Open Build Service

Open Build Service: Reference Guide

by Adrian Schröter and Frank Schreiner

Publication Date: 04/15/2016

SUSE LLC
10 Canal Park Drive
Suite 200
Cambridge MA 02141
USA
https://www.suse.com/documentation

Copyright © 2006– 2016 Novell, Inc. 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 Novell trademarks, see the Novell Trademark and Service Mark list http://www.novell.com/company/legal/trademarks/tmlist.html. Linux* is a registered trademark of Linus Torvalds. All other third party trademarks are the property of their respective owners. A trademark symbol (®, ™ etc.) denotes a Novell trademark; an asterisk (*) denotes a third party trademark.

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

Contents

About this Guide ix

1 Distributing Free Software: Herding Cats 1

1.1 Goals of the OBS 1

Automated software builds for packagers 1 • Reproducibility for Free Software projects and Independent Software Vendors (ISV) 2 • Easy access to software for users 2

1.2 History of the OBS 2

1.3 Future of the OBS 2

2 OBS Concepts 4

2.1 Project organization 4

Project meta data 4 • Project build configuration. 5 • Project build macro configuration 6 • An OBS Package 6

2.2 The API 6

2.3 The OBS Interconnect 6

2.4 Download on Demand Repositories (DoD) 7

Motivation 7 • XML Document Hirarchy 7 • The Daemon 7 • The <download> element 8 • The <master> subelement 8 • The <pubkey> subelement 9 • Repository Types 9

2.5 Integrate External Source Repositories 13

Motivation 13 • Creating a external SCM reference 13 • Working with local checkouts 14 • Manage your build description in SCM 14

2.6 Attribute System 15

2.7 Automatic source processing 15

3 Build Process 16

3.1 How is a build process defined 16

3.2 How does a build process work 16

Preinstall phase 16 • Install Phase 16 • Package build 17 • Post build steps 17

3.3 Different ways to build 17

3.4 Security aspects 17

4 Source Management 18

4.1 Find Package Sources 18

5 Request And Review System 19

5.1 What a request looks like 19

Action types 20 • Request states 21 • Reviewers 22 • Request creation 22 • Request operations 22

6 Package Formats 24

6.1 Setting up a format 24

6.2 Recipe Specials 24

spec 24 • deb 24 • pkg 25 • kiwi appliance 25 • kiwi product 25 • simpleimage 26

7 Build Configuration 27

7.1 Configuration File Syntax 27

Required: <PACKAGES> 27 • Support: <PACKAGES> 27 • Preinstall: <PACKAGES> 27 • Keep: <PACKAGES> 28 • VMInstall: <PACKAGES> 28 • Runscripts: <PACKAGES> 28 • Order: <PACKAGE_A>:<PACKAGE_B> 28 • ExportFilter: <REGEXP> <ARCHITECTURES> 28 • PublishFilter: <REGEXP> 29 • Prefer: <PACKAGES> 29 • Prefer: <PACKAGE_A>:<PACKAGES> 29 • Ignore: <PACKAGES> 29 • Ignore: <PACKAGE_A>:<PACKAGES> 29 • FileProvides: <FILE> <PACKAGES> 30 • Substitute: <PACKAGE_A>

<PACKAGES> 30 • BuildFlags: <FLAG>:<VALUE> 30 • BuildEngine: <ENGINE> 30 • OptFlags: <TARGET_ARCHITECTURE> <FLAGS> 31 • Type: <TYPE> 31 • BinaryType: <TYPE> 31 • Target: <TARGET_ARCHITECTURE> 31 • HostArch: <HOST_ARCHITECTURE> 31 • type: <TYPE> 32 • Binarytype: <TYPE> 32 • Repotype: <TYPE[:OPTIONS]> ... 32 • Patterntype: <TYPES> 32 • Constraint: <SELECTOR> <STRING> 32

7.2 Macro Section 33

8 Source Services 34

8.1 Using services for validation 34

8.2 Different Modes when using services 35

Default Mode 35 • trylocal Mode 35 • localonly Mode 35 • serveronly Mode 36 • buildtime Mode 36 • disabled Mode 36

8.3 How are source service definitions stored 36

8.4 Dropping a source service again 37

8.5 How to write a source service 37

8.6 Interfaces for using source services 37

9 Signing 38

9.1 Package signing 38

9.2 Repository signing 38

9.3 Product signing 38

9.4 Configure sign key 38

10 Product Building 39

10.1 Requirements of a product 39

10.2 Possible shapes of a product 39

10.3 Product Setup in OBS 39
Package building 39 • Appliance setup with kiwi 39 • Product creation with kiwi 39 • Product Line Setup 40
10.4 Release Management 40
10.5 Typical Review Process Setup 41
11 Maintenance Support 42
11.1 Simple Project Setup 43
11.2 How to use the maintenance process 44
Way A: A maintainer builds an entire update incident for submission 44 • Way B: Submitting a package without branching 45 • Way C: Process gets initiated by the maintenance team 46 • Maintenance Incident Processing 46 • Incident gets released 47 • Incident gets reopened and re-released 47 • Using custom update IDs 48
11.3 OBS Internal Mechanisms 48
The workflow step by step 48 • Search for incidents 50
11.4 How to setup projects for doing a maintenance cycle 51
Defining a maintenance space 51 • Maintained Project Setups 52
11.5 Optional Channel Setup 52
Defining a channel 52 • Using channels in maintenance workflow. 52
12 Binary Package Tracking 55
12.1 Which binaries get tracked? 55
12.2 What data is tracked? 55
12.3 What data is tracked? 55
Binary Identifier 55 • Binary informations 56 • Product informations 56
12.4 osc search interface 57
12.5 webui search interface 57

12.6 API search interface 57
Examples 57

13 Cross Architecture Build 59

14 Administration 60

14.1 Server Components 60

14.2 Tools for the admin 60
obs_admin 60

14.3 Integrate OBS into your environment 60
Integrate Notifications 60 • Integrate your user management in OBS 60

15 Scheduling and Dispatching 61

15.1 How is a build process defined 61

15.2 Scheduling strategies 61
Build Trigger Setting 62 • Block Mode 62 • Follow Project Links 63

16 Build Job Constraints 64

16.1 hostlabel 64

16.2 sandbox 65

16.3 linux 65
version 65 • flavor 66

16.4 hardware 66
cpu 66 • processors 67 • disk 67 • memory 68 • physicalmemory 69

17 Build Preinstall Images 70

18 Authorization 71

18.1 OBS Authorization Methods 71
Default Mode 71 • Proxy Mode 71 • LDAP Mode 71

18.2 OBS Token Authorization 71
Manage tokens of a user 72 • Execute a source service 72 • Change Request Review Status 73

19 Quality Assurance(QA) Hooks 74

19.1 Source related checks 74

19.2 Build time check 74
In-package checks 74 • Post build checks 75 • Post build root checks 75 • KIWI specific post build root checks 75

19.3 Work-flow checks 75
Automated test cases 76

20 openSUSE Factory 77

20.1 openSUSE:Factory project 77

20.2 Devel Projects 77

Glossary 78

A How to work on this Book 80

B GNU Licenses 81

B.1 GNU General Public License 81

B.2 GNU Free Documentation License 84

About this Guide

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

These books are not considered to be focused on a special OBS version. They are also not a replacement of the documentation inside of the openSUSE wiki. But content from the wiki may get consolidated and included in these books.

