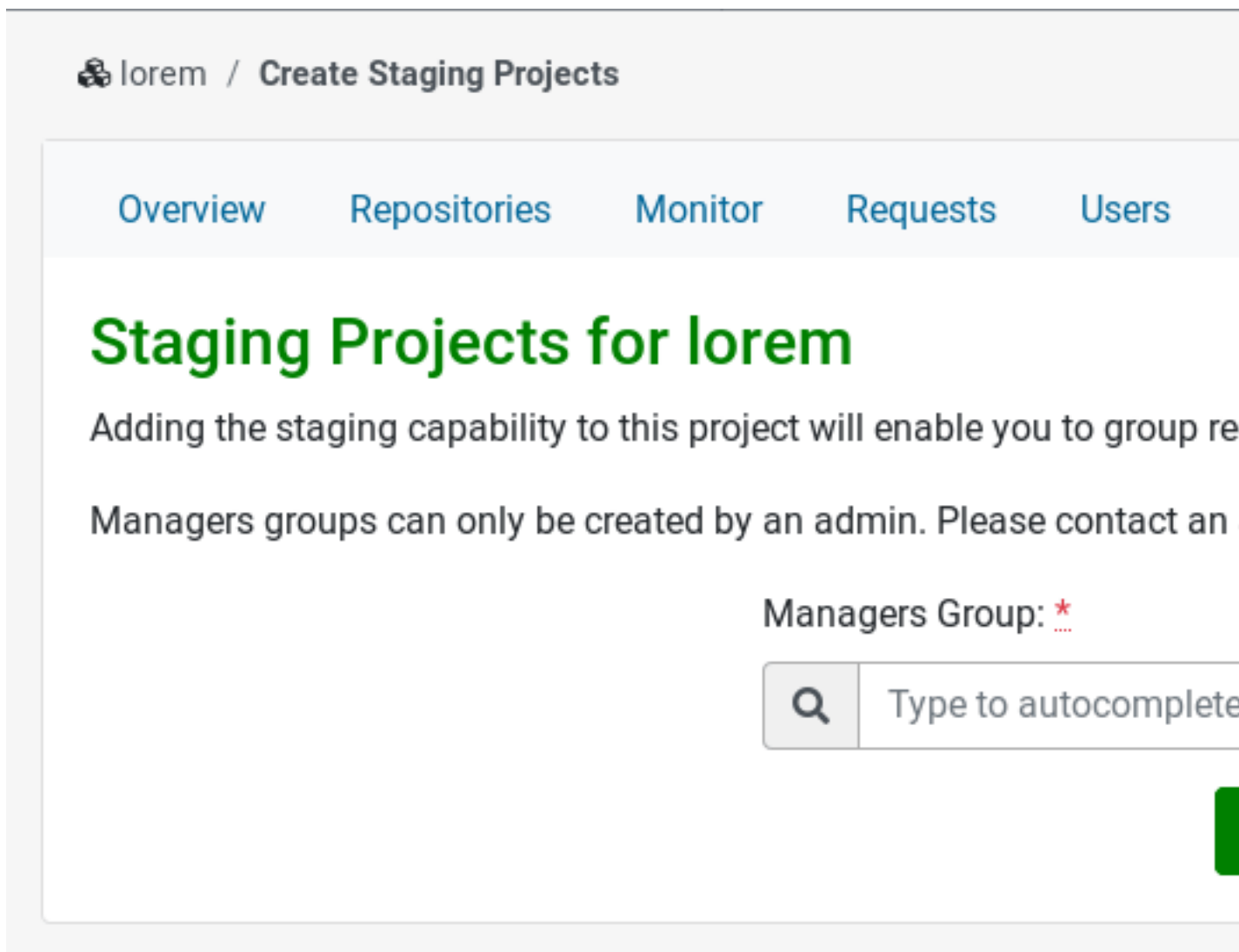
FIGURE 12.37: STAGING WORKFLOW SCHEMA

12.10.1 Creating a Staging Workflow

At the moment, it is possible to create a Staging Workflow for any kind of project unless the project is already one of the Staging Projects.

All the Staging Workflow starts in the tab 'Staging' which can be found on the project's page. It will take you to the first step to create a new Staging Workflow or to the dashboard if the Staging Workflow already exists.

The creation of a Staging Workflow will automatically create two Staging Projects as a subproject of the main project. Before creating, we need to select a group of managers, they will be in charge of assigning requests to the Staging Projects and also excluding requests from the Staging Workflow.



The screenshot shows a web interface for creating staging projects. At the top, there is a breadcrumb trail: a factory icon followed by 'lorem / Create Staging Projects'. Below this is a horizontal navigation bar with five tabs: 'Overview' (selected), 'Repositories', 'Monitor', 'Requests', and 'Users'. The main content area has a large green heading 'Staging Projects for lorem'. Below the heading, there is explanatory text: 'Adding the staging capability to this project will enable you to group re' and 'Managers groups can only be created by an admin. Please contact an'. On the right side, there is a label 'Managers Group: *' with a red asterisk and a dropdown arrow. Below this is a search input field with a magnifying glass icon and the placeholder text 'Type to autocomplete'. A green button is partially visible on the right edge.

FIGURE 12.38: CREATING A STAGING WORKFLOW FOR OPENSUSE:FACTORY



Note

An Admin should previously create the manager groups.

12.10.2 Start Using Staging Workflow

In this view, we can find all the Staging Projects with an associated request and their current state.

ories

Monitor

Requests

Users

Subprojects

Project C

Staging for home:Admin




Staging Project	Requests		
<div>Staging:A</div> <div>unacceptable</div>	<div>aut_0 </div> <div>dolor</div>	<div>autem </div>	<div>accusamus_2 </div>

FIGURE 12.39: STAGING WORKFLOW SHOW SCREEN

- Table content:
 - *Staging Project*: Shows the Staging Project name, its overall state (see legend), and the overall build progress of the packages within the project.
 - *Requests*: Show the associated requests and their current state.
 - *Problems*: Shows build problems of packages within the project and status problems reported to the Build Service's Status API by external services like openQA.
- Info section:
 - *Managers*: Shows the Staging Managers group.
 - *Empty projects*: Staging projects without assigned requests.
 - *Backlog*: List of requests that can be assigned to a Staging Project.
 - *Ready*: List of requests that were in the backlog and have an accepted review.
 - *Excluded*: List of requests excluded from this Staging Workflow.

12.10.3 Delete a Staging Workflow

Next to the title, there is a icon that allows us to delete the Staging Workflow.

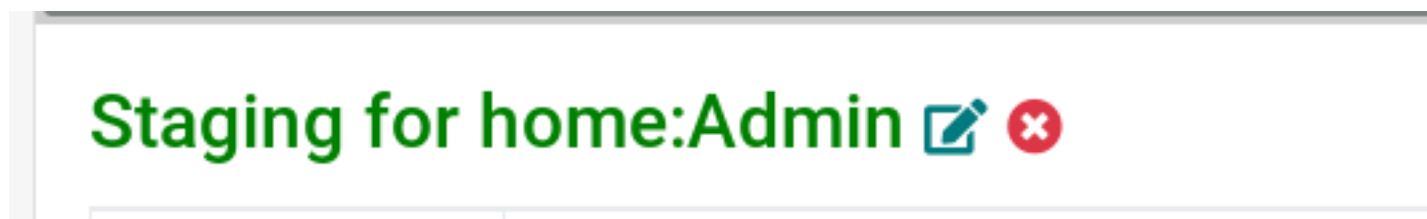


FIGURE 12.40: STAGING WORKFLOW DELETE ICON

By clicking on the delete icon on the Staging Workflow index page, we are able to delete a Staging Workflow.

By selecting the associated Staging Projects in the appearing modal window, we are able to delete them as well. If not selected, they will remain as regular subprojects.

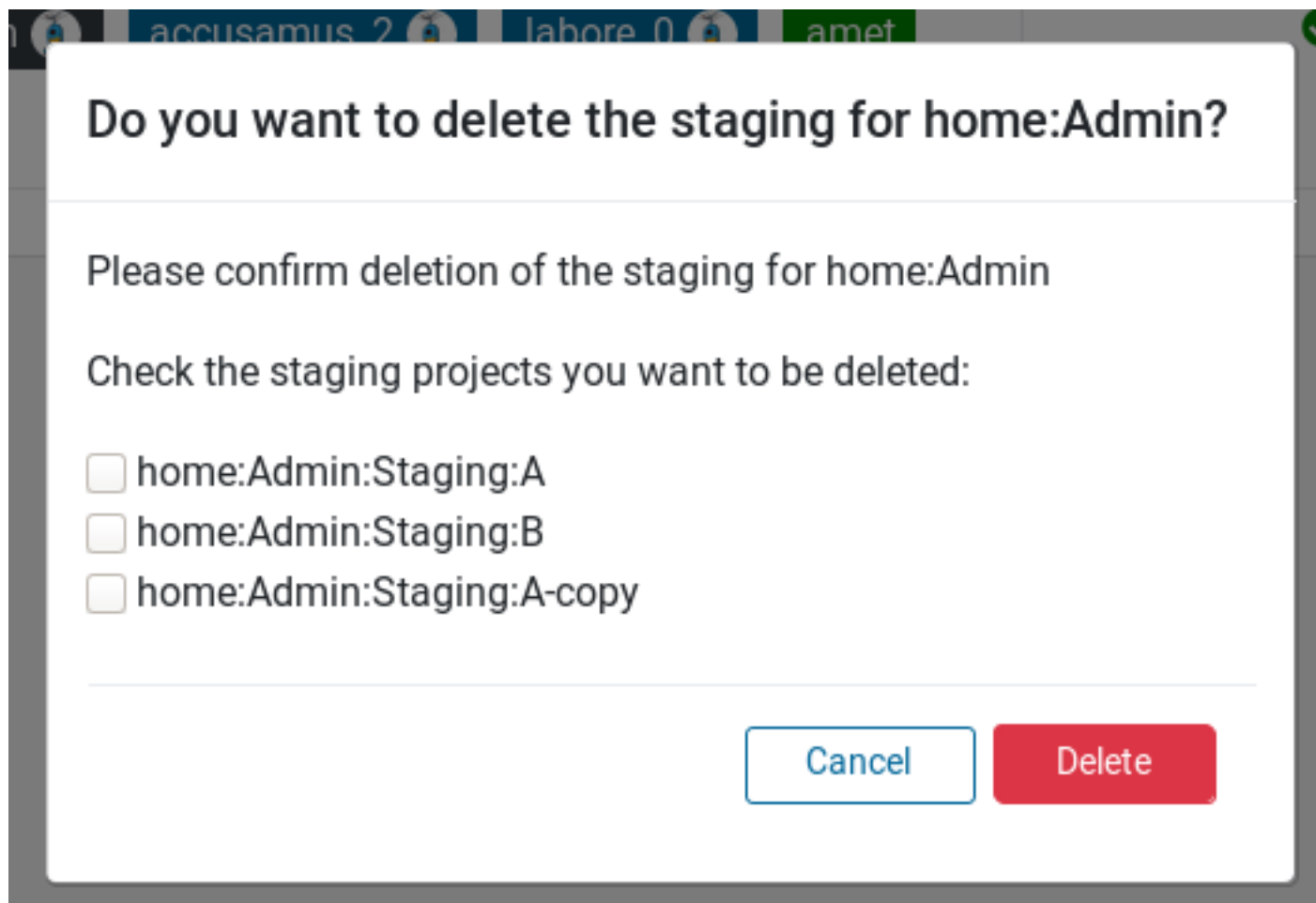


FIGURE 12.41: DELETE A STAGING WORKFLOW

12.10.4 Configure a Staging Workflow

Next to the title, there is a link to the Staging Workflow configuration's page.



FIGURE 12.42: STAGING WORKFLOW CONFIGURE ICON

From the configuration page it is possible to delete a Staging Project, create one from scratch or create a copy of an existent one. But also to change the Managers Group of the Staging Workflow.

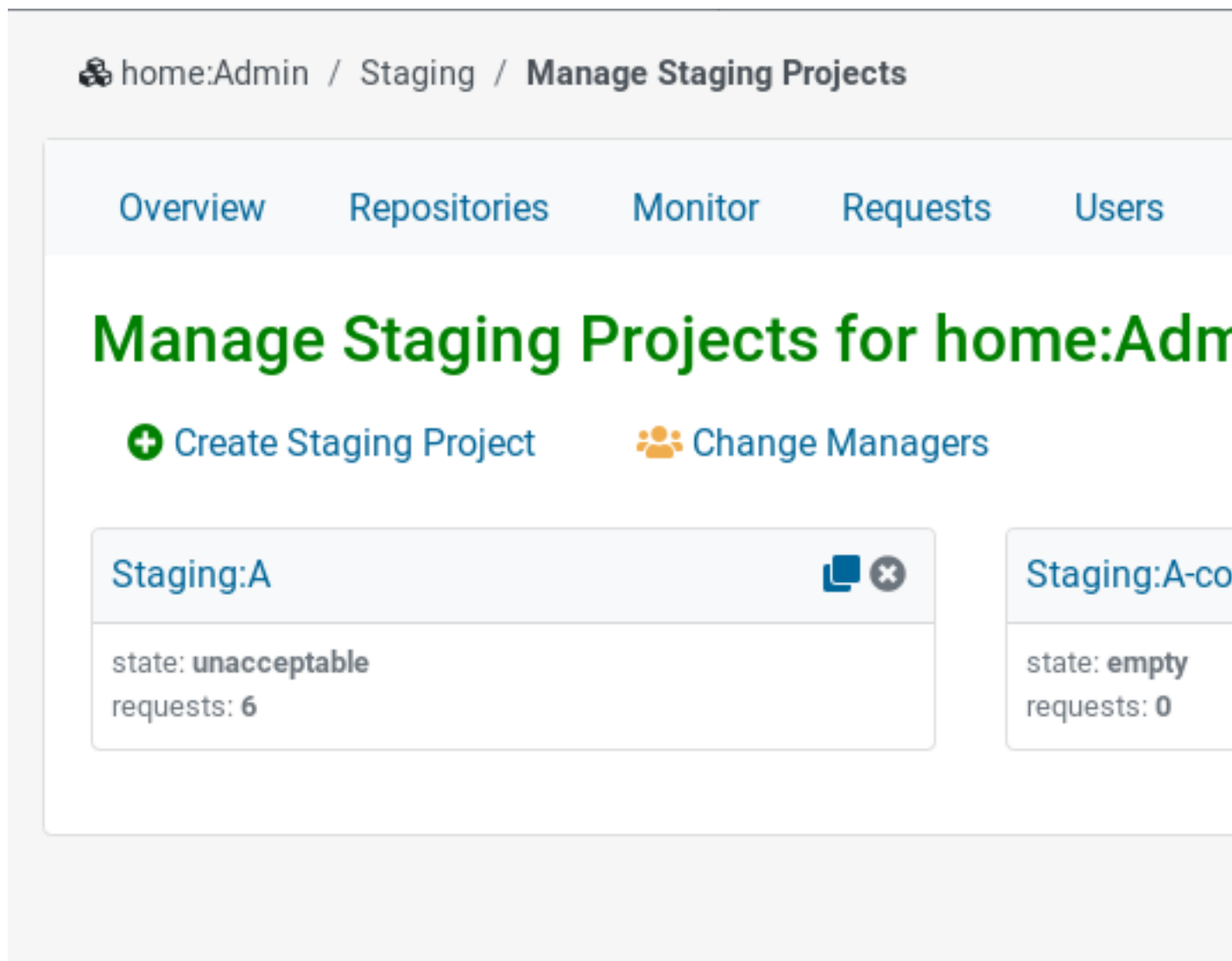


FIGURE 12.43: CONFIGURING A STAGING WORKFLOW



Note

Changing the Managers Group of a Staging Workflow will automatically unassign the old group and assign the new group to the related Staging Projects.

12.10.4.1 Create Staging Project from Scratch

Right after the creation of a Staging Workflow, two new Staging Projects are automatically created and assigned to it: Staging:A and Staging:B. However, it is also possible to create a new Staging Project from scratch.

On the Staging Workflow dashboard, click on configure icon next to the title and then on *Create Staging Project* to add a name for the new Staging Project.

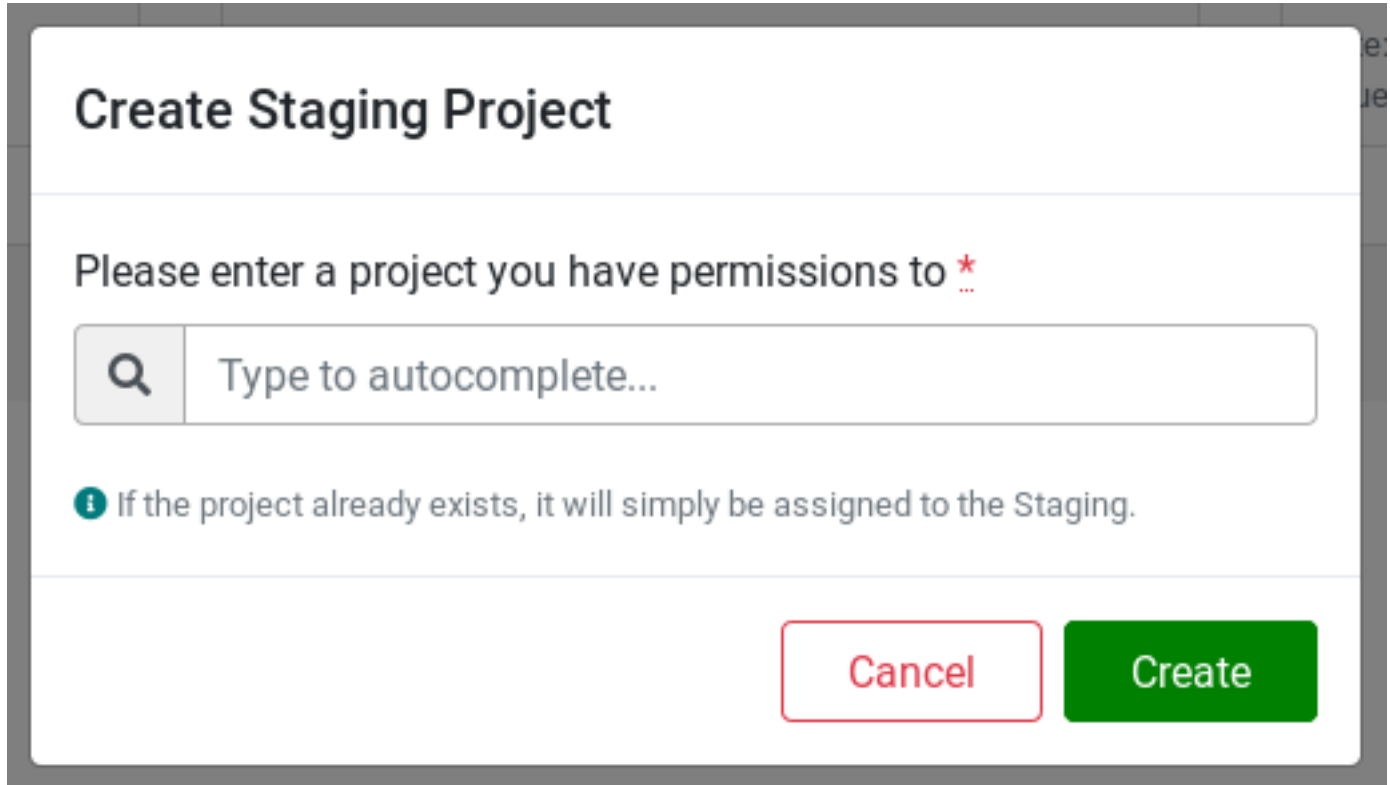
A screenshot of a 'Create Staging Project' dialog box. The title 'Create Staging Project' is at the top. Below it, a text prompt says 'Please enter a project you have permissions to *'. Underneath is a search input field with a magnifying glass icon and the placeholder text 'Type to autocomplete...'. Below the input field is an information icon (i) followed by the text 'If the project already exists, it will simply be assigned to the Staging.' At the bottom right are two buttons: 'Cancel' (outlined in red) and 'Create' (solid green).

FIGURE 12.44: CREATE A NEW STAGING PROJECT

12.10.4.2 Create Staging Project from a Template

It is possible to create a Staging Project from a template. Inside Staging Workflow's configuration page, simply choose the Staging Project you want to copy from (the template), click on its Copy icon and add a new name. The Staging Project copy is processed in the background, so there might be a delay before it shows up.