Furthermore these books get written by the OBS community, please check the chapter how to work on these books. We request esp. experienced users and administrators to join and to contribute to these books. It is not required to be a very good or even native English speaker, because we rely on community editors to improve the language.

1 Distributing Free Software: Herding Cats

The basic challenge when distributing Free and Open Source Software (FOSS) is compiling into machine code all of the source code that the awesome community produces. It may sound simple, but it isn't. A typical GNU/Linux distribution consists of thousands of individual software projects. They build on top of each other, have functional dependencies, some of the projects are interdependent, conflict with each other and what not. In short, it's like herding cats. Another challenge distributing software is that, sadly, software is sometimes defective and people, even developers, make mistakes. One has to master the art of exchanging pieces without breaking the whole: in other words, software updates. The Open Build Service (OBS) is a tool that makes this possible. It is a generic system to build and distribute binary packages from sources in an *automatic*, *consistent* and *reproducible* way. The OBS is free software itself; you can run, copy, distribute, study, change and improve it. The source code and the developers are on github (https://github.com/openSUSE/open-build-service).

1.1 Goals of the OBS

The goals of the OBS are oriented towards the three groups of people that use it: People who contribute to the Free Software community by building binary packages of software, so called *packagers*. *Free Software projects and Independent Software Vendors (ISV)* who provide complete software stacks and distributions to their users. And, of course, *users* who are looking to install software on their GNU/Linux distribution.

1.1.1 Automated software builds for packagers

For people who contribute to the Free Software community by packaging software, so called packagers, the OBS tries to automate that task as much as possible. Especially the need to maintain a lot of different operating system (OS) installations, of different OS versions, on a lot of different hardware architectures is something the OBS makes easier.

1.1.2 Reproducibility for Free Software projects and Independent Software Vendors (ISV)

The OBS emphasizes making builds reproducible. It builds packages in a jailed environment that the OBS sets up from scratch each time and once the build succeeds it also rebuilds all the dependent software. This ensures that changes propagate throughout the entire stack and that users get a consistent product to install. Reproducibility is especially needed if the delivered software stack consists of a lot of parts that have functional dependencies. Because if something is reproducible, it is predictable, and that is what projects and ISVs aim for. If you can predict how each of the software packages within your GNU/Linux distribution influences the other, you can make sure that if you change one of them, the whole thing still works as expected. The OBS also has collaboration among developers baked-in to its structure and workflows, which helps projects and ISVs to harness the power of the open source development model.

1.1.3 Easy access to software for users

The OBS aims to make it easy for users to download the latest version of software as binary packages for their operating system. They don't have to know about tarballs, make install, package formats or other stuff from the 90's. Another priority is to make use of the package management tools of the users OS with which they are already familiar. Users should get their software from the OBS just as they get software from their OS supplier, no special tools needed. And once users are connected to a repository, they get served with maintenance or security updates or even add-ons for their software.

1.2 History of the OBS

From building packages on workstations over autobuild to OBS.

1.3 Future of the OBS

With the advent of test driven development and similar agile methods in recent years, the software development community has shown a clear trend towards automated and continuous integration (https://en.wikipedia.org/wiki/Continuous_Integration), deployment (https://en.wikipedia.org/wiki/Continuous_deployment) and delivery (https://en.wikipedia.org/

wiki/Continuous_delivery). The OBS is moving more and more from a standalone solution to be one link in the chain of software development tools. And while "software is eating the world", it is also becoming more complex every day. This prompts for the need to know who changed what and when. And when and how was that delivered to the users? Tracking and analyzing this, making trends visible to be able to act upon them is another challenge for the future. All of this besides the general challenge of keeping up with the fast and disruptive evolution of the Free Software ecosystem, which constantly supports new hardware, software, standards and tools makes for a very interesting future.

2 OBS Concepts

We describe here the high level concepts how Open Build Service is designed, manages it's content and is supposed to work.

2.1 Project organization

All sources and binaries which are hosted inside of OBS are organized in projects. A project is the container defining a larger task. It defines who is working there

2.1.1 Project meta data

A project gets configured in the project **`/source/$PROJECT/_meta`** path. It can be edited in web interface in the **`RAW Config`** tab or via command line via

```
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`** flag defines if package sources should get build. 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 build for an architecture. A publish also gets triggered, when the publish flag is enabled after a repository finished 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. A flag change is not triggering 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 buildenvironment. When this is disabled, the build has no influence on builds of other packages using this repository. In case a former 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. A flag change is not triggering 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 is checking 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.

2.1.2 Project build configuration.

A project gets 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 setup a build environment.

2.1.3 Project build macro configuration

The macro configuration is part of the build configuration in `/source/$PROJECT/_config`. It can be added at the end behind a `Macro:`

2.1.4 An OBS Package

An OBS Package is a sub name space below a project. It contains the specification of a single package build for all specified repositories.

2.2 The API

...

2.3 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.

2.4 Download on Demand Repositories (DoD)

2.4.1 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 availible
- simple to configure in project meta without need of shell access to repo servers

In download repotype's where package checksums can be verified (e.g. susetags, rpmmd and deb), it`s recommended to 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 **`sslfinger`** in order to avoid man in the middle attacks by peer verification.

2.4.2 XML Document Hirarchy

```
<project>
  <repository>
    <download>
      <master/> (optional)
      <pubkey/> (optional)
    </download>
  </repository>
</project>
```

2.4.3 The Daemon

The bs_dodup daemon periodically checks for new meta data 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
```

2.4.4 The <download> element

mandatory attributes:

- arch
- url
- repotype

2.4.5 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 repotype`s

- susetags
- rpmmd
- deb

This option could be defined by any valid url (http and https) to the origin of the repository but it´s stronly ecommended to use **`https`** with a **`sslfingerprint`** to bs_dodup possibility to verify it`s peeer 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 by calling the following command:

```
openssl s_client -connect <host>:<port> < /dev/null 2>/dev/null | openssl x509 -
fingerprint -noout
```

2.4.6 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 have a look at the repotype "deb" documentation below.

2.4.7 Repository Types

2.4.7.1 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" />
    <download arch="i586" url="http://mirror.example.org/path/to/iso"
 repotype="susetags" />
    <arch>x86_64</arch>
    <arch>i586</arch>
  </repository>
</project>
```

2.4.7.2 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"
 repotype="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"
 repotype="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>
```

2.4.7.3 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 seperated 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>
```

2.4.7.4 Arch Repository (arch)

Please be aware that there is no way atm 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"/>
    <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>
```

2.4.7.5 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>
```

2.5 Integrate External Source Repositories

2.5.1 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 a 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 have 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 incremental and need less storage on the server.

2.5.2 Creating a external SCM reference

External refernces 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 are doing the following:

- obs_scm: is mirroring the source. It is storing it as cpio archive, but for the build process this looks like a directory. It stores also additional information from the meta data to a file with obsinfo suffix.
- tar: creates a tar ball from the directory
- recompress: applies a compression on the tar ball
- set_version: reads the version from the obsinfo file and adapts the build descriptions to it.

Please note that only the first service (obs_scm) is running on the OBS server. The other services are running during the build process. They can also be replaced by any user by providing alternative implementations of them or writing their own service from scratch.

2.5.3 Working with local checkouts

Using

```
osc build
```

in any package with such a defintion will do the same process local. 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. No matter if they got pushed to the upstream server or not. However, you need to push it to 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.

2.5.4 Manage your build description in 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 description files.

2.6 Attribute System

...

2.7 Automatic source processing

...

3 Build Process

Each package build is done in a fresh environment. This is done to ensure that no dependencies are missing and every later build produces identical results.

3.1 How is a build process defined

...

3.2 How does a build process work

All sources and binaries which are hosted inside of OBS are organized in projects. Projects do host in sources inside of package containers again. The sources get build according to the repository configuration inside of the project.

3.2.1 Preinstall phase

The step of this phase depends on the type of the buildroot (building environment). OBS supports three different types buildroot:

- chroot
- XEN
- KVM

In the preinstall phase the OBS Worker creates a small base system (chroot or VM Image) with manually extracted packages (filesystem, coreutils, binutils, rpm/debutils, etc.), copies all neccessary build requirements into the base system.

3.2.2 Install Phase

Depending of the chosen build root the worker starts the XEN/KVM virtual maschine or enters the build root. If this was successfull the install phase reinstalls all base package from above and additionally all packages you've defined in your package recipe. After this phase the environment is ready to process the recipe.

3.2.3 Package build

Regarding the package type the buildroot executes different build commands. On RPM based distributions it's called "rpmbuild". On debian-based it's "dpkg-buildpackage" and on archlinux you use "pacman". Now everything depends on the quality and the type of your build recipe. In most cases the source code will be compiled now and get packed into the specific package format. On RPM based distributions there are additional security and cleanness checks called rpmlint, which have the goal to improve the quality of the packages.

3.2.4 Post build steps

The generated packages are taken from the Worker get signed by the OBS Signer and are published to the Repository.

3.3 Different ways to build

...

3.4 Security aspects

...

4 Source Management

..

4.1 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 webui. 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.

5 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.

5.1 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.
- descripton: 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.

Each request can only be accepted entirely or not at all. Therefore it may 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 he will not be allowed to accept the request.

5.1.1 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 please check the xml schema for requests.

5.1.1.1 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.

5.1.1.2 delete

A delete action can request removal of a project or package instance.

5.1.1.3 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.

5.1.1.4 set_bugowner

set_bugowner is similar to add_role, but removes all other bugowner roles in the target. This happens to have a uniq identifier to be used when assigning bugreports in external tools like bugzilla.

5.1.1.5 change_devel

can be used to update the devel package information in the target.

5.1.1.6 maintenance_incident

Official request to open a maintenance incident for official support products. This create by developers who want to start an official maintenance process. Find detail in the maintenance chapter about this. If accepted, a new maintenance incident project is created and package sources listed are copied there. All sources of all actions in one request will be merged into the same maintenance incident.

5.1.1.7 maintenance_release

Is used to release a ready maintenance update. Unlike maintenance_incident or submit, sources and binaries are copied without a rebuild. Details can be found in maintenance chapter.

5.1.1.8 group

Deprecated. Was never in a released OBS version. It is not allowed to be used anymore.

5.1.2 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 a declined requests remains an active, returnng to the submitter's active request queue (i.e. the submitter will need to take action now).
- revoked: The submitter has taken back his 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.

5.1.3 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.

5.1.3.1 Manual added reviews

Reviewers can be added manually by anyone involved in a request. This can be used to hand over a review. In that situatiation the new reviewer needs to be added and the own review needs to be accept. The request becomes declined when any of the reviewers are declining the request.

5.1.3.2 Automatic added reviews

Project and package objects can have users or groups with a reviewer role. They are added automatically to a request as reviwer when a request is created which has them as target. In case the project and package bother specify reviewer all of them are added to the request.

5.1.4 Request creation

The api is doing a number of checks on 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 refer 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.

5.1.5 Request operations

Requests can be modified only in very limited ways after creation. This is to avoid that reviewers reviewed a request but the nature of the requests is changing afterwards. 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

6 Package 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' recipe format, which builds products and isos, but uses rpm packages to setup the build process.

OBS currently supports 'rpm', 'deb' (debian), 'arch' as package formats and 'spec', 'dsc', 'kiwi' (both product and appliances), 'arch', 'preinstallimage' and 'simpleimage' as recipe formats.

6.1 Setting up a format

If no recipe format and binary format is specified in the prjconf, 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' recipe config, as a 'rpm' package format will select 'spec' builds as default. This configuration is done by adding a 'Type' line to the project configuration.

6.2 Recipe Specials

Each format has some specials

6.2.1 spec

For building rpms you need a .spec file for each package, containing its build description. 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 rpm's macro handling, so it's possible to use architecture specific BuildRequires, conditional builds and other advanced rpm features.

6.2.2 deb

To build *.deb files, which are used on debian-based distributions, you need the following files:

- debian.control

Here are the meta informations 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 do not get increased automatically during build unless the keyword DEBTRANSFORM-RELEASE gets added to the file.

6.2.3 pkg

This package format is for Archlinux and his derivates. To build a Archlinux pkg you need a PKGBUILD file containing the build description and the source tarball. PKGBUILD's have no macros like %{buildroot}. They have variables for example makedepens=(package1, package2). This variables are parsed by the OBS and uses them as dependencies. On Archlinux you typically build packages without subpackage. They are no *-dev oder *-devel packages.

6.2.4 kiwi appliance

KIWI is image producing plattform which builds images for ISO images and virtualization systems like QEMU, XEN and VMWare.

6.2.5 kiwi product

...

6.2.6 simpleimage

This format can be used to get simple rootfs tarball or squashfs image. It doesn't contain bootloader nor kernel. For advanced features, use kiwi. Use simpleimage for simple rootfs tarball/squashfs image of any distribution that is supported by OBS but doesn't have anything fancier than that.

Simple image uses similar syntax as spec file. Supported tags include Name, Version and BuildRequires which is the way how to specify list of packages to include. #!BuildIgnore can be also used. For additional customization %build phase is supported - run via bash. RPM macros are not supported, but $SRCDIR shell variable is available.

6.2.6.1 Example