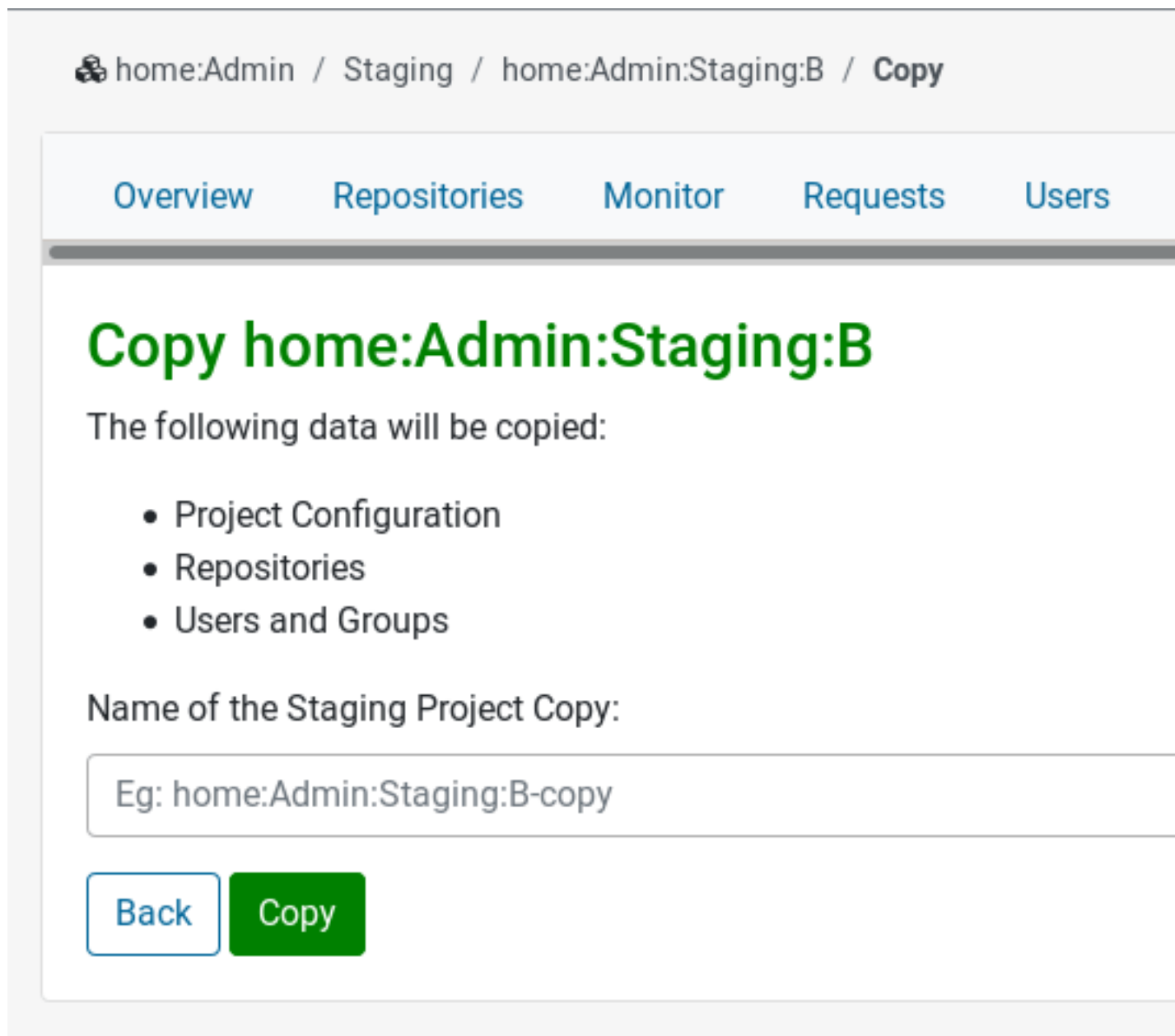


FIGURE 12.45: COPY STAGING PROJECT FROM TEMPLATE

12.10.5 Staging Project

A Staging Project contains requests assigned by a Staging Manager. There is an overview page for a Staging Project, where you can find more detailed information about the requests, reviews and checks.

Overview

Repositories

Monitor

Requests

Users

home:Admin:Staging:A

Packages

aut_0



autem



accusamus_2



labore_0



ame

Status



Untracked requests

None



Obsolete requests

[#369 \(declined\)](#), [#370 \(declined\)](#)



Missing reviews

[accusamus_2](#) by Requestor, labore_0



Building repositories

None



Broken Packages

None



Checks

None

FIGURE 12.46: LOOKING INTO A STAGING PROJECT

- *Obsolete Requests*: Requests that were declined, revoked or superseded.
- *Missing Reviews*: Requests with pending reviews.
- *Building Repositories*: List of packages that are still building.
- *Broken Packages*: List of packages with failing builds.
- *Checks*: List of checks of the Staging Project.

All the actions performed on requests that are assigned to the Staging Project are tracked. They are listed in the 'History' section.

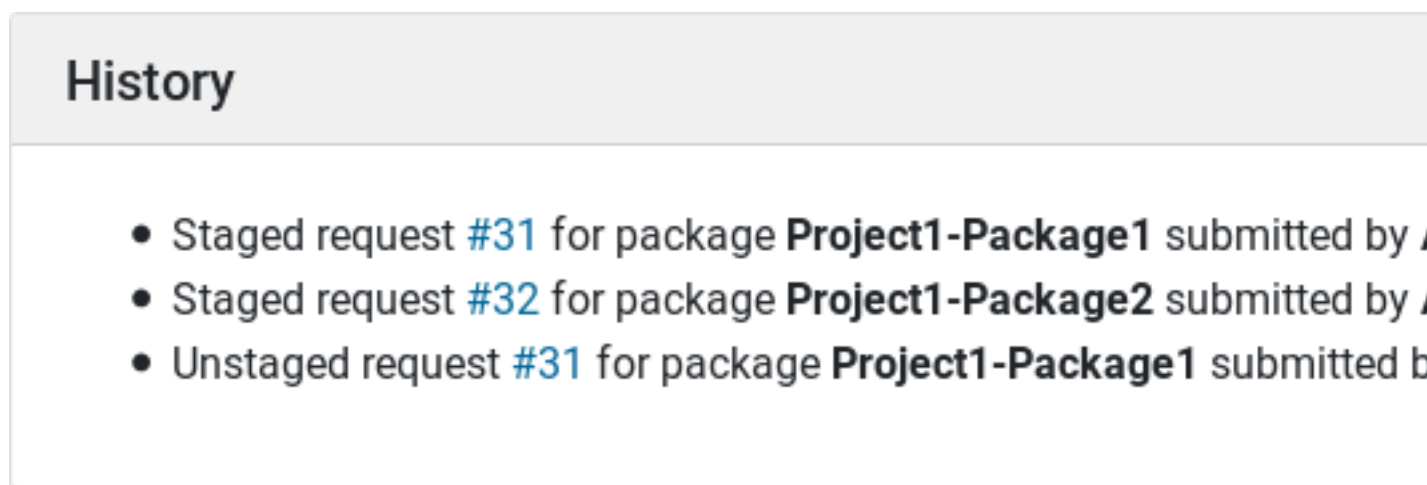


FIGURE 12.47: HISTORY

12.10.6 Working with Requests in Staging Workflow

12.10.6.1 Exclude Requests

Sometimes it can be useful to exclude a request and don't let it be available in the Backlog. This can prevent the staging project from being assigned with requests we are sure are causing conflicts, have some missing dependencies or have to wait for other request to be accepted.

By clicking on the 'Excluded' link on the 'Infos' section, it is possible to exclude requests or bring back already excluded ones.

Excluded Requests

Show entries

Request	↑↓
reiciendis	Missing dependencies.

Showing 1 to 1 of 1 entries

[+ Exclude request](#)

FIGURE 12.48: EXCLUDE REQUESTS

13 Basic Concepts and Work Styles

These best practices should be known by every OBS user. They describe how to set up projects and working with own or foreign sources.

13.1 Setup a project reusing other projects sources

You can also setup your own project using the sources, spec files and patches from another project and develop within this project.

```
#osc copypac SOURCEPRJ SOURCEPAC DESTPRJ
```

By default, Open Build Service will strip the maintainer info and now make it part of your own project. To clarify, when we speak of a project, it can mean just one package or a complete set of packages with their own build dependencies.

13.2 Contributing to External Projects Directly

In case a user does not have commit permissions for a project, they can request maintainership permissions for this project. This makes sense if the user is already known to the project owners and they trust them as a maintainer. There is a way to do this via the request system of OBS, but only via osc so far:

```
# osc createrequest -a add_me maintainer PROJECT
```

13.3 Contributing to Foreign Projects Indirectly

Users who are new to a given project, either because they are new users with Open Build Service or packaging or do not have any deeper knowledge about a certain project will not have direct commit permissions. However, they can still create a copy of any package source and ask back to merge their changes. Open Build Service has support to make this easy.

Wiki reference: [User comment page \(http://en.opensuse.org/openSUSE:Build_Service_Collaboration\)](http://en.opensuse.org/openSUSE:Build_Service_Collaboration) 

14 How to integrate external SCM sources

Application development usually happens in SCM systems like git, subversion, mercurial and alike. These external sources can be used directly in OBS via source services. OBS will always keep a copy of the sources to guarantee that the build sources are still available even when the external SCM server disappears or get altered.

14.1 How to create a source service

Let OBS create a tar ball out of an SCM repository. This just creates or extend a `_service` file with some rules how to download and package sources. The actual work happens on a local build or on a service side build. Please note that you need the `obs-service-obs_scm` installed for local runs.

```
# osc add https://SOME_URL.git
```

The web interface is creating as well a `_service` file when adding an URL to a SCM system.

14.1.1 Follow upstream branches

The created `_service` file is set up to follow latest source submissions on each run and looks like this:

```
<services>
  <service name="obs_scm">
    <param name="url">https://github.com/FreeCAD/FreeCAD.git</param>
    <param name="scm">git</param>
  </service>

  <service name="set_version" mode="buildtime"/>
  <service name="tar" mode="buildtime"/>
  <service name="recompress" mode="buildtime">
    <param name="file">*.tar</param>
    <param name="compression">xz</param>
  </service>
</services>
```

This will create an obspio archive via the obs_scm service with the latest sources. This archive will get extracted at build time and be processed via the other services to build a compressed tar ball for rpmbuild. To follow a specific branch and additional parameter for "revision" is needed for the obs_scm service.

14.1.2 Fixed versions

You may want to build an archive for a fixed version, for example an official release which has been tagged by the upstream project. It is recommend to specify the mode="disabled" and to submit the archive via the following

```
# osc service runall
# osc ar
# osc commit
```

commands.

14.1.3 Avoid tar balls

Tar balls are not a requirement by OBS, but by the packaging tool, for example, rpmbuild. However, you may want to decide not to ship a tar ball inside of the src.rpm. This makes sense for large sources where the compression time and needed disk space is just considered a waste for short living builds and where full source packages are not a requirement. You can simplify your _service file in that case, but you need to help rpmbuild to work directly in the source. Since RPM will not include the OBS provided SCM sources in the src.rpm, it is also a good practice to package the _service file instead of the tar ball to give the user a chance to rebuild the src.rpm as long the external SCM server is providing the sources. The simplified _service file looks like this:

```
<services>
  <service name="obs_scm">
    <param name="url">https://github.com/FreeCAD/FreeCAD.git</param>
    <param name="scm">git</param>
  </service>

  <service name="set_version" mode="buildtime"/>
</services>
```

The spec file needs some hints to build inside the extracted sources directly. The macro can be used to switch to build tar balls or not to keep it working for stable releases where you want to provide a complete source RPM.

```
...
%define build_tar_ball 0
...
%if %{build_tar_ball}
Source0:      %{name}-%version.tar.xz
%else
Source0:      _service
%endif
...
%prep
%if %{build_tar_ball}
%setup -q
%else
%setup -q -n %_sourcedir/%name-%version -T -D
%endif
```

15 Publishing Upstream Binaries

This chapter covers main step of using OBS to publish binaries of your project for multiple distributions.


15.1 Which Instance to Use?

15.1.1 Private OBS Instance


OBS is open source project and therefore you can set up your own instance and run it by your own. The main advantage of this approach is that you can keep all your sources and build recipes unpublished if you need to (for example because of NDA). Obvious downside of this approach is that you need to maintain your own server/servers for running builds, publishing and mirroring. Also making your project public may attract some potential contributors.

More information about setting up your own private OBS instance can be found in *Book "Administrator Guide", Chapter 7 "Setting Up a Local OBS Instance"* .


15.1.2 openSUSE Build Service

Other option is to use some publicly available instance of OBS. One good example is openSUSE Build Service at <http://build.opensuse.org> . This OBS instance can be used by anybody to freely create binaries for any of the supported distributions. Big advantage is that somebody is already taking care of all the infrastructure. You can store your sources there, build your packages and got them mirrored around the world. You do not need to get your own server and configure it, you can start using it right away.



15.2 Where to Place Your Project

This part helps you to decide on how to name and where to place your project and what project structure to create. This is more important if you are sharing your OBS instance with other people like in [openSUSE Build Service \(http://build.opensuse.org\)](http://build.opensuse.org) .

15.2.1 Base Project

If there are more packages in OBS, like for example in [openSUSE Build Service \(http://build.opensuse.org\)](http://build.opensuse.org) , these packages need to be somehow divided into projects so it is easier to find what people are looking for and it is not all just one big mess.

In openSUSE Build Service, packages are divided into categories regarding their function. MySQL is in *server:database* repository, lighttpd in *server:http* and for example KMyMoney has its own subproject in *KDE:Apps*. So it is a good idea to think about in what category available on the OBS your application will fit the best.

If you need whole project for yourself - for example some of your dependencies is being built in the same project, you need to request creating subproject. In openSUSE Build Service, this is done through asking OBS admins for it on [buildservice mailing list \(mailto:buildservice@opensuse.org\)](mailto:buildservice@opensuse.org) . Its archive and link for subscribing can be found at <https://lists.opensuse.org/manage/lists/buildservice.lists.opensuse.org/> .

If you need to just put your package somewhere, you can create it in your home project and then send submitrequest to the project you want your package to get included in.

15.2.2 Supporting Additional Versions

If you want to support more than one version of your program, you need to use several projects. The same package cannot be contained in the same project multiple times.

15.2.2.1 Stable and Development Versions

Let's assume that you have found project suitable for your program. Some projects already have something like *STABLE* and *UNSTABLE* subprojects. So you can use these, if you discuss it with maintainers of these project. Other way is to ask somebody from the maintainers of the project to create either these subprojects (if they do not exist) or something similar. Always try to discuss it with the maintainers of the project. They might have good ideas, suggestions and may help you in various ways.

15.2.2.2 Multiple Stable Versions

If you want to support multiple version, you would need more projects than just two as suggested in previous section. These special projects should contain versions they are supposed to support in their name. If you are creating them under some project you are sharing with other packages, having you package name in the name of projects is a good idea as well.

GNOME is a good example: There is the *GNOME* project and many subprojects. Among them are, for example, *GNOME:STABLE:2.30*, *GNOME:STABLE:2.32*, and *GNOME:STABLE:3.0*. These projects hold different stable versions of GNOME with latest fixes.

15.3 Creating a Package

Packaging is quite a complex topic. Instead of trying to cover it in this book, it is a good idea to start with available internet documentation. One of the recommended online resource is Portal:Packaging on openSUSE wiki. You can find it at <http://en.opensuse.org/Portal:Packaging>. It contains links to several packaging tutorials and other packaging related documentation.

15.4 Getting Binaries



Note

The following sections discuss feature available only in openSUSE Build Service—a freely available instances of OBS.

For a nice download page for your software published on openSUSE Build Service, use the openSUSE download page. You can include it for example using either `iframe` or `object` on newer websites. An example of download page can be following one <http://software.opensuse.org/download.html?project=openSUSE:Tools&package=osc>. You can see how it looks like in *Figure 15.1, “openSUSE download page for package from OBS”*. It contains links to the packages and instructions how to install them.

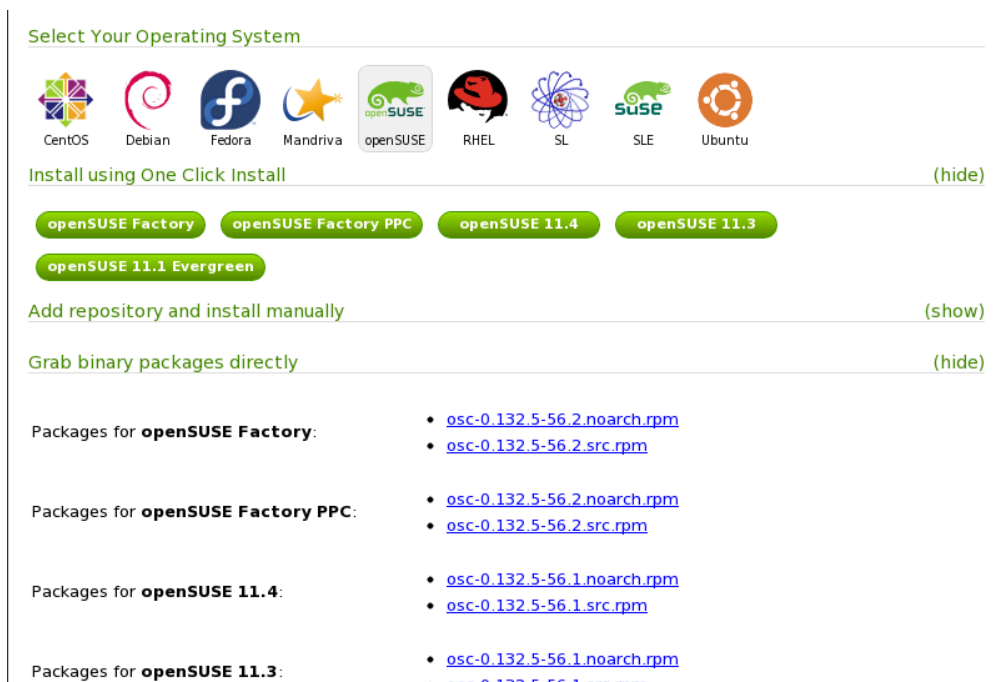


FIGURE 15.1: OPENSUSE DOWNLOAD PAGE FOR PACKAGE FROM OBS

URL always has to start with <http://software.opensuse.org/download.html?>. You can attach any number of &-separated parameters. But at least two of them - *project* and *package* - are required. All parameters with descriptions can be found in [Table 15.1, "Parameters for Download Page"](#).

TABLE 15.1: PARAMETERS FOR DOWNLOAD PAGE

parameter	description
<u>project</u>	Project in which your package is located
<u>package</u>	Name of your package as it is specified in OBS
<u>bcolor</u>	Background color in hex (for example <u>bcolor=004</u>) to make the download page better match your project page
<u>fcolor</u>	Text color in hex (for example <u>fcolor=fff</u>) to make the download page better match your project page
<u>acolor</u>	Link color in hex (for example <u>acolor=ff0</u>) to make the download page better match your project page

parameter	description
<u>hcolor</u>	Highlight color in hex (for example <u>hcolor=0ff</u>) to make the download page better match your project page

15.4.1 Examples

Now we will take a look at how to include the download page into your project pages. As an example, we will use the osc client from the *openSUSE:Tools* project. To demonstrate the colors change, we will use theme that would match Midnight Commander.

First we will start with recent web page supporting new standards. On such a website, we would use object to include download code:

```
<object type="text/html"
  data="http://software.opensuse.org/download.html?
project=openSUSE:Tools&package=osc&bcolor=004&acolor=ff0&fcolor=fff&hcolor=0ff"
  width="100%" height="100%">
  <param name="src"
    value="http://software.opensuse.org/download.html?
project=openSUSE:Tools&package=osc&bcolor=004&acolor=ff0&fcolor=fff&hcolor=0ff" />
  Your browser doesn't support objects, please continue to the
  <a href="http://software.opensuse.org/download.html?
project=openSUSE:Tools&package=osc&bcolor=004&acolor=ff0&fcolor=fff&hcolor=0ff">
  download page</a>
</object>
```


If you are using PHP on your server, you can make it easier by using following code:

```
<?php
  $url = "http://software.opensuse.org/download.html?
project=openSUSE:Tools&package=osc&bcolor=004&acolor=ff0&fcolor=fff&hcolor=0ff";
  echo '
<object type="text/html"
  data="' . $url . '"
  width="100%" height="100%">
  <param name="src"
    value="' . $url . '" />
  Your browser doesn't support objects, please continue to the
  <a href="' . $url . '">download page</a>
</object>
';
?>
```

If you are running some legacy website, you might have to use iframe :

```
<iframe src="http://software.opensuse.org/download.html?
project=openSUSE:Tools&package=osc&bcolor=004&acolor=ff0&fcolor=fff&hcolor=0ff"/>
```

16 Bootstrapping

This chapter explains Boot strapping. In this chapter, You would learn how you could have other OBS projects and packages to your local OBS instance after your OBS install. There are some useful OSC commands examples and OBS working mechanism explanation in this chapter also. Basically this chapter is a copy from Build Service portal. For information about OBS bootstrapping on the Build Service portal, see https://en.opensuse.org/openSUSE:Build_Service_private_instance_boot_strapping .

16.1 The Issue

If you create a private instance of an OBS it is likely to be fully independent. This means that your OBS needs to build its full reference tool chain. This process—called bootstrapping—presents the same problem as the Chicken and the Egg, which one came first! In other words, you need to create a tool chain with the tool chain that you want to create.

16.2 A Cheat Sheet

16.2.1 Creating Your First Project

Log on to the Web API. The default user Admin, with the password `opensuse` is available. Create your own login and password and set yourself as Admin. Log on to the Web UI as Admin and click on the icon *Configuration* and add the openSUSE Build Service as the remote instance. Select from under *Locations* › *Projects*. At the end of the list, click *Add Project*. Give it a name (e.g. Meego-test) Select your new project and create a sub-project 0.1. You have now a project Meego-test:1.0 Sub projects are handy to propagate Access Control Lists (acl) and for creating the version as a sub project simplifies the user and project administration.

16.2.2 Importing Your Base Linux Project

We are now going to import the base project. I will describe two methods, one where you have a login on a remote OBS instance, one where you have only access to the rpm repository. In both cases you will need access to binary and source rpm.

16.2.2.1 With a login on a remote OBS

The `osc` copypac (I assume that you have installed the `osc` package on your workstation) has an option `-t` which enables copying towards a remote target OBS instance. `osc help` and `osc help command` will advise on how to use these. First you need to import the project configuration.

```
$ export PROJECT=MeeGo-test:0.1
$ osc -A http://api-url-source-obs meta prjconf $PROJECT > my_project.conf
$ osc -A http://api-url-target-obs meta prjconf -F my_project.conf $PROJECT
```

Then import the project. As you might have some Links in the project that you import, it is a good idea to keep the source and target project names identical.

```
$ PRJ=ProjectToCopy; for i in `osc -A http://api.source.obs.domain ls $PRJ`; do \
osc -A http://api.source.obs.domain \
copypac -t http://api.target.obs.domain $PRJ $i $PRJ ;done
```

16.2.2.2 Without a Login on a Remote OBS

If you have access only to the repositories of your source reference target, then your life will be a bit more difficult. My advice would be to recheck if you find you cannot get a login on a public OBS service - such as provided by openSUSE or MeeGo - before proceeding this way. You will not be able to import the project config and you will have to create it by hand. This is too long to be covered in this HowTo. For more explanation about Build Service project config, see http://en.opensuse.org/openSUSE:Build_Service_prjconf.

Then you need to download all your rpm source on to a local machine and import it into your project with the command.

```
$ osc importsrcpkg
```

16.2.2.3 Bootstrapping

To initiate the build process, we will copy the rpm binary from the source OBS of the source project. These binary RPMs, from which we will remove any reference to release and version, will be used to trigger the first build. The OBS appliance will recompile all the RPMs until all RPMs in the project have been compiled only with packages compiled from their source code. Some base packages (e.g. tool chains) will be compiled several times during that process. Alternatively, you can at first build against a target which is similar to the base that you need

to bootstrap in lieu of building against your own base and change the build reference to your bootstrap base once that the first build has been successful. Remember that you can also build against remote baseline. Double check that the preliminary step have been executed correctly. You must have already: copied a Linux base distribution in an OBS project defined a build target for that base project.

If you have not defined a build target, the necessary directory structure will not exist. This is a mandatory step of preparation. Stop the scheduler as it will create a mess if the system is not stable:

```
# rcobsscheduler stop
```

* Add binaries to the :full directory of the Project ssh onto the OBS server. Now go to the project's build directory, and create a directory called ":full". Note : standard is the default name of your Build repository as defined in your project. It might change depending on who created the initial build repo.

```
# cd /obs/build/$PROJECT/standard/i586
```

This directory structure should already exist. If not, there is a problem (note that /obs is link and the target may vary with your implementation). Now create the ":full" directory. \$ mkdir :full Copy over all the binary RPMs of the project you are trying to build from scratch. These RPMs should have the release and version numbers stripped from them. e.g. alsa-util-1.0.22-2.7.i586.rpm -- should be -- alsa-utils.rpm Note : If the original project has a :full directory you can copy from there to avoid the issue of stripping version and release numbers. * Add binaries to the :full directory of the Project. Change all user/group privileges under /srv/obs/build/ to "obsrun"

```
# chown -R obsrun:obsrun /srv/obs/build
```

If you leave root as owner of :full, it will still build but the scheduler will fail (almost silently) to upgrade :full with the latest built packages. Except in very special cases, it is very unlikely that you want to do so. * Start the OBS scheduler

```
# rcobsscheduler start
```

* Force the obs to reindex your new :full directory. It will send an event to the scheduler which will create a file named :full.solv

```
# obs_admin --rescan-repository $OBS-PROJECT $REPO $ARCH
```

16.2.2.3.1 Troubleshooting

At that time you should see your project restarting to build. If that would not be the case. * check that your files in your target :full directory are all own by the user obsrun. The following command should not return any file name.

```
#find /obs/build ! -user obsrun
#chown -R obsrun:obsrun /obs/build (will correct ownership issue)
```

* Force the obs to reindex your new :full directory. It will create a file named :full.solv

```
$obs_admin --rescan-repository $OBS-PROJECT $REPO $ARCH
```

* Check that your rpm are valid (e.g. not damaged during transfer)

```
#cd /obs/build/$PROJECT/standard/i586:full
#for I in `ls *.rpm` ; do rpm -qlp $I >/dev/null; if [ $? -ne 0 ] ;then echo $I >>../
error.lst ; fi ; done
#cat ../error.lst (must be empty, all rpm in error needs re-installation)
```

* Still not working, get a look in the log files in the directory /obs/log. You can start by having a look at /obs/log/scheduler_TARGET_ARCH.log and search from the end for the string "expanding dependencies". You will find from there why the scheduler fails.

```
#tail -f /obs/log/scheduler_i586.log
```

16.3 Creating a First Project

After creating a dedicated user via the Web API, log onto the Web UI again with your new login. Open your home project and create a sub project called "MyTest". To add a package in your new Home project, simply create a link [link Package from other Project] with one of the packages recently copied in your new OBS instance (see previous chapter Import your base project). Pick up a small one to speed compilation time. Click on the "+" near Build Repositories to add a repository. Move to the end of the page where all the standard Linux distributions are listed and click on [Advance]. Give a name to your repo, e.g. my-test and pick from the list the project/repo that you have just imported and rebuilt. This will request the OBS to build your new Home project against that repository. You can now check out your Home project with the osc command, modify a file or two and at your next check-in, the OBS will rebuild your Home project. If your reference project changes, the OBS will also rebuild your Home project.

17 **osc** Example Commands

This chapter explains and shows OSC commands examples. You could use OBS much more efficiently with OSC commands. `$man OSC` will show you [GLOBALOPTS], SUBCOMMAND, [OPTS][ARGS...]. You also could find some OSC commands examples from OBS Build Service portal. This chapter will take every OSC command examples from OBS Build Service portal and describes it in here. You could visit Build Service portal OSC command explanation at https://en.opensuse.org/Build_Service/CLI ↗.

17.1 Package Tracking

With osc it is also possible to manage packages in a SVN like way. This feature is called package tracking and has to be enabled in `~/.osrc`'s [general] section

```
# manage your packages in a svn like way
do_package_tracking = 1
```

Add a new package to a project

```
osc mkpac [package]
```

Add an already existing directory and its files to a project

```
osc add [directory]
```

Remove a package and its files from a project

```
osc deletepac [package]
```

All the commands above only change your local working copy. To submit your changes to the buildservice you have to commit them (`osc ci -m [message]`). The status command also displays the state of the packages

```
osc st
```


18 Advanced Project Setups

These best practices describe more complex setups, for example how to rebuild an entire stack with minimal effort.

18.1 Rebuilding an Entire Project with Changes

18.2 Integrating Source Handling

18.3 Using OBS for Automated QA

19 Building Kernel Modules

20 Common Questions and Solutions

This currently an unsorted list of asked questions.

20.1 Working with Limited Bandwidth

Packages can contain large files, esp. some tar balls can become quite large, in some real life examples several hundred mega bytes. This can be a problem when you need to work on the package via a slow connection.

20.1.1 Using the Web Interface

The web interface is the easiest way to edit simple things without the need of the checkout.

Disadvantages are

- Not the preferred solution for power packagers
- No local build possible
- Still a significant bandwidth is needed compared to the size of the edited file.

20.1.2 Using **osc** with Size Limit

osc offers to skip files with a certain size (specified with **-l** switch) on checkout. The limit is stored locally and you can also run an update later without downloading any large files. All other files can be edited, diffed and committed as usual.

Disadvantages are

- The checkout is incomplete
- No local build possible

20.1.3 Using **download_url**

Manage your large files via source services. The easiest way is to use **osc add \$URL** which just stores a small **_service** file. The check will not contain the large files by default, but they get downloaded when needed via the service. However, they will never get committed, so this

is the best approach when you have a fast downstream, but slow upstream like with standard DSL connections. Also other users can trust your tar ball, esp. important when you do version upgrades on foreign packages.

Disadvantages are

- The generated files have the `_service:` prefix in check out (but not during build).

20.1.4 Using Source Services in trylocal Mode

Manage your large files via source services in try local mode for example with `download_url` or `download_files` service. This means you can be flexible depending on your current connection without changing the setup. The service is generating the file on the server side when you decide not to commit it, but you can also decide to commit it and avoid the `_service:` prefix on the files. Also other users can trust your tar ball, esp. important when you do version upgrades on foreign packages.

Disadvantages are

- A checkout may still need the size limit switch when last commit contained the large files.

VI Reference

- 21 OBS Architecture **196**
- 22 OBS Concepts **201**
- 23 Build Process **211**
- 24 Build Containers **214**
- 25 Source Management **218**
- 26 SCM Bridge **220**
- 27 Supported Formats **224**
- 28 Request And Review System **229**
- 29 Image Templates **234**
- 30 Multiple Build Description File Handling **236**
- 31 Maintenance Support **238**
- 32 Binary Package Tracking **249**
- 33 Scheduling and Dispatching **252**
- 34 Build Constraints **256**
- 35 Building Preinstall Images **270**
- 36 Authorization **271**
- 37 Quality Assurance(QA) Hooks **274**

21 OBS Architecture

21.1 Overview Graph

Open Build Service (OBS) is not a monolithic server; it consists of multiple daemons that fulfill different tasks:

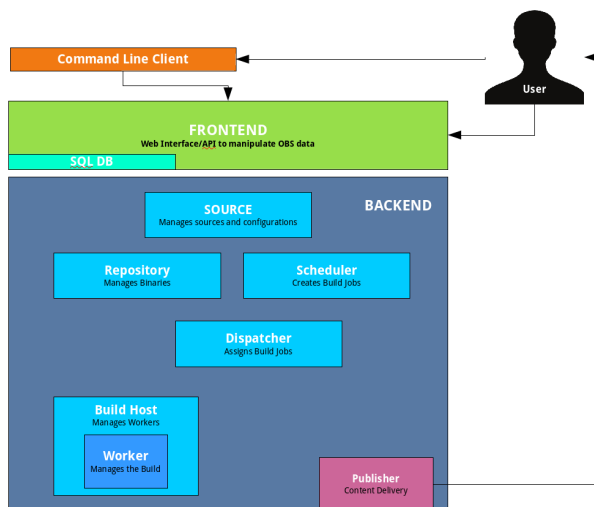


FIGURE 21.1: SIMPLIFIED OBS COMPONENT OVERVIEW

The OBS Backend manages the source files and build jobs of the OBS.

Source Server

Maintains the source repository and project/package configurations. It provides an HTTP interface, which is the only interface for the Front-end and It may forward requests to further back-end services.

The Source Server keeps track of all sources that are available for building. It takes care of file deduplication so that every source file is stored only once. This is done by keeping track of the MD5 hashes of the files in combination with the file names. All revisions of committed sources are stored and will not be deleted. This guarantees the ability to get the source for every delivered binary package.

Each OBS installation has one Source Server only. It maintains the "sources", "trees" and "projects" directories.

Repository Server

A repository server provides access to the binaries via an HTTP interface. It is used by the front-end via the source server only. Workers use the server for registration, requesting the needed binaries for the build jobs and storing the result. Notifications for schedulers are also created by repository servers. Each OBS installation has at least one repository server. A larger installation using partitioning has one on each partition.

Scheduler

A scheduler calculates the need for build jobs. It detects changes in sources, project configurations or in binaries used in the build environment. It is responsible for starting jobs in the right order and integrating the built binary packages. Each OBS installation has one scheduler per available architecture and partition. It maintains the content of the "build" directory.

Dispatcher

The dispatcher takes a job (created by the scheduler) and assigns it to a free worker. It also checks possible build constraints to verify that the worker qualifies for the job. It only notifies a worker about a job; the worker downloads the needed resources itself afterwards. Each OBS installation has one dispatcher per partition, one of which is the master dispatcher.

The dispatcher tries to assign jobs fairly between the project repositories. For this the dispatcher maintains a **load** per project repository (similar to the Unix system load) of used build time. The dispatcher assigned jobs to build clients from the repository with the lowest load (thereby increasing its load). It is possible to tweak this mechanism via dispatching priorities assigned to the repositories via the `/build/_dispatchprios` **API** call or via the **dispatch_adjust** map in the `BSConfig.pm` configuration file. See the dispatch priorities in reference guide for more details.

Publisher

The publisher processes publish events from the scheduler for finished repositories. It merges the build result of all architectures into a defined directory structure, creates the needed metadata, and may sync it to a download server. It maintains the content of the "repos" directory on the back-end. Each OBS installation has one publisher per partition.

Signer

The signer handles signing events and calls an external tool to execute the signing. Each OBS installation usually has one signer per partition and also on the source server installation.

Source Service Server

The Source Service Server helps to automate processes for **continuous integration**. The server can call different services for different tasks. It can download sources from websites and version control systems such as subversion and git. Services can also include working on the source to extract spec-files from archives, repacking the archives or adjusting version numbers in spec files. It is also often used to enforce policies by running checks. A failed check will appear as broken source and blocks a package from building.

The Source Service Server is optional and currently only one Source Service Server is supported.

Download on Demand Updater (dodup) (OBS version 2.7 or later)

The download on demand updater monitors all external repositories which are defined as download on demand resources. It polls for changes in the metadata and re-downloads the metadata in case. The scheduler will be notified to recalculate the build jobs depending on these repositories afterwards. Each OBS installation can have one dodup service running on each partition.

Delta Store (OBS version 2.7 or later)

The delta store daemon maintains the deltas in the source storage. Multiple obscpio archives can be stored in one deltastore to avoid duplication on disk. This service calculates the delta and maintains the delta store. Each OBS installation can have one delta store process running next to the source server.

Worker

The workers register with the repository servers. They receive build jobs from the dispatcher. Afterwards they download sources from the source server and the needed binaries from the repository server(s). They build the package using the build script and send the result back to the repository server. A worker can run on the same host as the other services, but most OBS installations have dedicated hardware for the workers.

21.2 Communication Flow

The communication flow can be split into the following major parts:

1. communication between users and front-end
2. communication between front-end and source server

3. communication between source server and other back-end components, in particular the repository servers.
4. communication between the back-end and the stage server to publish build results

The user uses the front-end (via tools like **osc**) to communicate with the Open Build Service. The front-end is providing a web interface and also an API. The front-end is implemented as a Ruby on Rails application. All communication happens via the HTTP protocol (usually encrypted, meaning HTTPS is used).

The communication between the front-end and the back-end also uses the HTTP protocol, using the back-end source server as the gateway to most other back-end components.

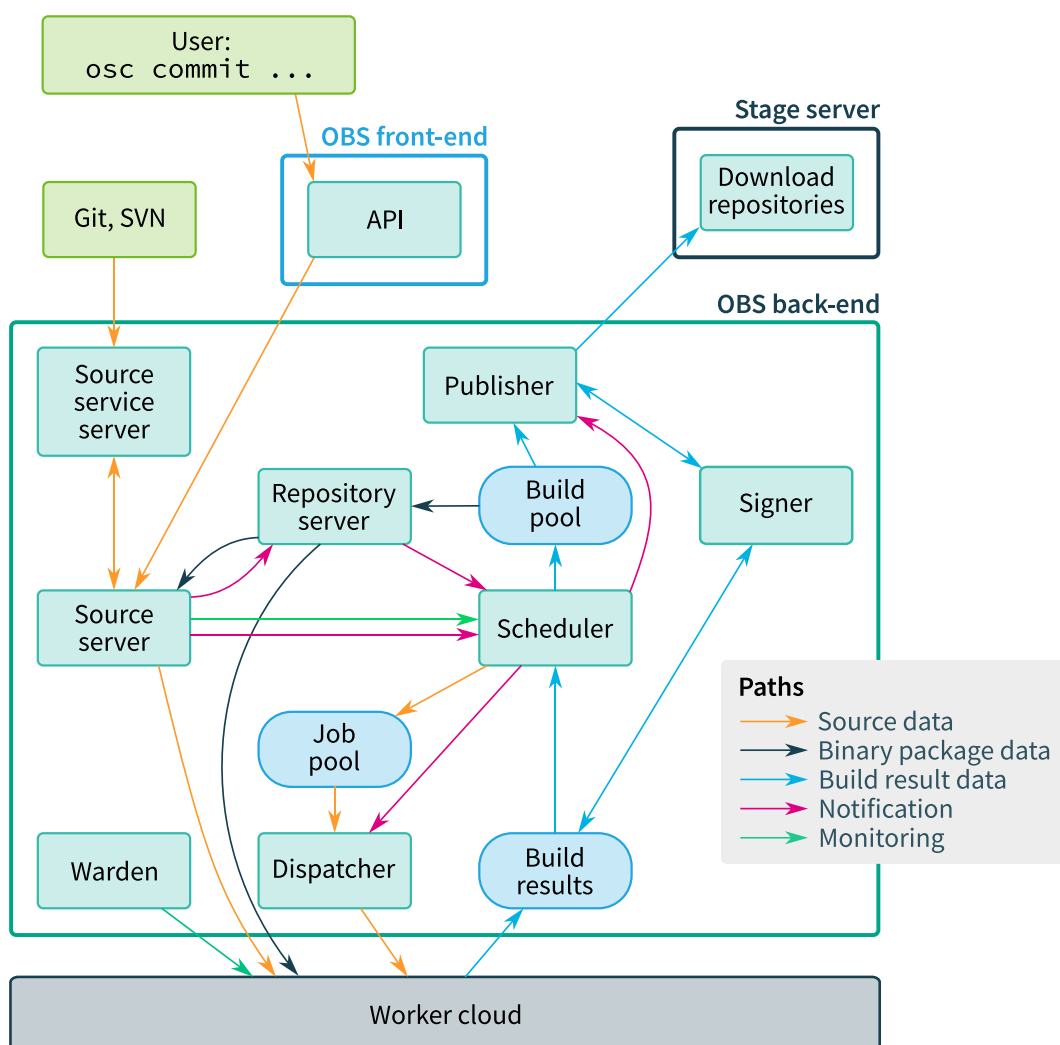


FIGURE 21.2: OBS COMMUNICATION (SIMPLIFIED)

The figure *Figure 21.2, “OBS Communication (Simplified)”* shows the communication flow between the OBS components if a package source (for example, a `_service` file) was updated:

1. The new source file is uploaded with an HTTP PUT operation to the **front-end**. The front-end authenticates and checks the permissions of the user.
2. If the user has appropriate permissions, the new file will be sent to the back-end **source server** via an HTTP PUT request. The source server stores the changed source under revision control.
It then checks whether this change makes source service runs necessary. If so, the **source service server** is informed via an HTTP PUT request of the `_service` file to run the requested services.
3. The **source service server** runs all required source services. For example, it could download the requested revision from a Git server. After running all services, it delivers the final sources back to the **source server**, which then stores these under revision control.
4. The **source server** then notifies the **schedulers** for each hardware architecture required about the change of the package via an event.
5. The **scheduler** then recalculates package and project state. If all build requirements are fulfilled, a build job is created and added to the **job pool**.
6. The **dispatcher** is notified and selects a free **worker** which meets the build constraints for the job and sends the job to it.
7. The **worker** downloads the sources from the **source server** and all required binary packages from the **repository server**. The package then will be built.
The **worker** is monitored by the **warden** service to detect any **worker** crashes.
8. If the build succeeds, the build results (including build logs) are uploaded to the **scheduler**.
If requested, the **signer** signs the packages.
9. The **scheduler** recalculates the project status, checking whether dependent packages need to be rebuilt. If not, it requests the **publisher** to publish the build results.
10. The **publisher** will create an updated version of the output repository and request the **signer** to sign the repository metadata.

22 OBS Concepts

We describe here the high-level concepts: how Open Build Service is designed, manages its content and is supposed to work.

22.1 Project Organization

All sources and binaries which are hosted inside of OBS are organized into projects. A project is the container defining a larger task. It defines who is working there.

22.1.1 Project Metadata

A project is configured in the project `/source/$PROJECT/_meta` path. It can be edited in the web interface using the **RAW Config** tab or via command line with

```
osc meta prj -e $PROJECT
```

This file contains:

- Generic description data in `title` and `description` elements.
- An ACL list of users and groups connected with a role. The `maintainer` role defines the list of users permitted to commit changes to the project.
- A number of flags controlling the build and publishing process and possible read access protections.
- A list of repositories to be created. This list defines what other repositories should be used, which architectures shall be built and build job scheduling parameters.

The following flags can be used to control the behavior of a package or project. Most of them can also be limited to specified repositories or architectures.