```
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
```

7 Build Configuration

The build configuration is needed at least in each base project to define the setup of the build system. In addition to that it can be used to handle compatibility layers or to switch on or off certain features during the build.

The build config is reachable via API /source/PROJECT/_config, via **`osc meta prjconf`** or via webui via Project Config tab.

7.1 Configuration File Syntax

The build configuration is parsed by OBS in rpm style independent of the used packaging format. This means that you can use rpm features like macros or conditions in the configuration. All lines consist of the form 'keyword: arguments'.

<PACKAGES> mean binary package provides, but not the full filename. For example gcc, but not gcc-1.2.3.i386.rpm. Provides of a package can not be used either.

7.1.1 Required: <PACKAGES>

Required lines contain one or more packages that always get installed for package builds. A change in one of the requires packages triggers a new build.

7.1.2 Support: <PACKAGES>

Support lines contain one or more packages which also get installed for package builds, but a change in one of the 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'.

7.1.3 Preinstall: <PACKAGES>

Preinstall packages are needed to run the package installation program. They get unpacked before the VM gets started. Included scripts are NOT executed during this phase. However these packages will get installed again inside of the VM including script execution.

7.1.4 Keep: <PACKAGES>

To eliminate build cycles the to-be-built package is not installed by default, even when it is required. Keep can be used overwrite this behaviour. 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.

7.1.5 VMInstall: <PACKAGES>

VMInstall is 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.

7.1.6 Runscripts: <PACKAGES>

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

7.1.7 Order: <PACKAGE_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. <PACKAGE_A> will get installed before <PACKAGE_B>.

7.1.8 ExportFilter: <REGEXP> <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 architector for building. The regexp 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.

7.1.9 PublishFilter: <REGEXP>

The publish filter can be used limit the published binary packages in public repositories. Only packages which match the regexp will be put in exported repository.

7.1.10 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 chosing 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.

7.1.11 Prefer: <PACKAGE_A>:<PACKAGES>

It is possible to define the prefer only when one package is creating the choice error. This package must be listed first with a colon.

7.1.12 Ignore: <PACKAGES>

Ignore can be used to break dependencies. This can be usefull to reduce the number of needed packages or to break cyclic dependencies. Be careful with this feature, as breaking dependencies can have surprising results.

7.1.13 Ignore: <PACKAGE_A>:<PACKAGES>

It is possible to define the ignore only for one package. This package must be listed first with a colon.

7.1.14 FileProvides: <FILE> <PACKAGES>

OBS ignores dependendencies 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.

7.1.15 Substitute: <PACKAGE_A> <PACKAGES>

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

7.1.16 BuildFlags: <FLAG>:<VALUE>

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

- `vmfstype:` Defines a specific filesystem when building inside of a VM.
- `kiwiprofile:` builds the selected profile in kiwi appliance builds.

7.1.17 BuildEngine: <ENGINE>

Use an alternative build engine. This is still chained inside of the build script for security reasons. Alternatives are mock (for Fedora and RedHat) and debootstrap (for Debian). This will avoid differences in the build environment setup, but it will also have an effect on speed and reduced features. It should only be used when you want to emulate the distribution build. debbuild enginge will build deb files out of a spec file description. It can be used by the following definition inside of the projet build config:

```
Repotype: debian
Type: spec
Binarytype: deb
BuildEngine: debbuild
Support: pax
```

```
Support: debbuild
Keep: debbuild
```

7.1.18 OptFlags: <TARGET_ARCHITECTURE> <FLAGS>

rpm only: Optflags exports compiler flags to the build. They will only have an effect when the spec file is using $RPM_OPT_FLAGS. The target architecture may be * to affect all architectures.

7.1.19 Type: <TYPE>

Build recepie 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

7.1.20 BinaryType: <TYPE>

Binary type. This is the format of the files which will be the result of the build jobs. This gets usually set depending on the build recepie type. In some situations, for example a kiwi build job result gets converted into an rpm, it can be used to overwrite it. Possible values are: rpm, deb or none

7.1.21 Target: <TARGET_ARCHITECTURE>

rpm only: Defines the target architecture. This can be used to build for i686 on i586 schedulers for example.

7.1.22 HostArch: <HOST_ARCHITECTURE>

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

7.1.23 type: <TYPE>

Defines the 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.

7.1.24 Binarytype: <TYPE>

(OBS 2.4 or later): Sets the binary format used to setup 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 specifed, OBS deduces it from the recipe type. If the recipe type is also not set, OBS looks at the Preinstall package list for a hint.

7.1.25 Repotype: <TYPE[:OPTIONS]> ...

Defines the repository format for pulished repositories. Valid values are: none, rpm-md, suse, debian, hdlist2, arch, staticlinks. The OPTIONS parameter depends on the repository type, for rpm-md the known options are 'legacy' to create the old rpm-md format, 'deltainfo' or 'prestodelta' to create delta rpm packages, 'rsyncable' to use rsyncable gzip compression. To split the debug packages in an own published repository the option splitdebug:$REPOSIORY_SUFFIX can be used.

7.1.26 Patterntype: <TYPES>

Defines the pattern format. Valid values are: none (default), ymp, comps.

7.1.27 Constraint: <SELECTOR> <STRING>

(OBS 2.4 or later): Define build constraints for build jobs. The selector is a colon seperated list which gets a string assigned. Please see the build job constraints page for details.

7.2 Macro Section

The macro section of the build configuration starts behind a Macros: line and is only used on rpm builds. The content gets exported to be used by rpmbuild during the build. It can be used to export a define using "%key value" lines.

8 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.

Design goals of source services were:

- server side generated files must be easy to identify and must not be modifiable by the user. This way others user can trust them to be generated in the documented way without modifications.
- generated files must never create merge conflicts
- generated files must be a separate commit to the user change
- services must be runnable at any time without user commit
- services must be runnable on server and client side in the same way
- services must be designed in a safe way. A source checkout and service run must never harm the system of a user.
- services shall be designed in a way to avoid unnecessary commits. This means there shall be no time-dependent changes. In case the package already contains the same file, the newly generated file must be dropped.
- local services can be added and used by everybody.
- server side services must be installed by the admin of the OBS server.
- service can be defined per package or project wide.

8.1 Using services for validation

Source Services may be used to validate sources. This can happen per package, which is useful when the packager wants to validate that downloaded sources are really from the original maintainer. Or validation can happen for an entire project to apply general policies. These services can't get skipped in any package

Validation can happen by validating files (for example using the **`verify_file`** or **`source_validator`** service. These services just fail in the error case which leads to the build state "broken". Or validation can happen by redoing a certain action and store the result as new file as **`download_files`** is doing. In this case the newly generated file will be used instead of the committed one during build.

8.2 Different Modes when using services

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

8.2.1 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.

8.2.2 trylocal Mode

The trylocal mode is running the service locally when using current osc versions. The result gets committed as standard files and not named with _service: prefix. Additionally the service runs on the server by default, but usually the service should detect that the result is the same and skip the generated files. In case they differ for any reason (because the webui or api was used for example) they get generated and added on the server.

8.2.3 localonly Mode

The localonly mode is running the service locally when using current osc versions. 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.

8.2.4 serveronly Mode

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

8.2.5 buildtime Mode

The service is running inside of the build job, 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. Network access is not be available when the workers are running in a secure mode.

8.2.6 disabled Mode

The disabled mode is neither running the service locally or on the server side. 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 disabledrun
```

The result will get committed as standard files again.