- **build** defines whether package sources should get built. If enabled, it signals the scheduler to trigger server-side builds based on events like source changes, changes of packages used in the build environment or manual rebuild triggers. A local build via CLI is possible independent of this flag. Default is enabled.
- **publish** can be used to enable or disable publishing the build result as repository. This happens after an entire repository has finished building for an architecture. A publish also gets triggered when the publish flag is enabled after a repository finishes the build. Default is enabled.
- **debuginfo** can be used to modify the build process to create debuginfo data along with the package build for later debugging purposes. Changing this flag does not trigger rebuilds, it just affects the next build. Default is disabled.
- **useforbuild** is used to control if a built result shall be copied to the build pool. This means it will get used for other builds in their build environment. When this is disabled, the build has no influence on builds of other packages using this repository. In case a previous build exists the old binaries will be used. Disabling this flag also means that "wipe" commands to remove binary files will have no effect on the build pool. Changing this flag does not trigger rebuilds, it just affects the next build. Default is enabled.
- **access** flag can be used to hide an entire project. This includes binaries and sources. It can only be used at project creation time and can just be enabled (making it public again) afterwards. This flag can only be used on projects. Default is enabled.
- **sourceaccess** flag can be used to hide the sources, but still show the existence of a project or package. This also includes debug packages in case the distribution is supporting this correctly. This flag can only be used at package creation time. There is no code yet which checks for possible references to this package. Default is enabled.
- **downloadbinary** permission still exists like before. However, unlike "access" and "sourceaccess" this is not a security feature. It is just a convenience feature, which makes it impossible to get the binaries via the API directly. But it is still possible to get the binaries via build time in any case. Default is enabled.

22.1.2 Project Build Configuration

A project is configured in the project `/source/$PROJECT/_config` path. It can be edited in web interface in the **Project Config** tab or via one of the following command lines

```
osc meta prjconf -e $PROJECT
osc co $PROJECT _project
```

This file contains information on how to set up a build environment.

22.1.3 Project Build Macro Configuration

The macro configuration is part of the build configuration in `/source/$PROJECT/_config`. It can be added at the end after a **Macros:** line.

22.1.4 An OBS Package

An OBS Package is a sub-namespace below a project. It contains the specification of a single package build for all specified repositories.

22.2 The OBS Interconnect

The OBS interconnect is a mechanism to connect two OBS instances. All content, including sources and binary build results, will be available in the connecting instance. Unlike other methods the instances will also notify each other about changes.

22.3 Download on Demand Repositories (DoD)

22.3.1 Motivation

In a DoD repository external software repositories can be configured which are used for dependency resolution and where packages will be downloaded at build time. A DoD repository has some main advantages in comparison to binary import projects:

- less disk usage as only really required packages will be downloaded
- automatic package updates when new upstream releases are available
- simple to configure in project meta with no for shell access to repo servers

In download repotypes where package checksums can be verified (e.g. suse tags, rpmmd and deb), we recommend that you use a mirror server URL in `<download>` in order to reduce traffic on the master server and configure a `<master>` with an **https** url and a **ssl fingerprint** in order to avoid man in the middle attacks by peer verification.

22.3.2 XML Document Hierarchy

```
<project>
  <repository>
    <download>
      <master/> (optional)
      <pubkey/> (optional)
    </download>
  </repository>
</project>
```

22.3.3 The Daemon

The `bs_dodup` daemon periodically checks for new metadata in remote repositories. This daemon can be enabled for startup with the command

```
systemctl enable obsdodup.service
```

and can be started with

```
systemctl start obsdodup.service
```

22.3.4 The download Element

mandatory attributes:

- arch
- url
- repotype

22.3.5 The master Subelement

The **<master>** tag as shown in the `rpmmd` example below is optional but strongly recommended for security reasons.

Verification is supported in the following repotypes

- `susetags`
- `rpmmd`
- `deb`

This option could be defined by any valid URL (HTTP and HTTPS) to the origin of the repository but it is strongly recommended to use **https** with a **ssl_fingerprint** to bs_dodup possibility to verify its peer in order to avoid man-in-the-middle attacks. The download URL can be a mirror as we validate package checksums found in repo data.

You can easily query the SSL fingerprint of a remote server with the following command:

```
openssl s_client -connect <host>:<port> < /dev/null 2>/dev/null | openssl x509 -  
fingerprint -noout
```

22.3.6 The pubkey Subelement

The `pubkey` element contains one or more GPG public keys in order to verify repository information but not packages. For an example, look at the repotype "deb" documentation below.

22.3.7 Repository Types

22.3.7.1 YAST Sources (susetags)

Example:

```
<project name="My::SuSE::CD">  
  
  [...]  
  
  <repository name="standard">  
    <download arch="x86_64" url="http://mirror.example.org/path/to/iso"  
    repotype="susetags" />  
  </repository>  
</project>
```

```

    <download arch="i586" url="http://mirror.example.org/path/to/iso"
reptype="susetags" />
    <arch>x86_64</arch>
    <arch>i586</arch>
</repository>
</project>

```

22.3.7.2 RPM Sources (rpmmd)

Example:

```

<project name="Fedora:Rawhide">

    [...]

    <repository name="standard">
        <download arch="x86_64" url="http://mirror.example.org/fedora/rawhide/x86_64/os"
reptype="rpmmd">
            <master url="https://master.example.org/whereever/fedora/rawhide/x86_64/os"
sslfingerprint="sha256:0a64..0303"/>
        </download>
        <download arch="i586" url="http://mirror.example.org/fedora/rawhide/i386/os"
reptype="rpmmd">
            <master url="https://master.example.org/whereever/fedora/rawhide/i386/os"
sslfingerprint="sha256:0a64..0303"/>
        </download>
        <arch>x86_64</arch>
        <arch>i586</arch>
    </repository>
</project>

```

22.3.7.3 Apt Repository (deb)

Apt supports two repository types, flat repositories and distribution repositories.

The download url syntax for them is:

- <baseurl>/<distribution>/<components>
- <flat_url>/.[/<components>]

You can specify multiple components separated by a comma.

An empty components string is parsed as "main".

Example:

```
<project name="Debian:8">

  [...]

  <repository name="ga">
    <download arch="x86_64" url="http://ftp.de.debian.org/debian/jessie/main"
    repotype="deb">
      <pubkey>
      -----BEGIN PGP PUBLIC KEY BLOCK-----
      Version: GnuPG v1.4.12 (GNU/Linux)

      [...]

      </pubkey>
    </download>
    <download arch="i586" url="http://ftp.de.debian.org/debian/jessie/main"
    repotype="deb">
      <pubkey>
      -----BEGIN PGP PUBLIC KEY BLOCK-----
      Version: GnuPG v1.4.12 (GNU/Linux)

      [...]

      </pubkey>
    </download>
    <arch>x86_64</arch>
    <arch>i586</arch>
  </repository>
</project>
```

22.3.7.4 Arch Repository (arch)

Be aware that there is currently no way to verify the origin of repository for Arch.

Example:

```
<project name="Arch:Core">

  [...]

  <repository name="standard">
    <download arch="x86_64" url="http://ftp5.gwdg.de/pub/linux/archlinux/core/os/x86_64"
    repotype="arch"/>
  </repository>
</project>
```

```

    <download arch="i586" url="http://ftp5.gwdg.de/pub/linux/archlinux/core/os/i686"
    repotype="arch"/>
    <arch>x86_64</arch>
    <arch>i586</arch>
  </repository>
</project>

```

22.3.7.5 Mandriva Repository (mdk)

Example:

```

<project name="Mageia:5">

  [...]

  <repository name="standard">
    <download arch="x86_64" url="http://mirror.example.org/Mageia/distrib/5/x86_64/media/
    core/release" repotype="mdk"/>
    <download arch="i586" url="http://mirror.example.org/mirrors/Mageia/distrib/5/i586/
    media/core/release" repotype="mdk"/>
    <arch>x86_64</arch>
    <arch>i586</arch>
  </repository>
</project>

```

22.4 Integrating External Source Repositories

22.4.1 Motivation

This chapter makes some recommendations how upstream resources can be integrated into the build process. SCM stands for source control management. git, subversion or CVS are concrete implementations of an SCM. The OBS itself comes also with an own SCM, but this is only intended to manage the files needed for packaging. However, you can add references to external SCM systems. The source service system will mirror the sources and provide it to the build systems.

OBS makes sure that you can access the sources of all builds also in the future, even when the upstream server delivers different or no content at all anymore. Using external SCM references has the following advantages:

- It is documented where a source comes from and how to create the archive.
- Working on the upstream sources can be done directly in local checkouts and changes can be tested via local builds before pushing to the SCM server.
- The sources can be stored incrementally and need less storage on the server.

22.4.2 Creating an Reference to an External SCM

External references are defined in `_service` files. The file can look like this:

```
<services>
  <service name="obs_scm">
    <param name="url">git://...</param>
    <param name="scm">git</param>
  </service>
  <service name="tar" mode="buildtime"/>
  <service name="recompress" mode="buildtime">
    <param name="file">*.tar</param>
    <param name="compression">xz</param>
  </service>
  <service name="set_version" mode="buildtime" />
</services>
```

The services do the following:

- `obs_scm`: mirrors the source. It stores it as a cpio archive, but for the build process this looks like a directory. It also stores additional information from the metadata to a file with `obsinfo` suffix.
- `tar`: creates a tar file from the directory
- `recompress`: applies a compression on the tar file
- `set_version`: reads the version from the `obsinfo` file and adapts the build descriptions to it.

Note that only the first service (`obs_scm`) runs on the OBS server. The other services run during the build process. They can also be replaced by any user by providing alternative implementations of them, or by writing their own service from scratch.

22.4.3 Working with Local Checkouts

Using **osc build** in any package with such a definition will do the same process locally. The only difference is that you get a local subdirectory with the SCM content. You can go inside and work as you are used to. Any changes inside will be used for your next local build, whether they were pushed to the upstream server or not. However, you need to push it upstream when you let the OBS server re-fetch the changes from upstream. The only way out would be to set the obs_scm service to mode disabled and upload your local archive.

22.4.4 Managing Build Recipes in a SCM

The obs_scm service allows you to export files next to the archive. You can specify one or more files using the extract parameter. Use it for your build recipe files.

23 Build Process

The build process creates new binaries from sources, binaries, and config. This process may run on the OBS server side or on a local workstation. Each package build is created in a fresh environment. This is done to ensure that the environment is reproducible.

23.1 Phases of a Build Process

All sources and binaries which are hosted inside Open Build Service are organized in projects. Projects host sources inside of OBS packages. The sources are built according to the repository configuration inside of the project.

23.1.1 Preinstall Phase

This phase depends on the type of the buildroot (building environment). OBS supports multiple types of build environments, for example:

- chroot
- Xen
- KVM
- Qemu

In the preinstall phase, the OBS Worker creates a small base system from the packages declared to be preinstalled (file system, coreutils, binutils, rpm/debutils, etc.). The tools installed in this phase must only provide the minimum functionality necessary to allow installing further packages. In addition it copies all necessary build requirements and the source into the base system.

23.1.2 Install Phase

Depending on the chosen build environment, the worker may start a virtual machine, an emulator or just enter the build root. If this was successful, the install phase reinstalls all base packages from above and additionally all packages you have defined in your build recipe plus dependencies. After this phase the environment is ready to process the build recipe.

23.1.3 Package Build

Depending on the type of package, the build environment executes different build commands, for example:

- RPM-based distributions: `rpmbuild`
- Debian-based distributions: `dpkg-buildpackage`
- Arch Linux: `pacman`
- Kiwi image: `kiwi`.

How the build continues depends on the quality and the type of your build recipe. In most cases, the source code will be compiled now and then be packed into the chosen package format.

To improve package quality, on RPM-based distributions there are additional checks provided via packages. A common toolchain for handling checks is for example `rpmlint`.

23.1.4 After the Build

The generated packages are extracted from the build environment and transferred back to the server by the worker. The build result might be postprocessed by followup build jobs. Afterwards the resulting files may get signed.

23.2 Identify a build

OBS is usually tagging each build with an identifier. This can be used to find the building OBS instance, the project, repository and exact source for a binary. This information is stored in some variable called `DISTURL` and is specified as `obs://$OBS_INSTANCE/$PROJECT/$REPOSITORY/$SOURCE_REVISION-$PACKAGE(:$FLAVOR)`. Note that the final segment, `:$FLAVOR`, is optional and exists only for packages built using the multibuild feature. The source specified via the `DISTURL` can be accessed by pasting the URL into the search interface of the OBS web interface. Or use the command line tool to check it out:

```
# osc checkout $DISTURL
```

You need to go to the right OBS instance as this is not handled automatically yet.

23.2.1 Read DISTURL from an RPM

RPM binaries contain the DISTURL as tag. It can be read from the rpm database for installed RPMs and also from the rpm binaries itself.

```
# rpm -q --qf '%{DISTRL}\n' $rpm
```

23.2.2 Read DISTURL from a container

Containers store the DISTURL as label. You will see only the DISTURL from the highest layer via

```
# docker inspect --format '{{.Config.Labels}}' $image_id
```

The disturl is always set via the key 'org.openbuildservice.disturl'.

24 Build Containers

Containers are workloads which embed all necessary files to make the workload independent of the running host OS. This includes (but is not limited to) libraries, executables and shared resource files.

24.1 Supported Container Formats

A container that is providing its own kernel is commonly called a virtual machine and will be referred to as such in this book. The Open Build Service (OBS) supports container builds either by supporting the native build format or as side product of a different format. This ranges from very simple chroot containers over server (for example, Docker) or desktop formats (for example, AppImages, Snaps or Flatpaks) up to full VM builds (such as for OpenStack, KVM, or as a Live CD via KIWI).

SimpleImage

SimpleImage is a special format which uses the rpm spec file syntax and just packages the resulting install root as tar ball or squashfs image. The format is just using the BuildRequires tags from a file called `simpleimage`, it supports also rpm macro handling to allow for exceptions depending on the build environment.

Docker

Docker images can be built either via the KIWI tool or from Dockerfile build descriptions.

AppImage

The desktop-oriented AppImage format is currently only created as a side effect of an RPM build. Open Build Service (OBS) supports signing and publishing the .AppImage files, the rest is handled via wrapper packages which converts an RPM (or DEB package) into an AppImage file. Own build rules can be provided via a `Recipe` file, fallback code will be used if no `Recipe` file is available.

Snap

The Snap format is supported natively. However, external resources are only supported via source services and therefore not all build types are supported. Snapcraft only works with Ubuntu-based base systems. (Code to support RPM-based distributions exists as well but has not been merged upstream yet.)


Flatpak

Flatpak packages can be built in the Open Build Service, see [Section 2.8, “Flatpak”](#) for further details.



Livebuild

Livebuild is the Debian livebuild support for ISO images.


Mkosi



Mkosi (<https://github.com/systemd/mkosi/>)  allows building images for rpm, arch, deb, and gentoo based distributions, see [Section 2.9, “mkosi”](#) for further details.

24.2 Container Registry


Container Registries are repositories that container images are published and can be automatically pulled from using tools like [podman](#) (<https://podman.io/>)  or [docker](#) (<https://www.docker.com/>) .

The Open Build Service will automatically publish container images in a OCI-compatible registry, with the URLs to the images constructed as follows: \$BASE/\$PROJECT/\$REPOSITORY/\$IMAGE_NAME:\$TAG with the following components:

- \$BASE: URL/IP under which the Open Build Service instance is reachable
- \$PROJECT: The name of the project where the image is build, with all colons replaced with forward slashes.
- \$REPOSITORY: The name of the repository where the containers are published.
- \$IMAGE_NAME: The name of this container image. It defaults to the name of the package from which the container is build. Alternatively, a different image name can be specified in the build recipe, e.g. via the `containerconfig` element in a KIWI file (see [the kiwi documentation](https://osinside.github.io/kiwi/building_images/build_container_image.html?highlight=containerconfig) (https://osinside.github.io/kiwi/building_images/build_container_image.html?highlight=containerconfig)  for further details).
- \$TAG: The image tag. This defaults to `latest` with alternatives being provided in a similar fashion as the image's name.

The [cooverview](https://github.com/openSUSE/cooverview) (<https://github.com/openSUSE/cooverview>)  project provides a simple user-facing webpage to search for containers published by the Open Build Service and can be used to conveniently obtain the correct registry URLs. It is used to power registry.opensuse.org (<https://registry.opensuse.org/>) .

24.3 Container Image Signatures

The Open Build Service automatically signs every package that has been build and publishes the cryptographic signature alongside with it. Container images are no exception to this and the detached signatures can be used by [podman](https://podman.io/) (<https://podman.io/>)  to verify every image that is pulled from the registry.

Podman has to be configured first as outlined in the following steps.

- Create a `yaml` file under `/etc/containers/registries.d/` with an appropriate name for your instance and the following contents:

EXAMPLE 24.1: `REGISTRY.YAML`

```
---
docker:
  REGISTRY_URL:
    sigstore: REGISTRY_URL/sigstore
```

Replace `REGISTRY_URL` with the appropriate URL to your instance of Open Build Service (for example, `registry.opensuse.org`).

- Add the following object into the key `transports` in the file `/etc/containers/policy.json`:

EXAMPLE 24.2: `POLICY.JSON`

```
"docker": {
  "REGISTRY_URL": [
    {
      "type": "signedBy",
      "keyType": "GPGKeys",
      "keyPath": "PATH_TO_PUBLIC_KEY"
    }
  ]
}
```

The complete `/etc/containers/policy.json` can then look like this:

EXAMPLE 24.3: `POLICY.JSON`

```
{
  "default": [
    {
      "type": "insecureAcceptAnything"
    }
  ]
}
```

```

],
"transports": {
  "docker-daemon": {
    "": [
      {
        "type": "insecureAcceptAnything"
      }
    ]
  },
  "docker": {
    "REGISTRY_URL": [
      {
        "type": "signedBy",
        "keyType": "GPGKeys",
        "keyPath": "PATH_TO_PUBLIC_KEY"
      }
    ]
  }
}
}

```

- Save the public key of the project where your image is build under PATH_TO_PUBLIC_KEY (you can choose any location to which you have read access, only ensure that you specify it in /etc/containers/policy.json as well).

Podman will from now on automatically fetch the published signatures from the backend and verify them before storing the images locally.

25 Source Management

25.1 Find Package Sources

OBS is adding information to each created package about the origin of the sources. This information is stored in the **DISTURL** tag of an rpm, which can be displayed as follows:

```
rpm -q --queryformat '%{DISTURL}\n' glibc
rpm -q --queryformat '%{DISTURL}\n' -p glibc-2.1.0-1.i586.rpm
```

The `disturl` can look like this: **obs://build.opensuse.org/openSUSE:Factory/standard/80d21fdd2299302358246d757b4d8c4f-glibc** It always starts with `obs://`. The second part is the name of the build instance, which usually also hosts the Web UI. Next comes the project name and the repository name where the binary got built. Last part is the source md5 sum and the package name.

The `disturl` can also be entered in the search field of the web interface of the build service.

rpm packages managed via the `scmsync` mechanic may have also the `VCS` tag. It provides the git repository URL when the project is build using the

```
BuildFlags: setvcs
```

flag in the build config.

25.2 Generating SLSA Provenance Data

OBS 2.11 can produce and publish additional SLSA provenance attestation files. This files are currently following the v0.2 Alpha spec, which is suspect to change. We will change the code to follow the specification, so the files might change in an incompatible way until a stable version has been released.

This can be enabled via the `BSConfig.pm` file only. The reason behind is that the functionality is expensive in regards of disk space. All old binaries used for build are kept. You may want to enable it nevertheless by setting the `slsaprovenance` variable in `BSConfig.pm` with a list of projects to enable it.

25.3 Generating SBOM (Software Bill Of Material) Data

OBS 2.11 can produce and publish additional SPDX data for certain build types. This is controlled via the project configuration. For details, refer to [Section 4.2, “Configuration File Syntax”](#) for sbom:FORMAT (under BuildFlags).

26 SCM Bridge

26.1 SCM Bridge

26.1.1 Introduction

The SCM bridge allows the sources of a single package or an entire project to be stored and maintained in any trusted SCM repositories. However, git is currently the only supported SCM system.

This allows to manage all packaging-relevant source changes in an external SCM repository using the native tooling of the SCM (e.g. git). It is recommended to have this SCM server on the same trust level as the OBS instance as it becomes the authoritative source. However, any SCM server reachable via network could be used.

osc can still be used to checkout sources. This will create an SCM repository, so any modification needs to be done using the native SCM tooling (e.g. git). osc can still be used for local building and checking server side build results.

Be aware that any source modification mechanics, submit requests or link behaviour may work differently or not at all. For example you can not use automatic package source merging via `_link` files anymore. This means also that workflows inside of OBS are not working anymore or are limited. A submit or release request would still work with a `scmsync` (source control management system sync) source, but the user would always need to manually specify the target. Functionality for when an entire project is managed via `scmsync` is even more limited.

Furthermore it is recommended to configure a global notification hook in the SCM server to allow OBS to follow the sources automatically.

26.1.2 Setup a package using the scm bridge

The setup is purely done in package meta by defining the SCM URL inside of the `scmsync` tag:

```
<scmsync>https://gitlab.com/some/repository</scmsync>
```

The repository is cloned including all subdirectories and submodules. However the build descriptions need to be placed in the top-level directory.

Large binary files may be stored using git LFS or by referencing external build assets, using asset management. These assets will get downloaded and verified as well. The advantage is that this information can also be used to compile Software Bills of Materials (SBOMs). Have a look in the [pbuild documentation \(http://opensuse.github.io/obs-build/pbuild.html#_remote_assets\)](http://opensuse.github.io/obs-build/pbuild.html#_remote_assets) for further details on this.

In another typical scenario, a package maintainer owns a git repository where a build description and distribution specific files are stored. The sources from the upstream project may be added via a git submodule. However, for many build types like rpm builds it is recommended to build a tar ball at buildtime for the source rpm. This can be achieved with buildtime source services.

26.1.3 Setup an entire project using the SCM bridge

An entire project can be setup by defining the `scmsync` tag in project meta data. Every top level subdirectory of the scm repository is considered as a package.

Large projects may use git submodules for each package. This avoids the need to clone the entire project to modify a single package.

The build configuration part can get stored as `'_config'` file in the top level directory.

It is possible to limit the actual used package sources by specifying an "onlybuild" cgi parameter as part of the `scmsync` url. This parameter can be used multiple times. This is useful, for example, when trying to do a test build of just a few packages without the need to modify the source. In that way an open merge request may be tested before merging. Example URL:

```
<scmsync>https://gitlab.com/some/repository?onlybuild=glibc&onlybuild=kernel</scmsync>
```

26.1.4 Implementation and Limitations

The sources are currently cloned for the OBS source server to allow OBS to process them. This is to be considered an internal implementation detail and may change in future.

- Git submodules: are cloned by default
- Git VCS history: is not included for server side builds by default to save storage space. It can be included by adding a `keepmeta=1` CGI parameter to the URL. This is not needed for local building when using `osc`.
- Git LFS files: are included by default. They can be excluded by adding a `lfs=0` CGI parameter to the URL.

- Using just a subdirectory of the git repository can be done by adding a `subdir = DIRECTORY` CGI parameter to the URL.
- Architecture specific remote assets can be selected by adding an `arch = ARCH` CGI parameter to the URL.

As an example, for the subdirectory case the URL would look like this:

```
<scmsync>https://gitlab.com/some/repository?subdir=MY_SUBDIRECTORY</scmsync>
```

26.1.4.1 Using a specific revision, tag or branch

The URL can define a revision, tag or branch via an URL fragment. This means the URL can get extended by a hash character and the revision, tag or branch.

```
<scmsync>https://gitlab.com/some/repository#MY_REVISION</scmsync>
```

This allows to set up multiple projects building for different branches. It is possible to use branches for implementing CI workflows. For example a submission test building just a subset, a clean build and a final reviewed build.

26.1.4.2 Converting to a project git

A project git repository is mostly the same as any project for the pbuild tool. The command

```
# osc create-pbuild-config
```

creates the `_config` and `_pbuild` files for a given repository and architecture. The `_config` file is used by OBS as well as by the build configuration (`prjconf`). The additional `_pbuild` file is analogous to the OBS project meta, but is not yet honored by OBS. Still, it can be included in the git tree for pbuild users.

26.1.4.3 Forking a scmsync package

The OBS server allows to create a cloned project with another package using a specified SCM repository. This can be used by tooling to create forked test builds for merge requests. The api POST route for this is

```
/source/PROJECT/PACKAGE?cmd=fork&scmsync=NEW_SCM_URL
```


26.1.5 SCM Source Updates

The OBS instance needs to get notified on any change in the SCM server. There are two ways to achieve this. One way is via single configurations for each git repository as documented in the [Section 7.6, “Trigger a service run via a webhook”](#) documentation. An alternative way is to configure it globally. This requires admin permissions on the git hosting side and another bridge implementation. The advantage is that any used repository will be synced automatically just by referencing it via the scm sync meta tag.

The update notification has been only implemented for gitea atm, you can find details in the [obs-gitea-bridge \(https://github.com/openSUSE/obs-gitea-bridge\)](https://github.com/openSUSE/obs-gitea-bridge) [↗](#) documentation.

27 Supported Formats

Open Build Service is by design format agnostic, but it needs format specific support to be able to parse build descriptions and running the build. This chapter is focusing on describing Open Build Service specifics of a format. Either limitations or extensions of Open Build Service builds.

27.1 Spec File Specials (RPM)

To create an RPM package, you need a spec file.

- A file with the extension `.changes` can be used to document the package history, but it is not required.
- OBS-specific RPM macros which are set are: `_%_project` and `_%_repository`
- `#!ForceMultiVersion` can be used to avoid resetting the build counter reset on version update. This is handled automatically in most cases. Explicitly adding this instruction is only needed when it is not possible to detect whether there are subpackages defining their own version. This can be the case when the subpackages are created dynamically via external rpm macros.
- `#!BcntSyncTag: STRING` defines the build counter sync tag in a spec file. It is used when the `bcntsynctag` element tag is not defined in package meta. Also it can depend on architecture, repository or flavor for example.
- `#!BuildConflicts: STRING` defines a conflicting package or dependency at build time only.
- `#!BuildIgnore: STRING` ignores a dependency at build time.
- `#!BuildRequires: STRING` Requires a build dependency. Usually the rpm native `BuildRequires` is the better way to define this. Use this tag only when the dependency should only exist when using the build script or OBS.
- `#!needsrootforbuild` The build will be executed by root user. Please note that this must be permitted either by the OBS admin (via the `BSConfig.pm` configuration) or enabled in via build configuration rule. Also osc requires an additional parameter for local build (`--userootforbuild`).

- `#!needsslcertforbuild` This will prepare an SSL certification based on the project key. It is used to create secure boot enabled packages usually. Please note that this will fail, if the project still has an old DSA key.
- `#!needsappxsslcertforbuild` Creates an appx compatible ssl certificate based on the project key. Please note that this will fail, if the project still has an old DSA key.
- `#!needsbinariesforbuild` All package files are usually removed before starting the build (after finishing to setup the build environment). This marker keeps them also during build.

27.2 OBS Extensions for (KIWI) Appliance Builds

KIWI appliance builds create images which can be used for direct consumption. Note: builds in non-VM environments do fail often due to the bootloader setup. Use **`osc build --vm-type=kvm`** for local building. OBS is evaluating kiwi builds at least two times. One time for the build host dependencies and another time for the target distribution used inside of the image. For container builds a third evaluation happens to find the base container.

KIWI builds inside of OBS need to fulfill following requirements:

- `config.xml` files need to be renamed to a filename with `.kiwi` suffix.
- repositories which are used must use either a URL which starts with the OBS download prefix or they must be written in `obs://PROJECT/REPOSITORY` syntax.

OBS extends kiwi functionality with following options. These covers cases which would need explicit command line commands, so they are covered via the tags to have them in a reproducible way.

- A repository defined as `obsrepositories:/` will used the expanded list of repositories as defined in the project meta. This is useful when moving kiwi image descriptions between projects, for example for development and release builds.
- Defining build counter sync tag **`OBS-BcntSyncTag: STRING`**
- An exclusive architecture list to build for can be defined with an xml comment **`OBS-ExclusiveArch: ARCH_LIST`**
- An excluded architecture list to skip builds can be defined with an xml comment **`OBS-ExcludeArch: ARCH_LIST`**

- A list of profiles to build can be defined via an xml comment **OBS-Profiles: PROFILE_LIST**
This can also be handled via `_multibuild` flavor lists by setting a fixed string of **OBS-Profiles: @BUILD_FLAVOR@**
- Packages get picked usually in a fixed order from the most important repository in OBS. This happens independently of the version number of the packages. Set the xml comment **OBS-UnorderedRepos** to disable the repository order handling and to have a more similar behavior as with plain package manager builds. Note: this can result into switching repositories between builds and is therefore less reproducible.
- Container builds can specify a specific repository which shall be used for searching the base container. Use the XML comment **OBS-Imagerepo:\$REPOSITORY_URL** for this. The repository URL may be in `obs://$PROJECT/$REPOSITORY` style.
- Container builds can specify additional tags via **OBS-AddTag:\$TAG** (KIWI limits it to one tag) in `obs://$PROJECT/$REPOSITORY` style.
- A predefined milestone **OBS-Milestone: milestone**
This defines a milestone name (for example, `Beta1`) which will be applied during release operations. The candidate will receive its final tag then.

27.3 OBS Extensions for Dockerfile based builds

OBS needs to parse RUN commands to detect build dependencies (repositories and packages). Currently calls from `zypper`, `apt-get`, `yum` and `dnf` are supported. Additional downloads need to be covered by source services. This ensures reproducible builds and a safe build environment without network access.

OBS extends Docker functionality with the below tags. Instead of these options, you would otherwise need explicit command-line commands. However, to make it possible to create reproducible builds, use the OBS tags.

- A **#!UseOBSRepositories** tag will use the expanded list of repositories as defined via path elements in the project meta. This is useful when moving descriptions between projects, for example for development and release builds.
- The tag **#!UnorderedRepos** will disable the repository prioritization for build dependencies. This behavior is similar to plain Docker tooling. It introduces the risk that with each build another repository is prioritized or that dependency problems of newer package versions are hidden.
- **#!BcntSyncTag: TAG** to define the build counter sync tag.
- **#!BuildTag: TAG** to define one or multiple tags to be used for the container.
- **#!BuildName: NAME** Report back a defined NAME. Otherwise it gets derived from the first tag, replacing all / and : with - characters.
- **#!BuildVersion: VERSION** Report back a defined VERSION. By default it is set to zero. The version is used by OBS for tracking. This needs to be set together with BuildName, otherwise it will be ignored.
- **#!ExclusiveArch: ARCH_LIST** An exclusive architecture list to build for. The architecture list is separated by spaces.
- **#!ExcludeArch: ARCH_LIST** An excluded architecture list to skip builds. The architecture list is separated by spaces.
- **#!ArchExclusiveLine: ARCH_LIST** The next line will only be considered by the scheduler the listed architectures. The line will still get executed on all architectures during the build.
- **#!ArchExcludedLine: ARCH_LIST** The next line will not be considered by the scheduler on the listed architectures. The line will still get executed on all architectures during the build.
- **#!Milestone: MILESTONE** This defines a milestone name (for example, Beta1) which will be applied during release operations. The candidate will receive its final tag then.
- **#!NoSquash** disables the squashing of all layers created during the build to a single layer. Without this, every RUN line is an additional layer.

The used filename of the build results are only important inside of OBS. The publishing happens via the built-in registry interface which is providing all defined tags for an image. The filename itself may be used on aggregates or other binary filters in OBS. It is either derived by the first defined tag or by the BuildName and BuildVersion tag if both are defined.

28 Request And Review System

The OBS comes with a generic request system where one party can ask another to complete a certain action. This can be, for example, taking source changes, granting maintainer rights or deleting a package. Requests are also used deal with more complex workflows.

A request is an object in the database. It can be accessed via the `/request` API route. osc and the web interface can show and process these requests. There are also interfaces to show the requests which should be handled for a certain user.

28.1 What a request looks like

A request is an object in the database. It can be accessed via the `/request` API route. Main parts of the request are

- **state:** The state tells if the request still needs to processed or has been handled already and how.
- **actions:** these are the changes which will be applied when accepting the request.
- **reviewer:** reviewer can be added automatically at request creation time or manually by any involved party. Usually all of them should approve the request before it will be accepted. However, the target can ignore that and accept anyway optionally.
- **description:** an explanation of why the actions should be done.
- **history:** a history about state changes of the request.
- **accept_at:** the request will get accepted automatically after the given time. Such a request can only be created when having write permissions in the target. Automatic cleanup requests created by Admin user are using this.

Requests can only be accepted or rejected in their entirety. Therefore, it can make sense to have multiple actions in one request if changes should be applied in one transaction. For example, submitting a new package and removing an old instance: Do either both or nothing. This implies that the person accepting the request must have write access in all targets or they will not be allowed to accept the request.

28.1.1 Action Types

Actions always specify some target. This can be either a project or a package. Further information depend on the action type. The following gives an overview, for details, see the XML schema for requests.

28.1.1.1 `submit`

A submit action will transfer sources from one package to another package. Usually a submit request will refer to a specific revision in the source, but it does not have to. If no revision is specified, then the current revision at the time of acceptance will be used. This should be avoided when relying on complex reviews during the request process. Hence, it is recommended to identify a specific version in your submitrequest (`osc submitrequest -r 42 ...`).

The submit action can support options to update the source or even to remove the source. Tools like osc are applying the cleanup rule by default when submitting from a default user home branch project.

28.1.1.2 `release`

Is used to release a finished build. Sources and binaries are copied without a rebuild (The target project should have build disabled). A release target needs to be defined with `trigger="manual"`.

28.1.1.3 `delete`

A delete action can request removal of a project or package instance.

28.1.1.4 `add_role`

An add_role requests a specific role for a given user or group to the target. For example, one could use this to ask for maintainer rights, or to become a default reviewer.

28.1.1.5 `set_bugowner`

`set_bugowner` is similar to `add_role`, but removes all other bugowner roles in the target. This happens to have a unique identifier to be used when assigning bug reports in external tools like Bugzilla.

28.1.1.6 `change_devel`

can be used to update the devel package information in the target.

28.1.1.7 `maintenance_incident`

Official request to open a maintenance incident for official support products. These requests are created by developers who want to start an official maintenance process. Details are described in the maintenance chapter. A new maintenance incident project is created and package sources get copied there when accepting it. All sources of all actions in one request will be merged into the same maintenance incident project.

28.1.1.8 `maintenance_release`

Is used to release a maintenance update. Sources and binaries are copied without a rebuild. Open Build Service also creates a unique update identifier. Details can be found in the maintenance chapter.

28.1.1.9 `group`

Deprecated. Was never in a released OBS version. It is not allowed to be used anymore.

28.1.2 Request states

- **new:** The default value for newly created requests. Everybody involved in the specified targets can see the request and accept or decline it.
- **accepted:** The request has been accepted and the changes applied. history files have a reference to this request.

- **declined:** The request has been reviewed and not (yet) been accepted by the target. This is often used to ask for some more information from the submitter, since declined requests remain active, returning to the submitter's active request queue (that is, the submitter will need to take action now).
- **revoked:** The submitter has taken back their request. The request is considered to be inactive now.
- **superseded:** This request is obsolete due to a new request. The request is considered to be inactive now. The superseding request is linked in this request.
- **review:** There are still open reviews inside of the request. Nobody has declined it yet. The request is not yet visible to the target by default. The state will change automatically to new when all reviewers accept.

28.1.3 Reviewers

Reviews can be done by users, groups, projects or packages. Review by project or package means that any maintainer of them is asked for reviews. This is handy to avoid the need to figure who actually is a maintainer of a certain package. Also, new maintainers of a package will see requests in case the old maintainer did not handle them.

28.1.3.1 Manually added reviews

Reviewers can be added manually by anyone involved in a request. This can be used to hand over a review. In that situation the new reviewer needs to be added and the original reviewer's own review needs to be accepted. The request becomes declined when any of the reviewers are declining the request.

28.1.3.2 Automatically added reviews

Project and package objects can have users or groups with a reviewer role. They are added automatically to a request as reviewer when a request is created which has them as target. In case the project and package do specify reviewers, all of them are added to the request.

28.1.4 Request creation

The API is doing a number of checks at request creation time. In case a target is not specified it tries to set it according to the linked package. If an entire project is specified as source it expands it to all packages inside. This means it is replacing one action with multiple. When using the `addrevision` parameter it does also add the current revision of the package source to the action. This makes it easy to create new requests with little logic in the client.

28.1.5 Request operations

Requests can be modified only in very limited ways after creation. This is to avoid the nature of the request changing, after reviewers have reviewed it. Valid operations on a request are:

- `diff`: does not modify the request, just shows source modifications wanted by the request
- `changestate`: to change the state of the request, for example to accept it.
- `changereviewstate`: to change the state of a review inside of a request.
- `addreviewer`: add further reviewer to a request

29 Image Templates

Image templates are pre-configured image configurations. The [image templates page \(https://build.opensuse.org/image_templates\)](https://build.opensuse.org/image_templates) provides a list of these templates. Users can clone these templates and further configure them as they like.

29.1 Structure of Image Templates

As mentioned image templates are essentially pre-configured [KIWI \(https://github.com/OSInside/kiwi\)](https://github.com/OSInside/kiwi) image configurations. As any KIWI configuration they usually contain a tarball containing image sources, a config.sh file and the KIWI configuration XML file.

In addition, you can define an icon for your image templates by adding graphical image (for example, PNG, JPG) to your template sources and name it **_icon**. If that file exists, it will be used as icon for your image on the image templates page.



Note

For more information about KIWI images, see [Section 2.5, “KIWI Appliance”](#).

29.2 Adding Image Templates to/Removing Image Templates from the Official Image Template Page

The image templates page lists templates per project. New templates get added by setting the **OBS:ImageTemplates** attribute to a project. Any package container belonging to a project with that attribute will be shown on the template page.

Only admins can add / remove the OBS:ImageTemplates attribute from a project.

29.3 Receiving Image Templates via Interconnect

If your OBS instance is connected to a remote instance via interconnect, OBS will fetch image templates from the remote instance and present it on the image templates page. They appear below the local templates.

For more information about interconnects, see *Book “Administrator Guide”, Chapter 5 “Administration”, Section 5.2 “Managing Build Targets”*.

30 Multiple Build Description File Handling

30.1 Overview

A package source may contain multiple build description files. They can be used depending on the base distribution, the repository name or for different configurations. These mechanics can be also combined.

The right build description file gets picked by filtering. The build will not start when either no file matches or multiple candidates exist. The filtering happens with the following steps:

1. Based on the package build format of the based distributions. RPM-based distributions will use spec files for example.
2. Based on the file name of the file before the suffix. It is not important as long as just one file exists, but it has to match when multiple files exist. The name is defined by the build container name, which is either defined in a `_multibuild` directive file or is the source package name.
3. Specific files can be created to be built for a specific repository. Append the repository name of the build container behind the package name with a `-`. For example `hel-lo-openSUSE_13.2.spec`.

30.2 How Multibuild is Defined

Use the `_multibuild` directive to build the same source in the same repository with different flavors. This handy to define all flavors in one place without the need to maintain packages with local links. This allows also to transfer all sources including a possible changed flavor from one project to another with a standard copy or submit request.

The `_multibuild` file lists all build container names, each of them will be built as usual for each defined repository and each scheduler architecture.

For example, inside the `kernel` source package we can build both `kernel-source` and `kernel-obs-build` packages by listing them inside the file.

Multibuild packages are defined with the `_multibuild` directive file in the package sources.

The `_multibuild` file is an xml file. For example:

```
<multibuild>
```

```
<flavor>kernel-source</flavor>
<flavor>kernel-obs-build</flavor>
</multibuild>
```

Build description files are needed for each of them for each package (for example, kernel-source.spec or kernel-obs-build.dsc) inside of the sources. There will be another build in case there is also a matching file for the source package container name, otherwise it will turn into an “excluded” state. Dockerfile based build descriptions may provide own build descriptions for each flavor via Dockerfile.FLAVOR suffixed files.

31 Maintenance Support

This chapter explains the setup and workflow of a maintenance update in the openSUSE way. However, this should not be limited to openSUSE distribution projects but be usable anywhere (the entire workflow or just parts of it).

The goal of the OBS maintenance process is to publish updates for a frozen project, in this example an entire distribution. These updates need to be approved by a maintenance team and the published result must contain documentation about the changes and be applicable in the easiest way by the users. The result is a package repository with additional information about the solved problems and defined groups of packages to achieve that. Binary delta data can also be generated to reduce the needed download size for the clients.

Technically this results in a frozen software repository containing the original package distribution and an additional update repository with a subset of updated packages that will be preferred by the package manager thus superseding the packages from the original distribution.

31.1 Simple Project Setup

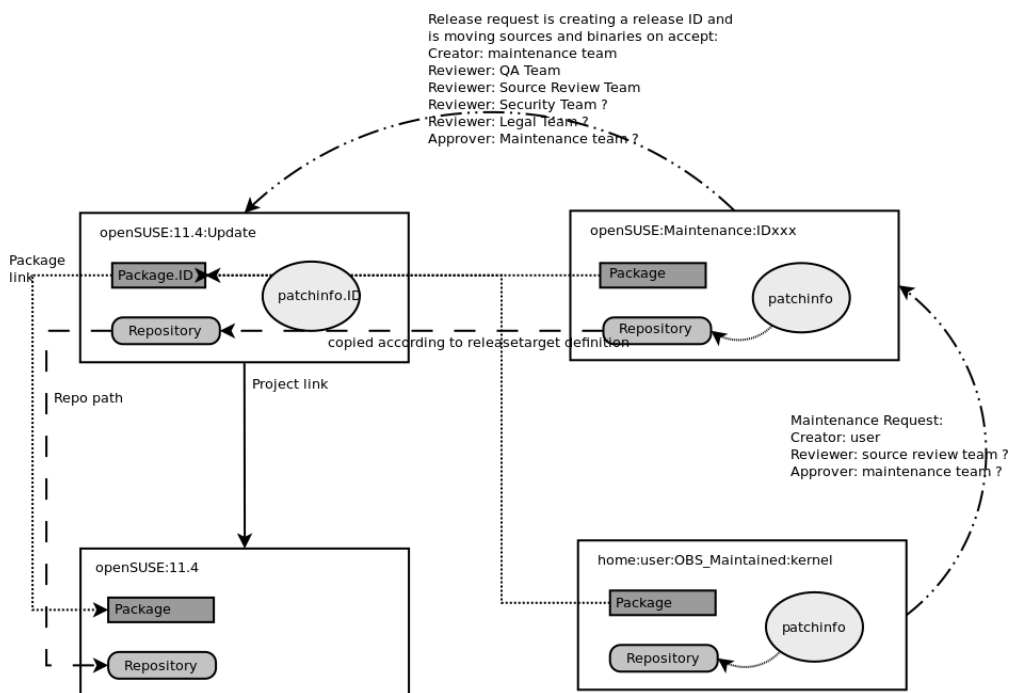


FIGURE 31.1: SIMPLE PROJECT SETUP

This figure gives an overview about the project setup and general workflow for a single package and single maintained distribution. It shows the "openSUSE:11.4" project, which is considered to be frozen and not changing at all anymore. The "openSUSE:11.4:Update" projects hosts all officially released updates. It does not build any binary, just gets it sources and binaries from the maintenance incident project via the release process. The incident project is named "openSUSE:Maintenance:IDxxx" in this example, which is under control of the maintenance team. Official updates get built and reviewed here. QA teams are also testing the binaries from here. However, a user can prepare it in the same way in their project and start the maintenance process via doing a "maintenance" request.