8.3 How are source service definitions stored

The called services are always defined in a **_service** file. It is either part of the package sources or used project-wide when stored inside the **_project** package.

The _service file contains a list of services which get called in this order. Each service may define a list of parameters and a mode. The project wide services get called after the per package defined services. The _service file is an xml file like this example:

```
<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>
  <service name="update_source" mode="disabled" />
</services>
```

This example downloads the files via download_files service via the given URLs from the spec file. When using osc this file gets committed as part of the commit. Afterwards the krabber-1.0.tar.gz file will always be compared with the sha256 checksum. And last but not least there is the **`update_source`** service mentioned, which is usually not executed. Except when **`osc service disabledrun`** is called, which will try to upgrade the package to a newer source version available online.

8.4 Dropping a source service again

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

8.5 How to write a source service

..

8.6 Interfaces for using source services

..

9 Signing

OBS can sign build results.

9.1 Package signing

Each package is signed with a PGP key to allow checking its integrity on user's machines.

9.2 Repository signing

...

9.3 Product signing

...

9.4 Configure sign key

...

10 Product Building

OBS is used to build complete product lines. It is employed for openSUSE and also for SUSE Linux Enterprise products.

10.1 Requirements of a product

...

10.2 Possible shapes of a product

product media, appliance, repository...

10.3 Product Setup in OBS

....

10.3.1 Package building

A project is configured in the project meta data. This meta data contains ... These projects contain ...

10.3.2 Appliance setup with kiwi

...

10.3.3 Product creation with kiwi

...

10.3.4 Product Line Setup

Bigger products, like openSUSE or SUSE Linux Enterprise usually contain multiple media. These media must be kept in sync in various ways, for example for installation patterns, version numbering and other meta data. To guarantee this, OBS provides the product definition xml. Submitting such a product xml will create all necessary sources for meta packages and also all needed kiwi source files.

10.4 Release Management

Products needs usually multiple approval steps and should only be released on manual approval. OBS is providing a mechanism which allows to copy sources and binaries to another project in this case. To use this the release target must be configured inside of a repository defintion of the project meta data:

```
<project name="BuildProject">
  ...
  <repository name="BuildRepo">
    <releasetarget project="ReleaseProject" repository="ReleaseRepo"
 trigger="manual" />
    ...
  </repository>
</project>
```

The ReleaseProject must be create before and it should be configured as build disabled. The release can be triggered then via

```
osc release BuildProject
```

All sources and build results will be copied over from BuildProject to the ReleaseProject and the publishing will happen afterwards. The operation will happen in background since it can take quite some time depending on the project size.

10.5 Typical Review Process Setup

...

11 Maintenance Support

This chapter explains the setup and workflow of an 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 informations about the solved problems and defined groups of packages to achieve that. Also binary delta data may get generated to reduce the needed download size for the clients.

11.1 Simple Project Setup

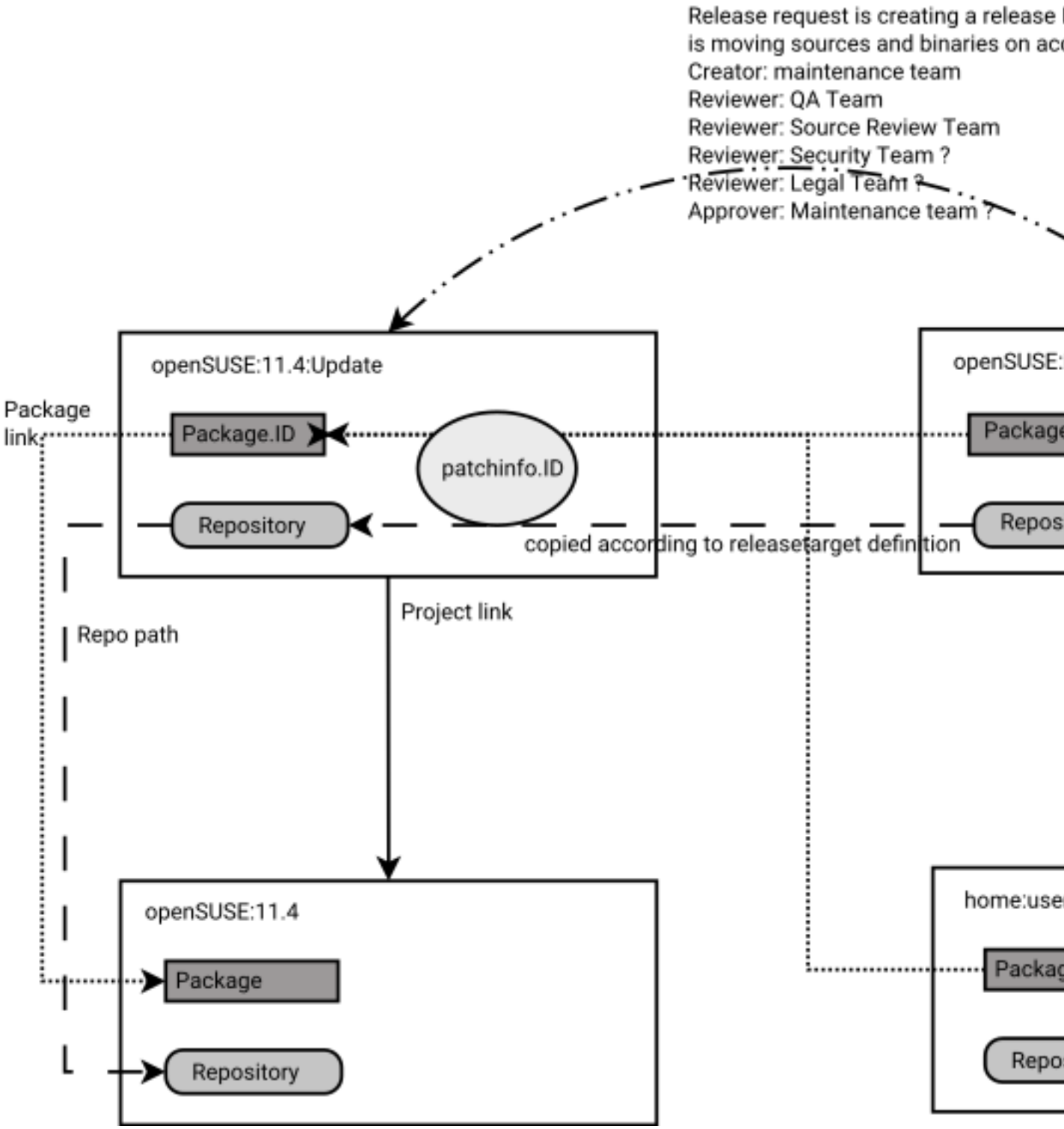


FIGURE 11.1: SIMPLE PROJECT SETUP

This figure is giving a basic 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 his project and start the maintenance process via doing a "maintenance" request.