- openSUSE:11.4 is the *GA Project* in this example. It is locked and not changing anymore.
- openSUSE:11.4:Update is the *Update Project* to release official updates for the locked openSUSE:11.4 project. Thus it links to the openSUSE:11.4 project, inheriting all package sources from there.
- openSUSE:Maintenance is the *Maintenance Project* which in this case maintains the openSUSE:11.4:Update project (and optionally others as well).
- openSUSE:Maintenance:IDxxx is a *Incident* project created automatically by accepting a maintenance request.

31.2 Project setup for the Maintenance Process

All workflow related projects must be set up with a proper project meta configuration.

- It is recommended to lock the *GA Project* by the project maintainer by using the `osc lock [PROJECT]` command
- The *Update Project* has to have the `<link project="[PROJECT]"/>` element in the project meta configuration.

It is very useful to define groups of bugowners, maintainers and reviewers and to make use of bots for further quality assurance tasks.

- The *Maintenance Project* has to have the

```
<project name=... kind="maintenance">
```

attribute in the project meta configuration, as well as a

```
<maintenance>
```

element containing one or more

```
<maintains project="[PROJECT]"/>
```

elements. It is very useful to define groups of maintainers and reviewers and to make use of review bots to enforce desired quality properties here.

31.3 Using the Maintenance Process

This describes all required steps by all involved persons from preparing to releasing a maintenance update.

31.3.1 Workflow A: A Maintainer Builds an Entire Update Incident for Submission

A user is usually starting to prepare an update by creating a maintenance branch. This is typically done by creating an own maintenance project. Usually multiple released products are affected, so the server can find out which one are maintained by a given source package name, in this example for glibc including checkout via

```
osc mbranch glibc
osc mbranch --checkout glibc
```

This is equivalent to the API call [`/source?cmd=branch&package=glibc`](#).

It is also possible to branch only one defined version, if it is known that only one version is affected. In this example the openSUSE:12.1 version:

```
osc branch --maintenance openSUSE:12.1 glibc
osc branch -M -c openSUSE:12.1 glibc
```

In a simple setup as described before, create the maintenance branch from the package of the *Update Project* as the *GA Project* can never be changed anymore.

NOTE: both branch commands do support the `--noaccess` parameter, which will create a hidden project. This may be used when a not yet publicly known security issue is get fixed.

Afterwards the user needs to do the needed modifications. Packages will be built and can be tested. Afterwards they may add information about the purpose of this maintenance update via

```
osc patchinfo
```

If the source changes contain references to issue trackers (like Bugzilla, CVE or FATE) these will be added to the `_patchinfo` file.

The server will create a full maintenance channel now, in case the user wants to test this as well. After the user has tested, they have to create a `maintenancerequest` to ask the maintenance team to accept this as an official update incident:

```
osc maintenancerequest
```

On accepting this request all sources of the entire project will get copied to the incident project and be rebuilt. The origin project gets usually removed (based on the request cleanup options).

31.3.2 Workflow B: Submitting a Package Without Branching

You may submit a package source from a project which is not prepared as maintenance project. That works via the `maintenancerequest` mechanism by specifying one or more packages from one project. As a consequence it means also that the first testable build will happen in the maintenance incident project. Also, the maintenance team needs to write the update information on their own.

```
osc maintenancerequest [ SOURCEPROJECT [ SOURCEPACKAGES RELEASEPROJECT ] ]
```

The following example is submitting two packages (`kdelibs4` and `kdebase4`) from the project `KDE:Devel` project as update for `openSUSE:12.1`

```
osc maintenancerequest KDE:Devel kdelibs4 kdebase4 openSUSE:12.1
```



Note: Specifying an Existing Incident

It is also possible to specify an existing incident as target with the `--incident` parameter. The packages will then be merged into the existing incident project.

31.3.3 Workflow C: Process Gets Initiated By the Maintenance Team

The maintenance team may start the process (for example because a security issue was reported and the maintenance team decided that a fix is required). In this case the incident gets created via the Web UI or via the API call:

```
osc createincident [PROJECT]
```

```
osc api /source/PROJECT?cmd=createmaintenanceincident
```

```
osc api /source?cmd=createmaintenanceincident&attribute=OBS:Maintenance.
```

To document the expected work the creation of a patchinfo package is needed. This can be done via

```
osc patchinfo [PROJECT]
```

It is important to add Bugzilla entries inside of the _patchinfo file. As long these are open Bugzilla entries, the bug assignee will see this patchinfo on their "my work" Web UI and osc views, so they know that work is expected from them.

31.3.4 Maintenance Incident Processing

The maintenance incidents are usually managed by a maintenance team. In case the incident got started by a maintainer a maintenance request is targeted towards the defined maintenance project, in our example this is openSUSE:Maintenance. The defined maintainer and reviewers in this project need to decide about this request. In case it gets accepted, the server is creating a subproject with a unique incident ID and copies the sources and build settings to it. The origin project will get removed usually via the cleanup option. This maintenance project is used to build the final packages.

If the maintenance team decides to merge a new maintenance request with an existing incident, they can run the **osc rq setincident \$REQUESTID \$INCIDENT** before accepting the request.

The maintenance team may still modify them or the patchinfo data at this point. An outside maintainer can still submit changes via standard submit request mechanism, but direct write permissions are not granted. When the maintenance people are satisfied with the update, they can create a request to release the sources and binaries to the final openSUSE:11.4:Update project.

```
osc releaserequest
```

The release request needs to specify the source and target for each package. In case just the source package or project is specified the API is completing the request on creation time. It is using this based on the source link target of each package and the release information in the repository definitions.

31.3.5 Incident Gets Released

The release process gets usually started via creating a release request. This sets all affected packages to the locked state, which means that all commands for editing the source or triggering rebuilds are not allowed anymore.

The release request typically needs to be approved by QA and other teams as defined in the Update project. In case something gets declined, the necessary changes need to be submitted to the maintenance project and a new release request has to be created.

A unique release ID will be generated and become part of the updateinfo.xml file in the target project on release event. This ID is different from the incident ID and is usually in the style of "YEAR-COUNTER". The counter is strictly increasing on each release. In case of a re-release of the same incident a release counter will be added.

A different naming scheme can be defined via the OBS:MaintenanceIdTemplate attribute value. The release will move all packages to the update project and extend the target package name with the incident ID. Binaries will be moved as well without modification. The exception is the updateinfo.xml which will be modified by replacing its incident id with the release id.

31.3.6 Incident Gets Reopened and Re-Released

An update should not, but may have an undetected regression. In this case the update needs a re-release. (If another problem shall be fixed a new incident should be created instead.)

If the current update harms the systems, the maintenance team may decide to take it back immediately. It can be done by removing the patchinfo.ID package container in the Update projects. This will create a new update channel without this update.

To re-open a release incident project, it must get unlocked and marked as open again. Unlocking can be done either via revoking a release request or via explicit unlocking the incident. The explicit unlock via osc: **`osc unlock INCIDENT_PROJECT`** is also triggering a rebuild to ensure to

have higher release numbers and adding the "trigger=maintenance" flags to the release target definitions. Afterwards the project can be edited again and also gets listed as running incident again.

31.3.7 Using Custom Update IDs

The used string of update IDs can be defined via the OBS:MaintenanceIdTemplate attribute value of the master maintenance project.

31.4 OBS Internal Mechanisms

OBS is tracking maintenance work and can be used as a database for future and past updates.

31.4.1 Maintenance Incident Workflow

A maintenance incident is started by creating the incident project, either via a developer request or by the maintenance team.

1. Incident project container is created. This is always a sub project to the maintenance project. A unique ID (counter) is used as subproject name. Build is disabled by default project wide.
2. Default content for an incident is added via branch by attribute call:
 - Package sources get added based on given package and attribute name from all existing project instances. The package name is extended by the source project name to allow multiple instances of same package in one project. Source revision links are using the xsrmd5 to avoid that other releases will affect this package instance.
 - Build repositories are added if missing. All repositories from all projects where the package sources gets branched from are used. The build flags in the package instances gets switched on for these.
 - A release target definition is added to the repository configuration via additional releasetarget element. The special release condition "maintenance" gets defined for this.
3. Fixes for the packages need to get submitted now.

4. A patchinfo file need to get added describing the issue.
5. OBS server is building packages according to the sources and update information according to the patchinfo data.
6. one or more release requests get created. It does also set the project to "freeze" state by default, this means no source changes are possible anymore and all running builds get canceled.
7. Usually the request is in review state with defined reviewers from the release project. All reviewers need to review the state in the incident project.
8. Request changes into state "new" when all reviewers accepted the release request.
9. The release happens on accepting the request by the maintainers of the release project.
 - All package sources and binaries get copied into a package container where the package name gets extended by the incident number.
 - A main package gets created or updated, it just contains a link to the current incident package. For example, glibc points to glibc.42. The purpose of this main package is to have a place to refer to the current sources of a package.
 - The release target condition = maintenance gets removed.
 - The updateinfo.xml gets updated with the existing or now created unique updateinfo ID.
 - The server will update the repository based on all existing binaries.
10. OPTIONAL: A maintenance coordinator may remove the release by removing the package instances inside the release project. The source link has to be fixed manually. (We may offer a function for this).
11. OPTIONAL: A maintenance incident can be restarted by
 - Removing the lock flag.
 - Adding again the condition = maintenance attribute to the release target which requires a re-release.

NOTE: The step 1 and 2 may be done via accepting an incident request instead.

31.4.2 Searching for Incidents

The Web UI shows the running and past incidents when going to the maintenance project (openSUSE:Maintenance in our example). It shows the open requests either for creating or release an incident. Also, the open incidents, which are not yet released are visible.

All users need usually just to visit their "my work" screen in Web UI or osc to see requests or patchinfos where actions of them are expected: **`osc my [work]`**

The following items list some common ways to search for maintenance incidents via the api:

- A developer can see the work to be done by them via searching for patchinfos with open Bugzilla entries:

```
/search/package?match=( [kind='patchinfo' and issue/[@state='OPEN' and owner/  
@login='$USER_LOGIN' ] ] )
```

- A maintenance coordinator can see requests for doing a maintenance release via searching for open requests with maintenance_incident action against the maintenance project. They are visible in the Web UI request page of that project or via

```
/search/request?match=(state/@name='new') and action/@type='maintenance_incident'  
and action/target/@project='openSUSE:Maintenance')
```

- A maintenance coordinator can see open incidents via searching for incidents project repositories which have a release target with maintenance trigger. Note: this search result is showing all repositories of a matching project.

```
/search/project?match=(repository/releasetarget/@trigger='maintenance')
```

- A maintenance coordinator can see updates which currently are reviewed (for example by a QA team) via

```
/search/request?match=(state/@name='review') and action/@type='maintenance_release')
```

- A maintenance coordinator can see updates ready to release via searching for open requests with maintenance_release action.

```
/search/request?match=(state/@name='new') and action/@type='maintenance_release')
```


31.5 Setting Up Projects for a Maintenance Cycle

31.5.1 Defining a Maintenance Space

An OBS server is using by default a maintenance space defined via the `OBS:Maintenance` attribute. This must get created on a project where maintenance incident projects should get created below. This project is also defining the default maintenance maintainers and reviewers in its ACL list.

It is possible to have multiple and independent maintenance name spaces, however the maintenance request must be created against this other namespace manually or using a different attribute.

31.5.2 Maintained Project Setups

Maintained projects must be frozen, this means no changes in sources or binaries. All updates will be hosted in the defined update project. This project gets defined via the `OBS:UpdateProject` attribute which must contain a value with the update project name. In addition to this, an attribute to define the active maintenance should also be defined, by default the `OBS:Maintained` attribute. The `osc mbranch` command will take packages from this project as a result.

The Update project should be defined as build disabled as well. Also define a project link to the main project and at least one repository building against the main project.

31.6 Optional Channel Setup

Channels are optional definitions to publish a sub-set of binaries into own repositories. They can be used to maintain a larger amount of packages in a central place, but defining to published binaries with an independent workflow which requires an approval for each binary.

31.6.1 Defining a Channel

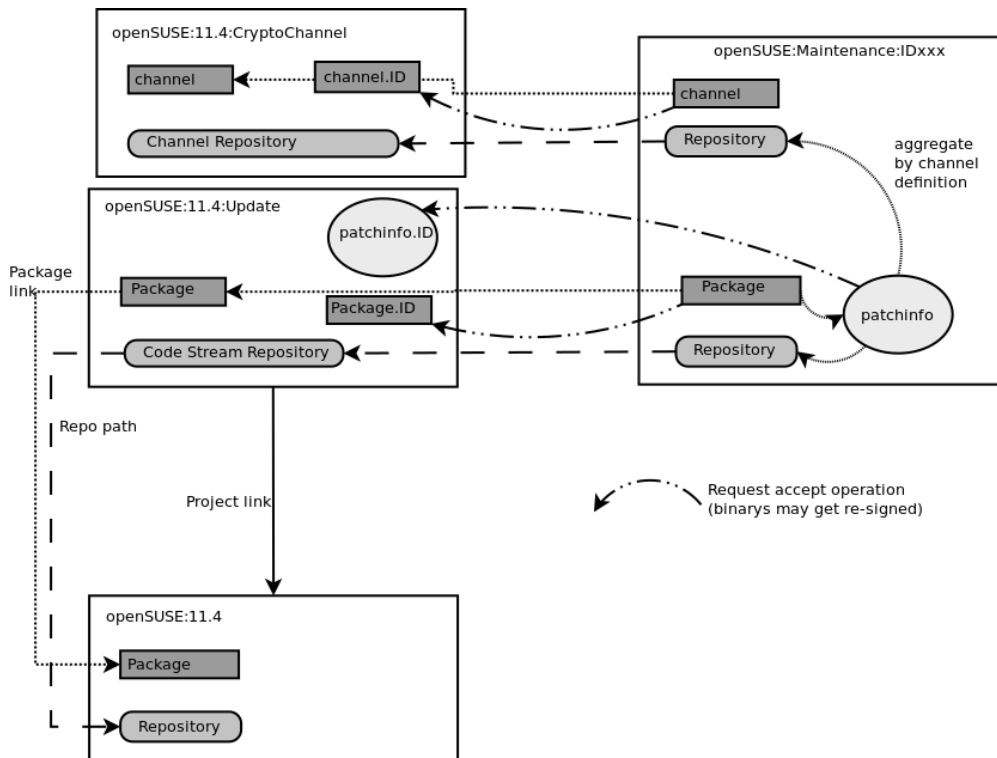
Channels get defined and maintained in an xml file inside of a package source. The file name of these lists must be `_channel`.

The file may contain a list of targets where binaries gets released to.

31.6.2 Using Channels in Maintenance Workflow

Channel definitions for existing packages do affect incident projects. Matching channel packages get automatically branched inside and additional repositories for the channels are created. The server will build the channel package by aggregating the binary packages into the channel repositories.

The `_channel` files can be modified inside of the incident project if needed. This can be necessary when binary packages get renamed or added with this update. The modification will be part of the maintenance release request as simple submit actions.



This example shows the setup where selected binary packages get released also to a defined channel. The `openSUSE:11.4:SecurityChannel` project contains a `_channel` definition inside of the channel package. This one gets branched as well into the incident in case a matching channel does exist. Also, the additional repository gets added. The resulting binaries will be transfer via a release request to the code stream project (`openSUSE:11.4:Update`) and the special channel project.

32 Binary Package Tracking

Products and updates to them are often officially supported by a company. To allow giving such support, there is binary package tracking. This feature allows checking which exact version of a package was shipped at what time. This feature is often important for release managers, maintenance engineers, QA engineers and supporters.

OBS can track these binary packages and offer a database to search them.

32.1 Which Binaries Are Tracked?

All binaries which are released into projects providing `kind=maintenance_release` are tracked. In addition to that, the OBS administrator can configure additional projects via the `packtrack` setting in `BSConfig.pm`.

32.2 What Data Is Tracked?

In short the information to identify a binary, its building place and timestamps are tracked. In addition to that also information about possible successor versions or if the binary got removed in the meantime. If products do reference the repositories the search interface offers also a listing of products which are supposed to use it. Either as part of the product media itself or in one of its update repositories.

32.2.1 Binary Identifier

A binary is identified by the following information which is extracted from components of the file path of the binary:

- **Repository:** Where is the binary hosted?
- **Name:** Name of the binary file
- **Epoch:** The epoch version (optional, usually not used)
- **Version:** The version
- **Release:** The release number

- **Architecture:** The hardware architecture
- **Medium:** Name of the medium (exists only for product builds)

32.2.2 Binary Information

Additional information about a binary is information which gets updated when a binary gets added or replaced.

- operation, got the binary added, removed or modified
- publish time, aka the time when the repository gets published by OBS. This is not the same time as when the release action got invoked.
- build time
- obsolete time, exists only when a binary gets removed or replaced
- supportstatus, meta information about the level of support which is granted for the binary at the time of releasing it.
- updateinfo id from rpm-md repository
- maintainer of the binary who has prepared this update
- disturl, the exact identifier to the source and build repository

32.2.3 Product information

Additional information about products referencing to this binary.

- updatefor: the listed products do reference the repository as update channel.
- product: exists when the binary was part of a product medium

32.3 API Search Interface

The search of released binaries is provided via the following API endpoints:

- `/search/released/binary/id` : short form, just listing the matched binary identifiers. See the [reference documentation \(https://api.opensuse.org/apidocs/#!/Search/get_search_released_binary_id\)](https://api.opensuse.org/apidocs/#!/Search/get_search_released_binary_id).
- `/search/released/binary` : long form, provides all other tracked information as described above. See the [reference documentation \(https://api.opensuse.org/apidocs/#!/Search/get_search_released_binary\)](https://api.opensuse.org/apidocs/#!/Search/get_search_released_binary).

Please visit the before mentioned links to get details and several examples of how to use these API endpoints.

33 Scheduling and Dispatching

One of the major functionalities of OBS is to calculate always the current state, based on available sources, binaries and user configurations. In case a change happened it will trigger builds to achieve a clean state again. The calculation of the need of a build job is called scheduling here. The assignment of a build job to a concrete build host (aka worker) is called dispatching.

33.1 Definition of a Build Process

A build process is calculated and executed based on the following

- The sources of a package defined which dependencies are required at build time. For example, `BuildRequires` lines in spec files defined which other packages must get installed to build a package
- The project configuration of the package defines repositories and architectures to build for. In case other repositories are used as a base the configuration from there is also considered.
- Dependencies of packages which are required are considered as well.
- Constraints regarding the worker are considered. A package may require certain amount of resources or specific features to build. Check the constraints chapter for details. However, apart from this the build should be independent of the specific worker where the job gets dispatched to.

33.2 Scheduling Strategies

The defaults have the goal of creating an always reproducible state. This may lead to more builds than practically necessary, but ensures that no hidden incompatibilities exist between packages and also that the same state can later be achieved again (with a subsequent rebuild of the same sources and configurations). This can also lead to multiple builds of the same package in the case of dependency loops.

In some setups this may not be wanted, so each repository can be configured differently. The usual options to modify the project meta configurations can be used to configure different strategies. For example using osc:

```
osc meta prj -e YOUR_PROJECT
```

A repository is configured as following by default, however only the name attribute is required to be set.

```
# Example <repository>
  name="standard" rebuild="transitive" block="all" linkedbuild="off"> [...]
</repository>
```

33.2.1 Build Trigger Setting

The build trigger setting can be set via the "rebuild" attribute. Possible settings are

transitive

The default behavior, do a clean build of all dependant packages

direct

Just build the package with changed sources and direct dependant packages. But not indirect dependant packages.

local

Just build packages with changed sources.



Note

Note: You can run into dependency problems in case you select direct or local without noticing this in the build state. Your packages might not even be installable or have random runtime errors (like not starting up or crashing applications), even when they claim to be "succeeded". Also, you cannot be sure that you will be able to re-build them later. So never do an official shipment with this setting of a release. This knob is exposed to allow deliberate suppression of the strictly reproducible builds (for example, to limit burning CPU unnecessarily).