- openSUSE:11.4 is the *GA Project* in this example. Frozen and not changing anymore.
- openSUSE:11.4:Update is the *Update Project* to release official updates.
- openSUSE:Maintenance is the *Maintenance Project*.
- openSUSE:Maintenance:IDxxx is the *Incident* project.

11.2 How to use the maintenance process

This describes all required steps by all involved persons from preparing to releasing a maintenance update.

11.2.1 Way 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
```

NOTE: both branch commands do support the --noaccess parameter, which will create a hidden project. This may be used when a not yet public known security shall get fixed.

Afterwards the user needs to do the needed modifications. Packages will be built and can be tested. Afterwards he may add informations 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, he has to create a maintenancerequest to ask the maintenance team to accept this as official update incident:

```
osc maintenancerequest
```

On accepting this request all sources of the entire project will get copied to the incident project and be rebuild. The origin project gets usually removed (based on the request cleanup options).

11.2.2 Way 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 specifing 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 need 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: it is also possible to specify an existing incident as target via the --incident parameter. The packages will be merged into that existing incident project then.

11.2.3 Way 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 webui 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 patchinfoi on his "my work" webui and osc views, so he knows that work is expected for him.

11.2.4 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.

11.2.5 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.

11.2.6 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 immediatly. 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.

11.2.7 Using custom update IDs

The used string of update IDs can be defined via the OBS:MaintenanceIdTemplate attribute value of the master maintenance project.

11.3 OBS Internal Mechanisms

OBS is tracking maintenance work and can be used as a database for future and past updates.

11.3.1 The workflow step by step

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 xsrcmd5 to avoid that other releases will affect this package instance. TBD: how to raise the source revision to current version manually ?
- 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. Eg 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.

11.3.2 Search for incidents

The webui 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 webui or osc to see requests or patchinfog were 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 him 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 webui 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')
```

11.4 How to setup projects for doing a maintenance cycle

11.4.1 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 it's 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.

11.4.2 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.

11.5 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.

11.5.1 Defining a channel

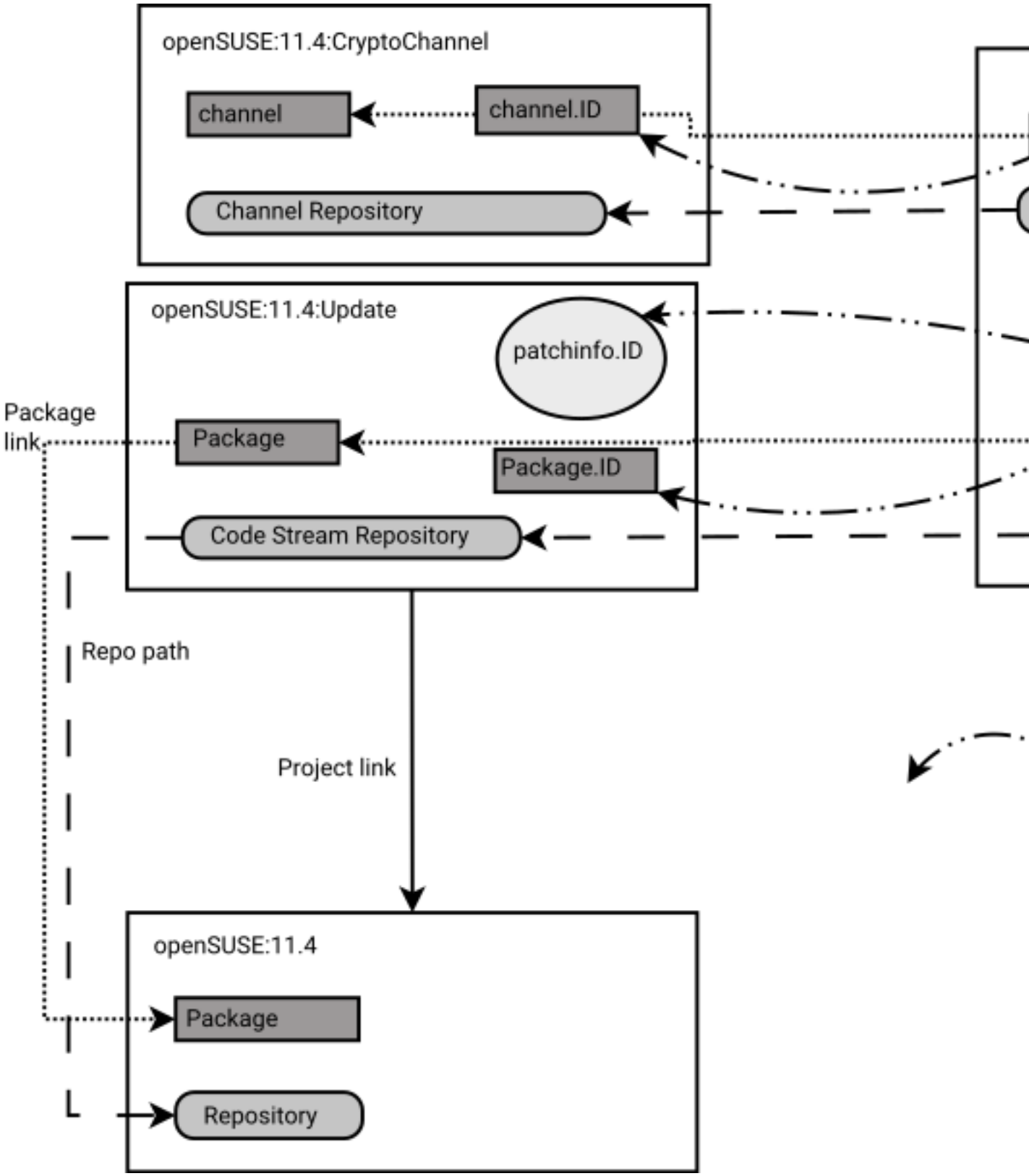
Channels get defined and maintained in a xml file inside of a package source. The file name of these lists must be _channel.

The file may contain a list of targets where binaries gets released to.

11.5.2 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 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.


openSUSE:11.4:CryptoChannel
channel
channel.ID
Channel Repository
openSUSE:11.4:Update
patchinfo.ID
Package
link
Package
Package.ID
Code Stream Repository
Repo path
Project link
openSUSE:11.4
Package
Repository

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.

12 Binary Package Tracking

Official products and their updates are often claimed to be supported. Multiple parties, like release managers, maintenance engineers, QA and support need to be able to check which exact versions of a package got released. OBS server can track these binaries and offers a database to search them.

12.1 Which binaries get tracked?

All binaries which get released in to projects providing kind = maintenance_release get tracked. In addition to that the OBS admininistrator may configure addition projects via the packtrack setting in BSConfig.pm

12.2 What data is tracked?

product media, appliance, repository...

12.3 What data is tracked?

In short the informations to identify a binary, it's building place and timestamps are tracked. In addition to that also information about possible successor versions or if the binary got removed meanwhile. If products do reference the repositories the search interface does offer also a listing of products which are supposed to use it. Either as part of the product media itself or in one of it's update repositories.

12.3.1 Binary Identifier

A binary gets identified by the following informations. This is basically the components of the storage path and filename of the binary.

- repository: where is the binary hosted
- name: the name of the binary file

- epoch: the epoch version (optional, usually not used)
- version: the version
- release: the release number
- architecture: the architecture
- medium: the medium name, this exists only for product builds

12.3.2 Binary informations

Additional informations of a binary are informations which get 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 gots 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

12.3.3 Product informations

Additional informations 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

12.4 osc search interface

not yet implemented

12.5 webui search interface

not yet implemented

12.6 API search interface

The search is provided via the generic xpath search interface. It is provided below the pathes:

- /search/released/binary/id : short form, just listing the matched binary identifiers
- /search/released/binary : long form, provides all other tracked informations as described above

12.6.1 Examples

To find the latest version of given glibc-devel binary in all products. Skipping old and revoked versions:

```
/search/released/binary?match=@name"=glibc-devel"+and+obsolete[not(@time)]
```

To find a specific version by given updateinfo id. This ID is visible in the update tools to the end user:

```
/search/released/binary?match=updateinfo/@id="OBS-2014-42"
```

To find a specific version by given disturl. Used to find all affected products by a certain build of a binary:

```
/search/released/binary?match=disturl="obs://..."
```

When got the specific package version got released the first time:

```
/search/released/binary?match=@name='kernel-default'+and+@version='1.0'+and
+@release='1'+and+@arch='i586'+and+supportstatus='l3'+and+operation='added'
```

All binaries in a given repository:

```
/search/released/binary?match=repository/[@project='BaseDistro3'+and
+@name='BaseDistro3_repo']
```

All binaries part of a product release:

```
/search/released/binary?match=product/[@project='openSUSE'+and+@name='openSUSE'+and
+(@arch='x86_64'+or+not(@arch))]
```

All binaries part of the update repositories of a product:

```
/search/released/binary?match=updatefor/[@project='openSUSE'+and
+@product='openSUSE'+and+(@arch='x86_64'+or+not(@arch))]
```

All binaries part of the update repositories of a versioned product:

```
/search/released/binary?match=updatefor/[@project='openSUSE'+and
+@product='openSUSE'+and+@version='13.2']
```

All binaries part of the update repositories of a versioned product (enterprise style):

```
/search/released/binary?match=updatefor/[@project='openSUSE'+and
+@product='openSUSE'+and+@baseversion='12'+and+@patchlevel='1']
```

13 Cross Architecture Build

OBS can build for architectures without having the hardware. This is done via emulation steps. However, this is not a classic build with cross tool chains. The advantage is that the build environments of the packages don't need support for cross tools.

14 Administration

This chapter describes the components of an OBS server and the typical administration tasks for an OBS administrator.

This chapter is not intended to describe special installation hints for a certain OBS version, please refer to the online resources or the README.SETUP file for this.

14.1 Server Components

To be written...

14.2 Tools for the admin

To be written...

14.2.1 obs_admin

To be written...

14.3 Integrate OBS into your environment

To be written...

14.3.1 Integrate Notifications

To be written...

14.3.2 Integrate your user management in OBS

To be written...

15 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.

15.1 How is a build process defined

A build process is calculated and executed based on the following

- The sources of a package defined which dependencies are required at build time. eg. `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. Please 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.

15.2 Scheduling strategies

The defaults have the goal of creating an always reproducible state. This may lead to more builds then practically necessary, but ensures that no hidden incompabilities 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 in a different way. 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>`

15.2.1 Build Trigger Setting

The build trigger setting can be set via the "rebuild" attribute. Possible settings are

transitive
: The default behaviour, do a clean build of all depending packages

direct
: Just build the package with changed sources and direct depending packages. But not indirect depending 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 a 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).

15.2.2 Block Mode

Usually the build of a package gets blocked when a package required to build it is still building atm. The "block" attribute can modify this behaviour:

all

The default behaviour, don't start the build if a depending package is currently building.

local

Just care about packages in your project for the block mode.

never

Never set a package to blocked.

Note

Note: When using something else then "all" you will have to deal with a number of 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.

15.2.3 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

all

treat packages from project links like local packages

16 Build Job Constraints

The build job constraints can define requirements to the hardware or software of the used build host. The constraints can be defined per package or for repositories.

The build constraints for a package is part of of the package sources. An _constraints source file can contain following values. The format of the file is XML and gets validated on submission.

The build constraints for entire project or specific repositories is part of the project config. It contains for each constraint a line

```
Constraint: <SELECTOR> <STRING>
```

The selector is a collon seperated list.

NOTE: If no build host is offering the constraints, the package will stay in scheduled state and never get dispatched.

16.1 hostlabel

The hostlabel is any string which can be assigned to build hosts when starting the bs_worker process. It can be used to run on specific hosts, which may be used for running benchmarks in a reproducible way. This constraints does also support to be defined as a negative definition via the exclude = true attribute. However, the hostlabel is always specific to one OBS instance. You should avoid it as much as possible, since building with this constraint in another instance is usually not possible. Use any of the other constraints if possible.

Example for _constraints file:

```
<constraints exclude="false">
  <hostlabel>benchmark_runner</hostlabel>
</constraints>
```

Example for project configuration:

```
Constraint: hostlabel benchmark_runner
```

16.2 sandbox

Defines the sandbox which is used for the build job. Sandboxes are chroot, xen or kvm environments. There is also the virtual secure sandbox, which allows building on xen or kvm. This constraints does also support to be defined as a negative definition via the exclude = true attribute.

Example for _constraints file:

```
<constraints exclude="true">
  <sandbox>secure</sandbox>
</constraints>
```

Example for project configuration:

```
Constraint: sandbox secure
```

16.3 linux

The linux kernel specific part.

16.3.1 version

To require a specific linux kernel version.

Example for _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
```

16.3.1.1 min

Minimal kernel version.

16.3.1.2 max

Maximal kernel version.

16.3.2 flavor

A specific kernel flavor like default or smp (from kernel packages kernel-default or kernel-smp).

Example for _constraints file:

```
<constraints>
  <linux>
    <flavor>default</flavor>
  </linux>
</constraints>
```

Example for project configuration:

```
Constraint: linux:flavor default
```

16.4 hardware

To require hardware or build instance features.

16.4.1 cpu

To require a specific linux kernel version.

16.4.1.1 flag

CPU features which are provided by the hardware. On linux they 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
```

16.4.2 processors

To require a minimal number of processors for the build job.