33.2.2 Block Mode

Usually the build of a package gets blocked when a package required to build it is still building at the moment. The "block" attribute can modify this behaviour:

all

The default behavior, do not start the build if a dependant package is currently building.

local

Just care about packages in your project for the block mode.

never

Never set a package to blocked.



Note

When using something other than “all” you will have to deal with the following problems:

- Intermediate builds can have dependency and runtime problems.
- Your packages will get built more often, take more resources on the server side. As a result the dispatcher will rate your repository down.

33.2.3 Follow Project Links

off

DEFAULT: do not build packages from project links

localdep

only build project linked packages if they depend on a local package result.

alldirect

build all packages from the linked projects. Indirectly linked projects get not build.

all

build all packages from direct and indirect linked projects

33.3 Release Number Handling

Most build formats define a version and a release number. The version is usually defined by the upstream project, while the release number is handled by the distributor. In addition to this Open Build Service differentiates between a source change counter (called check-in counter aka CI_CNT) and a build counter (BCNT). The check-in counter gets increased when any source change is done. The build counter gets increased when a build gets triggered due to any other reason than a source change.

The version and check-in counter are defined by the package source, while the build counter is stored per project, package, repository and architecture combination.

33.3.1 Build Counter Syncing via Architectures

The default configuration is to sync build counters across all architectures for a given project, package and repository. This can only be changed by the OBS administrator.

33.3.2 Build Counter Syncing via multiple packages

In some cases it is critical keep the build counter in sync via multiple packages. The old way to do so is to define a common `bcntsynctag` in package meta. The used string identifies the packages to be kept in sync. The package name is the default when no `bcntsynctag` exists. Please note that this affects always all flavors of a multibuild source.

The newer recommended way is to define these tags in the sources instead. These work also with git implementations and are more flexible.

34 Build Constraints

Build constraints provide a way for the user to specify build worker parameters that the Build Service will use to decide which build workers are "qualified" to undertake a given build.

They are intended to be used for defining known, hard requirements for successfully building a given package (for example, disk space, memory, or certain CPU functionality).

34.1 Build Resource Usage and Statistics

Ideally, the build constraints should be set to the minimum values that enable a build to succeed, because any higher setting than the minimum might unnecessarily reduce the number of build workers available to build the package.

Now, in the real world, we do not always have a precise idea of what the minimum values are for all the different build worker parameters that can be controlled using OBS build constraints. That need not deter us from setting build constraints, however. It is not necessary to wait for a build failure before setting minimum memory and disk space constraints, for example, because the OBS can give us reasonable values for memory and disk space based on a successful build.

Each successful build produces a file called `_statistics` which can be examined to get details of the resources our build consumed. We can then use this information to set appropriate values for the relevant build constraints.

34.1.1 Displaying the build statistics

The information from the `_statistics` file can be found in the Build Service web UI, by clicking on the build target we are interested in and then clicking on "Show resources". Alternatively, the `_statistics` file itself can be downloaded from the Build Service, either from the command line or using the OBS API.

34.1.1.1 Downloading the `_statistics` file using `osc`

Since the `_statistics` file is a build artifact produced by every successful build, it is always included among the build artifacts downloaded by `osc getbinaries`.

34.1.1.2 Downloading the `_statistics` file using the OBS API

If you are using the OBS API, the relevant call is:

```
GET /build/{project_name}/{repository_name}/{architecture_name}/{package_name}/_statistics
```

Important

When reviewing the build statistics, it's important to be aware that the numbers can vary significantly from build to build depending on build parallelism (e.g. `make -j`).

34.2 Constraint Qualifiers

In general, build constraints are specified in terms of a qualifier and a value. The qualifier expresses "what" - the build worker parameter that is to be constrained - and the semantics of the value depend on the qualifier. If the qualifier takes a numeric value, it generally expresses "how much", or, in other words, the minimum value (of that parameter) that a given build worker must meet in order to fulfill the constraint. But there are some qualifiers, such as `hostlabel`, that take a simple string value.

In the simplest cases, the qualifier is just a string. In more complicated cases, the qualifier can include subqualifiers and be modified by attributes. For example:

```
hardware:disk:size unit=G
```

The string `hardware:disk:size` in this example means "size, a subqualifier of disk, which is itself a subqualifier of hardware", and `unit=G` means "the value is expressed in units of Gigabytes". For a full treatment of constraint syntax, see [Section 34.4, "Constraint syntax"](#).

34.3 Constraint scope and precedence

Depending on the required scope, constraint qualifiers and their values can be set at four different levels: instance, project, package, and build recipe. Setting constraints at the OBS instance level is up to the administrator of the OBS instance and is covered in the OBS Admin Guide. Project-wide constraints are defined in the project configuration. Package constraints are defined in a special file, `_constraints`, in the packages sources. It is also possible to insert constraints directly in the individual build recipes (RPM spec files, Dockerfiles).

The constraints that are in force for a particular build are determined by merging all constraints defined at all levels: site, project, package, and build recipe. The merging is done in that order, with later settings overwriting earlier ones. That means, for example, that site-wide constraints can be overridden at any of the lower levels (project, package, build recipe), project-level constraints can be overridden at the package and build recipe levels, etc.

34.3.1 Project-scoped constraints

Build constraints for an entire project, or for specific repositories within it, or for specific architectures within those repositories, are defined in the project config by adding lines as so:

```
Constraint: <QUALIFIER> <VALUE>
```

The QUALIFIER syntax is the same as used in RPM spec files, documented in [Section 34.4, “Constraint syntax”](#). Within the project configuration, individual `Constraint` lines can be enclosed in guards to make a constraint apply only to certain architectures or repositories. For example:

```
%ifarch ppc ppc64 ppc64le
Constraint: hardware:cpu:flag power8
%endif
```

or

```
%if "%_repository" == "images"
Constraint: hardware:disk:size unit=M 4000
%endif
```



Important

Constraints set in project configuration affect not only the project itself, but also all projects that build against it.

For a full treatment of constraint syntax, see [Section 34.4, “Constraint syntax”](#).

34.3.2 Package-scoped constraints

Setting constraints at the package level is achieved by including an XML file called `_constraints` in the package sources. The Build Service will attempt to validate this file when it is committed (from the osc command line) or saved (in the web UI) to prevent invalid XML from reaching the Build Service.

Here is a simple example showing what a `_constraints` file might look like:

```
<?xml version="1.0"?>
<constraints>
  <hardware>
    <physicalmemory>
      <size unit="M">2000</size>
    </physicalmemory>
    <disk>
      <size unit="G">5</size>
    </disk>
  </hardware>
  <sandbox>kvm</sandbox>
  <hostlabel exclude="true">SLOW_CPU</hostlabel>
</constraints>
```

For details on constraint qualifiers and how to specify them in a `_constraints` file, see [Section 34.4, “Constraint syntax”](#).

34.3.3 Build recipe-scoped constraints

At the build recipe level, constraints are set by embedding lines containing constraint qualifiers and values directly in RPM spec files or Dockerfiles. Such lines take the form

```
#!/BuildConstraint: <QUALIFIER> <VALUE>
```

The `#` character at the beginning of the line causes the build recipe parser (RPM, Docker, podman) to ignore the whole line, but in combination with the `!` character signifies an OBS-specific directive that is picked up by a pre-processor within the OBS back-end.

Instead of specifying subqualifiers by nesting directives like in XML, colons are used. For example:

```
#!/BuildConstraint: linux:version:min 3.0
```

In this example, "linux:version:min" is the constraint qualifier and "3.0" is the value.

As described for project-scoped conditionals, above, `#!/BuildConstraint` lines can be guarded with various conditionals to limit their effect to certain architectures or, e.g., multibuild flavors.

Important

Be careful when setting build constraints. The idea is to use them to express minimum values for the various parameters, below which builds are likely to fail. If you set a constraint too high, you risk reducing the pool of compliant build workers down to a very

low number, or even to zero. (A low number of compliant build workers means your build may not start for a long time, and no compliant workers at all will cause the build to fail. See [Section 34.5, "Constraint Handling"](#) for details.)

Important

By default, constraints applied to build workers regardless of architecture. However, you may only be interested in certain architectures and not in others. See [Section 34.6, "Checking Constraints with `osc`"](#) for how to get architecture-specific information on which workers satisfy your constraints.

34.4 Constraint syntax

This section describes the various constraint qualifiers and their syntax.

34.4.1 `hostlabel`

The "hostlabel" qualifier is any string which can be assigned to build workers when starting the `bs_worker` process. Since its intended use is to restrict a build to specific workers, it should only be used after consultation with OBS administrators who have detailed knowledge of the build farm, and ideally as a negative definition, using the `exclude=true` attribute. Even then, keep in mind that `hostlabel` settings are not portable, since they are always specific to a given OBS instance and should therefore only be used as a last resort in situations that cannot be addressed by any of the other qualifiers.

An example use case is to run benchmarks in a reproducible way.

Example for `_constraints` file:

```
<constraints exclude="false">
  <hostlabel>benchmark_runner</hostlabel>
</constraints>
```

Example for project configuration:

```
Constraint: hostlabel benchmark_runner
```

Example for RPM spec file or Dockerfile:

```
#!/BuildConstraint: hostlabel benchmark_runner
```

34.4.2 sandbox

Defines the "sandbox" which is used for the build. The "sandbox" is the virtual environment in which the build takes place: each build worker is configured with a fixed sandbox type.

The configuration of this build constraint is typically left to OBS administrators, and there is usually no reason for a project or package maintainer to set it.

Example for _constraints file:

```
<constraints>
  <sandbox exclude="true">kvm</sandbox>
</constraints>
```

Example for project configuration:

```
Constraint: sandbox exclude="true" kvm
```

Example for RPM spec file or Dockerfile:

```
#!/BuildConstraint: sandbox exclude="true" kvm
```

34.4.3 linux

This is a category of constraints specific to the Linux kernel, applied to the kernel running on the build worker.

34.4.3.1 version

To require a specific Linux kernel version or version range.

Example for the _constraints file:

```
<constraints>
  <linux><version>
    <min>3.0</min>
    <max>4.0</max>
  </version></linux>
</constraints>
```

Example for project configuration:

```
Constraint: linux:version:min 3.0
```

```
Constraint: linux:version:max 4.0
```

Example for RPM spec file or Dockerfile:

```
#!/BuildConstraint: linux:version:min 3.0
#!/BuildConstraint: linux:version:max 4.0
```

34.4.3.1.1 `min`

Minimum kernel version.

34.4.3.1.2 `max`

Maximum kernel version.

34.4.3.2 `flavor`

A specific kernel flavor, such as `default` or `smp` (corresponding to the kernel packages `kernel-default` and `kernel-smp`, respectively).

Example for `_constraints` file:

```
<constraints>
  <linux>
    <flavor>default</flavor>
  </linux>
</constraints>
```

Example for project configuration:

```
Constraint: linux:flavor default
```

Example for RPM spec file or Dockerfile:

```
#!/BuildConstraint: linux:flavor default
```

34.4.4 `hardware`

To specify that build workers must meet certain minimum hardware specifications or possess certain hardware features.

34.4.4.1 **cpu**

To require a specific CPU feature.

34.4.4.1.1 **flag**

CPU features which are provided by the hardware. On Linux, these can be found in `/proc/cpuinfo`. The `flag` element may be used multiple times to require multiple CPU features.

Example for `_constraints` file:

```
<constraints>
  <hardware><cpu>
    <flag>mmx</flag>
    <flag>sse2</flag>
  </cpu></hardware>
</constraints>
```

Example for project configuration:

```
Constraint: hardware:cpu:flag mmx
Constraint: hardware:cpu:flag sse2
```

Example for RPM spec file or Dockerfile:

```
#!/BuildConstraint: hardware:cpu:flag mmx
#!/BuildConstraint: hardware:cpu:flag sse2
```

`EL0` is a special flag that that can be used on hardware that only supports level-0 exceptions, such as certain armv8l systems. This means that VMs or 32-bit kernels are not supported but userland is supported. This flag can be used to block builds on such hardware if no 64-bit kernel is available for a project.

Example for project configuration:

```
Constraint: hardware:cpu:flag exclude=true EL0
```

Example for RPM spec file or Dockerfile:

```
#!/BuildConstraint: hardware:cpu:flag exclude=true EL0
```

Additional flags are also reported for effective architecture level of the CPU. This includes the following flags:

- `power7`
- `power8`

- power9
- x86-64-v2
- x86-64-v3
- x86-64-v4

34.4.4.2 processors

To specify a minimum number of processor cores, virtual or physical (i.e., as reported by `/proc/cpuinfo`), provided by the build worker and usable for the build

Example for `_constraints` file:

```
<constraints>
  <hardware>
    <processors>4</processors>
  </hardware>
</constraints>
```

Example for project configuration:

```
Constraint: hardware:processors 4
```

Example for RPM spec file or Dockerfile:

```
#!/BuildConstraint: hardware:processors 4
```

34.4.4.3 jobs

Each build worker is configured with a given default for the `JOB` environment variable, which expresses how many parallel build processes the worker should be able to sustain. To require a minimal number of pre-configured parallel jobs for the build, use this qualifier.



Note

This specifies the number of parallel jobs the build tooling should use, so even though it is under `hardware` it is not actually a hardware requirement.

Example for `_constraints` file:

```
<constraints>
```

```
<hardware>
  <jobs>4</jobs>
</hardware>
</constraints>
```

Example for project configuration:

```
Constraint: hardware:jobs 4
```

Example for RPM spec file or Dockerfile:

```
#!/BuildConstraint: hardware:jobs 4
```

34.4.4.4 disk

Hard disk specific constraints.

34.4.4.4.1 size

To specify a minimum amount of free disk space, below which a build on the worker will not be attempted.

Example for _constraints file:

```
<constraints>
  <hardware>
    <disk>
      <size unit="G">4</size>
    </disk>
  </hardware>
</constraints>
```

Example for project configuration:

```
Constraint: hardware:disk:size unit=G 4
```

Example for RPM spec file or Dockerfile:

```
#!/BuildConstraint: hardware:disk:size unit=G 4
```

34.4.4.5 memory

To specify a minimum amount of RAM memory that the worker must be equipped with.

34.4.4.5.1 size

To require a minimum memory size, *including swap space*.

Example for `_constraints` file:

```
<constraints>
  <hardware>
    <memory>
      <size unit="M">1400</size>
    </memory>
  </hardware>
</constraints>
```

Example for project configuration:

```
Constraint: hardware:memory:size unit=M 1400
```

Example for RPM spec file or Dockerfile:

```
#!/BuildConstraint: hardware:memory:size unit=M 1400
```

34.4.4.6 physicalmemory

Memory specific.

34.4.4.6.1 size

To require a minimal memory size. Swap space is not taken into account here.

Example for `_constraints` file:

```
<constraints>
  <hardware>
    <physicalmemory>
      <size unit=M>1400</size>
    </physicalmemory>
  </hardware>
</constraints>
```

Example for project configuration:

```
Constraint: hardware:physicalmemory:size unit=M 1400
```

Example for RPM spec file or Dockerfile:

```
#!BuildConstraint: hardware:physicalmemory:size unit=M 1400
```

34.5 Constraint Handling

What happens when someone sets a constraint so high, that the OBS instance does not have even a single worker that meets it? What happens when just a few workers satisfy all the constraints, but all of them are busy building packages, or have been taken down for maintenance? This section describes how the OBS handles these "low compliant worker" situations.

34.5.1 At least one compliant worker is available

After determining which build workers satisfy the defined constraints for a given build, the scheduler checks if any of them are available to start building. If at least one is available, the build begins. The rest of this section describes the OBS's behavior when no compliant build workers are free to start building a given package.

34.5.2 More than half of existing workers satisfy the constraints

The build will stay in state "scheduled" until one of the compliant workers becomes available. No further notification is set.

34.5.3 Less than half of existing workers satisfy the constraints

The build will stay in state scheduled until one of the compliant workers becomes available. In addition, the dispatch details are set to tell the user that this build might take a long time to complete. The notification looks like this:

```
waiting for 4 compliant workers (4 down)
```

In this case, all four compliant workers are down. The notification helps the user understand why the build is not starting. The dispatch details can also be retrieved using the OBS API or, e.g., using the command `osc results -v`.

34.5.4 No existing workers satisfy the constraints

If no worker can handle the constraints defined by the package or project, the build fails. In such cases, the build log will mention why the build failed:

```
package build was not possible:

no compliant workers (constraints mismatch hint: hardware:processors sandbox)

Please adapt your constraints.
```

34.6 Checking Constraints with **osc**

You can check the constraints of a project or package with the **osc** tool. You have to be in an **osc** working directory.

```
osc checkconstraints [OPTS] [REPOSITORY] [ARCH] [CONSTRAINTSFILE]
```

Either you give a repository and an arch or **osc** will check the constraints for all repository / arch pairs for the package. A few examples:

```
geeko > osc checkconstraints
Repository      Arch           Worker
-----
openSUSE_Leap_42.2  x86_64        1
openSUSE_Leap_42.1  x86_64        1
```

If no file is given it takes the local `_constraints` file. If this file does not exist or the `--ignore-file` switch is set only the project constraints are used.

```
geeko > osc checkconstraints openSUSE_Leap_42.1 x86_64
Worker
-----
x86_64:worker:1
x86_64:worker:2
```

If a repository and an arch is given a list of compliant workers is returned.

Another command to verify a worker and display the worker information is `osc workerinfo`.

```
geeko > osc workerinfo x86_64:worker:1
<worker hostarch="x86_64" registerserver="http://localhost:5252" workerid="worker:1">
  <hostlabel>MY_WORKER_LABEL_1</hostlabel>
  <sandbox>chroot</sandbox>
```

```
<linux>
  <version>4.1.34-33</version>
  <flavor>default</flavor>
</linux>
<hardware>
  <cpu>
    <flag>fpu</flag>
    <flag>vme</flag>
    <flag>de</flag>
  </cpu>
  <processors>2</processors>
  <jobs>1</jobs>
</hardware>
</worker>
```

It returns the information of the desired worker.

35 Building Preinstall Images

Preinstall images can optionally be used to install a set of packages in one quick step instead via single package installations. Depending on the build host even snapshots with copy-on-write support may be used which avoids any IO.

A preinstall image can be used if it provides a subset of packages which is required for the build job. The largest possible image is taken if multiple are usable.

To use a preinstall image there needs to be a package container inside of the project or in a repository used by the build job. This package needs a `_preinstallimage` file. The syntax of it is spec file like, but just needs a `Name:` and at least one `BuildRequires:` line.

To ignore packages despite existing dependencies, use `#!BuildIgnore:` tags or `%if`.

Preinstall image build jobs are always preferred to allow the best effect of them. We recommend defining images for often used standard stacks.

Example `_preinstallimage` file for a basic preinstall image:

```
Name: base
BuildRequires: bash
#!BuildIgnore: brp-trim-desktopfiles
```


36 Authorization

36.1 OBS Authorization Methods

Each package is signed with a PGP key to allow checking its integrity on user's machines.

36.1.1 Default Mode

OBS provides its own user database which can also store a password. The authentication to the API happens via HTTP BASIC AUTH. See the API documentation to find out how to create, modify or delete user data. Also a call for changing the password exists.

36.1.2 Proxy Mode

The proxy mode can be used for esp. secured instances, where the OBS web server shall not get connected to the network directly. There are authentication proxy products out there which do the authentication and send the user name via an HTTP header to OBS. This also has the advantage that the user password never reaches OBS.

36.1.3 LDAP Mode

LDAP authentication code is still part of OBS, but due to the lack of any test cases it is currently not recommended to use it.

36.2 OBS Token Authorization

OBS provides a mechanism to create tokens for specific operations. This can be used to allow certain operations in the name of a user to others. This is esp. useful when integrating external infrastructure. The create token should be kept secret by default, but it can also be revoked at any time if it became obsolete or leaked.