Example for _constraints file:

```
<constraints>
  <hardware>
    <processors>4</processors>
  </hardware>
</constraints>
```

Example for project configuration:

```
Constraint: hardware:processors 4
```

16.4.3 disk

Hard disk specific.

16.4.3.1 size

To require a minimal size of the disk.

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
```

16.4.4 memory

Memory specific.

16.4.4.1 size

To require a minimal 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
```

16.4.5 physicalmemory

Memory specific.

16.4.5.1 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
```

17 Build Preinstall Images

Preinstall images can optionally be used to install a set of packages in one fast 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. In addition also #! BuildIgnore: tags may be used to ignore packages despite of dependencies. Also usage of %if cases is possible.

Preinstall image build jobs get always preferred to allow the best effect of them. It is recommended to define images for often used standard stacks.

Example _preinstallimage file for a basic preinstall image:

```
Name: base
BuildRequires: bash
#!BuildIgnore: brp-trim-desktopfiles
```

18 Authorization

18.1 OBS Authorization Methods

Each package is signed with a PGP key to allow checking its integrity on user's machines.

18.1.1 Default Mode

OBS provides it's own user database which can also store a password. The authentification to the api happens via HTTP BASIC AUTH. Please see the api documentation to find out how to create, modify or delete user data. Also a call for changing the password exists.

18.1.2 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 authentification proxy products out there which do the authentification and send the user name via a HTTP header to OBS. This has also the advantage that the user password never reaches OBS.

18.1.3 LDAP Mode

LDAP authentification code is still part of OBS, but due to the lack of any test cases it is currently not recommended to use it.

18.2 OBS Token Authorization

OBS 2.5 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.

18.2.1 Manage tokens of a user

Tokens belong always to a user. A list of active tokens can received via

```
osc token
```

```
osc token --delete <TOKEN>
```

18.2.2 Execute a source service

A token can be used to execute a source service. The source service has to be setup for the package first, check the source service chapter for this. A typical example is to update sources of a package from git. A source service for that can be setup with

```
osc add git://....
```

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 and execute operation with

```
osc token --create
```

```
osc token --trigger <TOKEN> <PROJECT> <PACKAGE>
```

```
osc api -X POST /trigger/runservice?
token=<TOKEN>&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>
```

```
osc api -X POST /trigger/runservice?token=<TOKEN>
```

18.2.3 Change Request Review Status

NOT YET IMPLEMENTED

19 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.

19.1 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 optionaly be enforced on the server side.

Automated source processing is done by source services in OBS world. Please 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.)

19.2 Build time check

19.2.1 In-package checks

Checks running during the build of a package are usually test cases provided by the author of a package. However, also the packager might add simple checks, esp. for stuff which is known to break on version updates for example and might be forgotten when the package gets 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 distros is also providing build time checks for testing the installed files inside of the build root. It is to be used for test cases which shall run on all packages which are build inside of a distribuition. This hook can be used by installing a file to /usr/lib/rpm/brp-suse.d/ directory. These scripts have also the power to modify the installed files if needed.

19.2.2 Post build checks

The standard to test built package binaries is rpmlint for rpm based distros. Debian distributions are using the lintian tool.

These checks are executed by the build script after the a successful build happened. Please note that these are executed as standard user by default.

19.2.3 Post build root checks

Files in /usr/lib/build/checks/* are executed as root user. Typical uscases are install tests of the build packages to ensure that the scripts inside of the packages are working in general.

19.2.4 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 a rpm.

19.3 Work-flow checks

Work flow steps, for example transfering 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.

19.3.1 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 also be used to accept or decline the reviews of these requests after running the automated test. Also a comment can be given to show errors which appeared. It is important that requests, which are not tested, for example because they are of a not matching type (eg. deleting packages) needs to get also a review accept. Otherwise the process would get blocked.

20 openSUSE Factory

This chapter describes how the development of the future openSUSE distribution is done within OBS.

20.1 openSUSE:Factory project

The main project is openSUSE:Factory. This project is controlled by a small group which does review all submissions according to the policies. Submissions are possible via submit requests, which get reviewed by default by two groups: The Legal team and the code review team. ...

20.2 Devel Projects

The goal of openSUSE:Factory is to always have a working state. This is needed to allow all developer groups to use it as a base for testing their own, possibly experimental work in their own projects. To have a complete test of ...

Glossary

Open Build Service

The Open Build Service and its acronym OBS is used to speak about the server part of the build service. When speaking about OBS all possible instances are affected.

openSUSE Build Service

The openSUSE Build Service is the concrete instance of *Open Build Service* from the openSUSE project at http://build.opensuse.org.

Appliance

A software appliance is a preconfigured combination of an application (for example, a Web server) and its configuration, and includes an operating system (for example, SUSE Linux Enterprise Server). All these parts are integrated into a single image and can be deployed on industry hardware or on a virtual environment.

EULA

End User License Agreement. For software that needs a special license (usually non-open source) which the user has to agree to before installing.

KIWI

KIWI provides a complete operating system image solution. It can create images for Linux supported hardware platforms or for virtualization systems.

Overlay Files

Files which are created, removed, or modified in your testdrive are considered as *overlay files*. These files can be added later as a supplement to your appliance.

Binaries

Binaries are considered as build results of OBS Projects. Binaries can be reused in an environment to build further binaries. Currently OBS is supporting rpm, deb and all formats generated by *KIWI*.

GA Project

The GA 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.

Update Project

The update project is a *Release Project* which provides official updates for the products generated in the *GA Project*. The Update project is usually linking (sources and repositories) against the *GA Project*.

Maintenance Project

The maintenance project is a project without sources and binaries, defined by the maintenance team. *Incident*s are created as sub projects of this project.

Incident

The maintenance incident describes a concrete 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.

Release Project

A release project is hosting a release repository which is not building any packages ever. It is just used to copy sources and binaries to this project on a release event.

A How to work on this Book

These books are written with docbook and can be converted to html or pdf documentation. Please use the following command to checkout the official source of this book:

```
git clone https://github.com/openSUSE/open-build-service-documentation.git
```

Please check the README file for descriptions how to validate and generate them.

Please use the standard github work-flow to work on these books. This means fork your own copy of the repository, commit your changes there and create a pull request. This can be done at https://github.com/openSUSE/open-build-service-documentation

It is even possible to host instance specific content in the official subversion repository, it is just a matter to tag them correctly. Special parts of this documentation are tagged as `<para os="opensuse;meego">` for example. In this case the paragraph will become only visible when creating the openSUSE or MeeGo book.

B 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 announcement, 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) 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. Secondarily, 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 permission 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 Transparent 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.