36.2.1 Managing User Tokens

Tokens always belong to a user. A list of active tokens can be viewed using

```
osc token
```

```
osc token --delete <TOKEN>
```

36.2.2 Executing an Action

A token can be used to execute specific operations. This can be a source service trigger, a rebuild call, or release action. The setup needs to be prepared for the action. For example a source service for that can be setup with:

```
osc add git://....
```

The best way to create a token is bind it to a specific package. The advantage is that the operation is limited to that package, so less bad things can happen when the token leaks.

```
osc token --create <PROJECT> <PACKAGE>
```

Also, you do not need to specify the package at execution time. But keep in mind that such form only works when you run it on an as checkout of a package. Both commands below do the same thing but in a different way:

```
osc token --trigger <TOKEN>
```

```
osc api -X POST /trigger/runservice?token=<TOKEN>
```

A token can be registered as generic token, means allowing to execute all source services in OBS if the user has permissions. You can create such a token by skipping project/package on creation command:

```
osc token --create
```

In this case, you are forced to specify project/package along with the token. On the other hand, you are not limited from where you execute it. Again, two examples doing same thing:

```
osc token --trigger <TOKEN> <PROJECT> <PACKAGE>
```

```
curl -H "Authorization: Token <TOKEN>" -X POST https://$obs_instance/trigger/runservice?project=<PROJECT>&package=<PACKAGE>
```

You can also limit the token to a specific package. The advantage is that the operation is limited to that package, so less bad things can happen when the token leaks. Also you do not need to specify the package on execution time. Create and execute it with:

```
osc token --create <PROJECT> <PACKAGE>
```

```
osc token --trigger <TOKEN>
```

```
curl -H "Authorization: Token <TOKEN>" -X POST /trigger/runservice
```

A token to rebuild a package can be created and trigger by

```
osc token --operation rebuild --create <PROJECT> <PACKAGE>
```

```
osc token --operation rebuild --trigger <TOKEN>
```

```
curl -H "Authorization: Token <TOKEN>" -X POST https://$obs_instance/trigger/rebuild
```

A token to release a package can be created and trigger by

```
osc token --operation release --create <PROJECT> <PACKAGE>
```

```
osc token --operation release --trigger <TOKEN>
```

```
curl -H "Authorization: Token <TOKEN>" -X POST https://$obs_instance/trigger/release
```

37 Quality Assurance(QA) Hooks

OBS provides multiple hooks to place automated or manual tests at different points of time.

This chapter describes the different possibilities to provide and execute QA checks. The order of the items is sorted by the order in a typical development process. It is preferred to add a check as early as possible in the process to keep turn-around times small.

37.1 Source Related Checks

Things which can be verified based on a given source can be checked even before commit time on the developers workstation. This is the earliest possible point of time to add a check. But it can also optionally be enforced on the server side.

Automated source processing is done by source services in OBS world. Check the source service chapter how to use or write one. It is important to decide if the test case shall output warning messages and when it shall report an error by exit status.

Test cases in source services get usually applied to all packages of a project. (It is possible to execute it only for specific packages though.)

37.2 Build Time Checks

37.2.1 In-Package Checks

Checks running during the build of a package are usually test cases provided by the author of a package. However, the packager can also add simple checks, for example, for code that is known to break on version updates and might be forgotten when the package is touched the next time.

These test are often specific for a concrete package only. So it is typically executed in %check section of rpm spec files directly. In case the check can be used with multiple package source, it is a good idea to package the test case in an own package and just call it from the other packages.

rpm calls %check after %install section and before creating the actual checks.

SUSE distributions also provide build time checks to test the installed files inside of the build root. It is to be used for test cases which shall run on all packages which are built inside of a distribution. This hook can be used by installing a file to /usr/lib/rpm/brp-suse.d/ directory. These scripts also have the power to modify installed files if needed.

37.2.2 Post Build Checks

The standard tool to test binary packages for RPM-based distributions is rpmlint. DEB-based distributions use the lintian tool instead.

These checks are executed by the build script after a successful build. Note that these are executed as the standard user by default.

37.2.3 Post Build Root Checks

Files in `/usr/lib/build/checks/*` are executed as root user. Typical use cases are install tests of the build packages to ensure that the scripts inside of the packages are working in general.

37.2.4 KIWI Specific Post Build Root Checks

The file `/usr/lib/build/kiwi_post_run` is executed after KIWI jobs have finished. It can be used to run the appliance or to modify it. For example to package an appliance into an rpm.

37.3 Workflow Checks

Workflow steps, for example transferring packages from one project to another, are done via requests in OBS. At least when multiple parties are involved. One or more of these parties can be automated test cases. Or human manual approval steps.

Default reviews can be defined inside of projects and packages. A new request to a certain package does get the reviewers added defined in target projects and packages. Reviewers can be currently users, groups or the maintainers of a specified project or package.

37.3.1 Automated Test Cases

Open requests can be requested in an XML parseable way via the API running

```
osc api /request?states=review&user=auto-review-  
user&roles=reviewer&reviewstates=new&view=collection
```

osc can be used to accept or decline requests after running the automated test. It can also add a review comment which you can use to give a reason (for example, error messages) for accepting or declining a request. Requests, which are not tested, for example because they are of a not matching type (for example, deleting packages) needs to get also a review accept. Otherwise, this would block the process.

Glossary

.changes File

In OBS, a file with the file extension .changes to store changelog information.

See also [Changelog](#).

API

API stands for application programming interface. It lets your product or service communicate with other products and services without having to know how they're implemented.

The OBS API is located here: <https://api.opensuse.org> [↗](#).

The documentation for the API is located here: <https://api.opensuse.org/apidocs> [↗](#).

ApplImage

An application and its dependencies packaged as a single file which can run on many distributions without unpacking or installing.

Appliance

An image built and preconfigured for a specific purpose. Appliances usually consist of a combination of an application (for example, a Web server), its configuration, and an operating system (for example, SUSE Linux Enterprise Server). Appliances can be copied as-is onto a hard disk, an SSD, or started as a virtual machine (*deployed*).

See also [Operating System Image](#), [Image \(Image File\)](#).

Archive (Archive File)

An archive file contains a representation of usually multiple files and directories. Usually, archive files are also compressed. Archive files are the basis for binary packages ([Binary Package \(Binary\)](#)).

Attribute

Attributes can be added to projects or packages to add meta information or to trigger actions. For example, you can use the attribute OBS:AutoCleanup to delete a project after a certain amount of time.

Binary Package (Binary)

An archive file that contains an installable version of software and metadata. The metadata includes references to the dependencies of the main software. Dependencies are packaged as additional binary packages.

Formats of binary packages include RPM and DEB. In the OBS context, binary packages are sometimes also called *binaries*.

See also [Container](#), [Operating System Image](#), [Source Package](#), [Deb](#), [RPM](#), [KIWI](#), [Archive \(Archive File\)](#).

Branch

Personal copy of another repository that lives on your home project. A branch allows you to make changes without affecting the original repository. You can either delete the branch or merge it into the original repository with a submit request.

See also [Submit Request](#).

Bug

Issue that documents incorrect or undesirable behaviour

Bugowner

In OBS, *Bugowner* is a user role which can be set for a project or a package. However, ideally, set this role for individual packages only. Users with this role can only read data but they are responsible for reacting to bug reports.

See also [Maintainer](#).

Build

Generating ready-to-publish binaries, usually for a specific distribution and architecture.

Build Log

Output of the build process of a certain package.

See also [Build](#).

Build Recipe

Generic term for a recipe file for creating a package. A build recipe includes metadata, instructions, requirements, and changelogs. For RPM-based systems like SUSE, a .spec file is used and contains all the previous points. Debian-based systems use a debian directory which splits all the information.

See also [Spec File](#).

Build Requirement

Package requirements that are needed to create or build a specific package.

See also [Installation Requirement](#), [Build Recipe](#).

Build Result

The current state of a package. Example of a build result could be succeeded, failed, blocked, etc.

Build Root

Directory where the `osc` command copies, patches, builds, and create packages. By default, the build root is located in `/var/tmp/build-root/BUILD_TARGET`.

See also [Build Target](#).

Build Target

Specific operating systems and architecture to build for.

Changelog

Listing of a high-level overview sorted by date. An entry of a changelog can contain information about version updates, bug and security fixes, incompatible changes, or changes related to the distribution.

See also [.changes File](#).

Commit

A record of a change to one or more files. Each record contains the revision, the author, the date and time, a commit checksum, an optional request number, and a log message.

See also [Revision](#).

Container

An image file that contains a deployable version of software and metadata. Dependencies of the main software are also included, such as additional libraries.

Unlike operating system images, containers do not include an operating system. Unlike binary packages, containers are deployed and not installed. Formats of containers include AppImage, Docker, Snap, and Flatpak.

See also [Binary Package \(Binary\)](#), [Operating System Image](#), [Image \(Image File\)](#).

Deb

A package format created and used by the Debian distribution.

See also [Package](#), [RPM](#).

Decision

Decision made by a moderator (*Cleared* or *Favor*) when they receive a report of problematic content or user.

Dependency

See [Requirement](#).

Devel Project

A set of related packages that share certain features. For example, the devel project `dev-el:languages:python` stores all packages related to the Python programming language.

See also [Home Project](#), [Project](#).

Diff

See [Patch](#).

DISTURL

The DISTURL is a unique identifier of a source and its build setup. It is written usually written inside of the build result to be able to identify the origin. A DISTURL is structured as `obs://OBS_NAME/PROJECT/REPOSITORY/REVISION-PACKAGE`. It can be shown for example via

- RPM packages: `rpm -q --qf '%{DISTURL}\n' PACKAGE_NAME`
- Locally built container images: `podman inspect IMAGE_ID | grep org.openbuildservice.disturl`
- Container images built on Open Build Service: `skopeo inspect docker://URL | grep org.openbuildservice.disturl`

Docker

Docker is a lightweight virtualization solution to run multiple virtual units (containers) simultaneously on a single control host.

See also [Container](#).

Download Repository

An area containing built packages available for download and installation through Zypper or YaST. The download repository belongs to a project and is specific to a distribution. An example of a download repository could be `http://download.opensuse.org/repositories/PROJECT/openSUSE_Tumbleweed/`.

EULA

End User License Agreement. For software that needs a special license (usually non-open source) which the user needs to agree to before installing.

Fix

See [Patch](#).

Flags

A set of switches that determine the state of package or repository. This includes building, publishing, and generating debug information.

GA Project

The GA (general availability) project builds an initial release of a product. It gets frozen after releasing the product. All further updates get released via the [Update Project](#) of this project.

GPG Key

An encryption key pair that in the context of OBS is used to verify the owner of the repository and packages.

Home Project

Working area in OBS for uploading and building packages. Each home project starts with `home:USERNAME`.

See also [Project](#).

Image (Image File)

An image file contains a bit-wise representation of the layout of a block device. Some types of image files are compressed. OBS allows building multiple types of image:

[Operating System Image](#), [Container](#)

Image Description

Specification to define an appliance built by KIWI. The image description is a collection of files directly used by KIWI (`config.xml` and `*.kiwi`), scripts, or configuration data to customize certain parts of the KIWI build process.

See also [KIWI](#).

Incident

Describes a specific problem and the required updates. If the problem exists for multiple code streams, one incident covers all of them. An incident is started by creating a maintenance incident project and the update get built here.

Installation Requirement

Package requirements that are needed when the package is installed.

KIWI

A tool to build operating system images. It can create images for Linux supported hardware platforms or for virtualization systems.

See also [Image \(Image File\)](#).

License

Written contract to specify permissions for use and distribution of software.

See also [Project](#).

Link

A concept that defines a relationship between a source and a target repository.

See also [Project](#).

Maintainer

In OBS, *Maintainer* is a user role which can be set for a project or a package. Users that have this role in a project can add, modify, and remove packages and subprojects, accept submit requests, and change metadata.

See also [Bugowner](#).

Maintenance Project

A project without sources and binaries, defined by the maintenance team. Incidents are created as sub projects of this project.

See also [Incident](#).

OBS Package

OBS packages contain the sources that are necessary to build one or more binary packages or containers. The content of OBS packages varies. In general, there is always a source file (such as a TAR archive of the upstream sources) and a build recipe.

To build an RPM package in OBS, you need a spec file as your build recipe, for example. An OBS package can also contain other files, such as a change log and patches.

OBS packages, unlike the name “package” suggests, do not consist of a single file. Instead, they are directories of a version-controlled repository. However, unlike most directories, they cannot contain subdirectories. (You can use subdirectories to simplify your work with the checked-out package but you cannot submit these directories.)

Open Build Service (OBS)

A Web service to build binary packages, containers and operating system images from source.

The term “Open Build Service” is used to speak about the server part of the build service.

Unlike the term openSUSE Build Service, the term Open Build Service refers to all instances.

openSUSE Build Service

A specific Web service instance of [Open Build Service \(OBS\)](#) from the openSUSE project at <http://build.opensuse.org> .

Operating System Image

An image file that contains an operating system. The operating system can be either installable or deployable. Depending on their purpose, operating system images are classified into: [Product Image](#), [Appliance](#), [Virtual Machine Image](#)

Formats of operating system images include ISO, Virtual Disk, and PXE Root File System. See also [Binary Package \(Binary\)](#), [Image \(Image File\)](#), [KIWI](#).

osc

A command line tool to work with OBS instances. The acronym osc stands for *openSUSE commander*. osc works similarly to SVN or Git.

See also [Open Build Service \(OBS\)](#), <https://github.com/openSUSE/osc> ↗.

Overlay File

A directory structure with files and subdirectories used by KIWI. This directory structure is packaged as a file (`root.tar.gz`) or stored below a directory (named `root`). The contents of the directory structure is copied over the existing file system (overlaid) of the appliance root. This includes permissions and attributes as a supplement.

See also [Appliance](#), [KIWI](#).

Package

OBS handles very different types of software package:

[Source Package](#), [OBS Package](#), [Binary Package \(Binary\)](#)

See also [Container](#).

Package Repository

A place where installable packages are available. This can be either from a media like CD, DVD, or from a remote online repository.

Official repositories can be divided into oss software (licensed under an open source license) and non-oss (for software released under other, non-open source licenses). Additionally, there are update source, and debug repositories as well.

Package Requirement

See [Requirement](#).

Patch

Textual differences between two versions of a file.

See also [Patch File](#).

Patch File

A file that contains a patch with the file extension `.diff` or `.patch`.

See also [Patch](#).

Product Image

An image that allows installing an operating system, usually from a removable medium, such as a USB disk or a DVD onto a hard disk or SSD.

Live images are a special case of operating system images. They can be run directly a USB disk or DVD and are often but not always installable.

See also [Operating System Image](#), [Image \(Image File\)](#).

Project

Unit which defines access control, repositories, architectures, and a set of packages containing sources.

Project Configuration

Settings to define the setup of the build system, usually to switch on or off certain features during the build or to handle circular dependencies.

See also [Project](#).

Publishing

Finished process when a package is successfully built and available in the download repository.

See also [Download Repository](#).

Release Project

A release project is hosting a release repository which is not building any packages ever. It is only used to copy sources and binaries to this project on a release event.

Repo File

A file with the name `PROJECT.repo` inside the download repository. The file contains information about the name of the repository, the repository type, and references to the download repository and the GPG key.

See also [Download Repository](#).

Repository

A distribution-specific area that holds dependencies required for building a package.

See also [Download Repository](#).

Requirement

In the OBS context, package requirements that are needed to create, build, or install a package.

See also [Build Requirement](#), [Installation Requirement](#).

Revision

A unique numeric identifier of a commit.

See also [Commit](#).

RPM

A package format. It stands for recursive acronym RPM Package Manager. Mainly used by SUSE, Red Hat, u.a.

See also [Deb](#), [Package](#).

Sandbox

Isolated region of a host system which runs either a virtual machine or a change root environment.

See also [Build Root](#).

Service File

An XML file that contains metadata required for building a package. This includes version information, upstream source repository, and actions.

Source

Original form, mostly written in a computer language.

See also [Package](#).

Source Link

See [Link](#).

Source Package

Source packages contain content similar to an OBS package but they are packaged in an archive file. They are also meant to allow building a single binary package or container format only. However, source packages allow rebuilding outside of an Open Build Service context.

An example of source packages are SRPMs which contain the source for accompanying RPM binary packages.

See also [Binary Package \(Binary\)](#), [Archive \(Archive File\)](#).

Source Service

A tool to validate, generate, or modify a source in a trustable way.

See also [Source](#).

Spec File

A file that contains metadata and build instructions. Metadata includes a general package description and dependencies required for building and installing the package.

See also [Build Recipe](#), [Patch](#), [Source](#).

Submit Request

Asking for integrating changes from a branched project.

Subproject

A child of a parent project.

See also [Devel Project](#), [Home Project](#), [Project](#).

SUSE Package Hub

An OBS project reachable under [openSUSE:Backports](#). It is a subset of openSUSE Factory which does not contain version updates and does not conflict with official packages supported by SUSE Linux Enterprise.

Target

A specific distribution and architecture, for example, openSUSE Tumbleweed for x86-64.

Also referenced as *build target*.

Update Project

A project which provides official updates for the products generated in the [GA Project](#). The update project usually links sources and repositories against the [GA Project](#).

See also [Release Project](#), [GA Project](#).

Virtual Machine Image

An image which is built (and sometimes preconfigured) to be the basis of virtual machines. Such images can usually be copied to the target computer and run as-is. As such, there is some overlap between virtual machine images and appliances.

See also [Operating System Image](#), [Image \(Image File\)](#).

Watchlist

A list of repositories that the user is interested in, available in the OBS Web UI.

Working Copy

See [Working Directory](#).

Working Directory

A directory on your local machine as a result from a **osc checkout** call for working and building before submitting your changes to an OBS instance.

Zypper

A command line package manager to access repositories, solve dependencies, install packages, and more.

A GNU Licenses

This appendix contains the GNU General Public License version 2 and the GNU Free Documentation License version 1.2.

GNU General Public License

Version 2, June 1991

Copyright (C) 1989, 1991 Free Software Foundation, Inc. 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

GNU GENERAL PUBLIC LICENSE TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License.

The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say,

a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation

in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive

it, in any medium, provided that you conspicuously and appropriately publish on each copy

an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that

refer to this License and to the absence of any warranty; and give any other recipients of the

Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a

work based on the Program, and copy and distribute such modifications or work under the

terms of Section 1 above, provided that you also meet all of these conditions:

a). You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b). You must cause any work that you distribute or publish, that in whole or in part contains

or is derived from the Program or any part thereof, to be licensed as a whole at no charge to

all third parties under the terms of this License.

c). If the modified program normally reads commands interactively when run, you must

cause it, when started running for such interactive use in the most ordinary way, to print or

display an announcement including an appropriate copyright notice and a notice that there

is no warranty (or else, saying that you provide a warranty) and that users may redistribute

the program under these conditions, and telling the user how to view a copy of this License.

(Exception: if the Program itself is interactive but does not normally print such an announce-

ment, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object

code or executable form under the terms of Sections 1 and 2 above provided that you also

do one of the following:

a). Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

b). Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c). Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order,

agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER

EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the “copyright” line and a pointer to where the full notice is found.

one line to give the program’s name and an idea of what it does.
Copyright (C) yyyy name of author

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) year name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details
type `show w`. This is free software, and you are welcome
to redistribute it under certain conditions; type `show c`
for details.
```

The hypothetical commands `show w` and `show c` should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than `show w` and `show c`; they could even be mouse-clicks or menu items--whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a “copyright disclaimer” for the program, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright
```

```
interest in the program `Gnomovision'
(which makes passes at compilers) written
by James Hacker.

signature of Ty Coon, 1 April 1989
Ty Coon, President of Vice
```

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the [GNU Lesser General Public License \(http://www.fsf.org/licenses/lgpl.html\)](http://www.fsf.org/licenses/lgpl.html) instead of this License.

GNU Free Documentation License

Version 1.2, November 2002

Copyright (C) 2000,2001,2002 Free Software Foundation, Inc. 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document “free” in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondly, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of “copyleft”, which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The “Document”, below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as “you”. You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A “Modified Version” of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A “Secondary Section” is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document’s overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The “Invariant Sections” are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The “Cover Texts” are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A “Transparent” copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not “Transparent” is called “Opaque”.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The “Title Page” means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, “Title Page” means the text near the most prominent appearance of the work’s title, preceding the beginning of the body of the text.

A section “Entitled XYZ” means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as “Acknowledgements”, “Dedications”, “Endorsements”, or “History”.) To “Preserve the Title” of such a section when you modify the Document means that it remains a section “Entitled XYZ” according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document’s license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document,

and from those of previous versions (which should, if there were any, be listed in the History

section of the Document). You may use the same title as a previous version if the original

publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship

of the modifications in the Modified Version, together with at least five of the principal authors

of the Document (all of its principal authors, if it has fewer than five), unless they release

you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright

notices.

F. Include, immediately after the copyright notices, a license notice giving the public permis-

sion to use the Modified Version under the terms of this License, in the form shown in the

Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts

given in the Document’s license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled “History”, Preserve its Title, and add to it an item stating at

least the title, year, new authors, and publisher of the Modified Version as given on the Title

Page. If there is no section Entitled “History” in the Document, create one stating the title,

year, authors, and publisher of the Document as given on its Title Page, then add an item

describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Trans-

parent copy of the Document, and likewise the network locations given in the Document for

previous versions it was based on. These may be placed in the “History” section. You may

omit a network location for a work that was published at least four years before the Document

itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled “Acknowledgements” or “Dedications”, Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled “Endorsements”. Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled “Endorsements” or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version’s license notice. These titles must be distinct from any other section titles.

You may add a section Entitled “Endorsements”, provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled “History” in the various original documents, forming one section Entitled “History”; likewise combine any sections Entitled “Acknowledgements”, and any sections Entitled “Dedications”. You must delete all sections Entitled “Endorsements”.

COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an “aggregate” if the copyright resulting from the compilation is not used to limit the legal rights of the compilation’s users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document’s Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled “Acknowledgements”, “Dedications”, or “History”, the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See <http://www.gnu.org/copyleft/>.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License “or any later version” applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (c) YEAR YOUR NAME.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.2
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.
A copy of the license is included in the section entitled “GNU
Free Documentation License”.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the “with...Texts.” line with this:

```
with the Invariant Sections being LIST THEIR TITLES, with the
Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.