User Guide

User Guide: Open Build Service

Publication Date: 01/23/2019

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

Contents

About this Guide

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

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

1 Available Documentation

The following documentation is available for OBS:

Book "Administrator Guide"

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

Article "Beginner's Guide"

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

Book "Best Practice Guide"

This guide offers step-by-step instructions for the most common features of the Open Build Service and the openSUSE Build Service.

Book "Reference Guide"

This guide covers ideas and motivations, concepts and processes of the Open Build Service and also covers administration topics.

User Guide

This guide is intended for users and developers who want to dig deeper into Open Build Service. It contains information on backgrounds, setting up your computer for working with OBS, and usage scenarios.

2 Feedback

Several feedback channels are available:

Bugs and Enhancement Requests

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

Bug Reports

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

Mail

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

3 Documentation Conventions

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

- `/etc/passwd`: directory names and file names
- *`PLACEHOLDER`*: replace *`PLACEHOLDER`* with the actual value
- `PATH`: the environment variable PATH
- **`ls`**, `--help`: commands, options, and parameters
- `user`: users or groups
- `package name` : name of a package
- Alt, Alt–F1: a key to press or a key combination; keys are shown in uppercase as on a keyboard
- *File*, *File* › *Save As*: menu items, buttons
- *Dancing Penguins* (Chapter *Penguins*, ↑Another Manual): This is a reference to a chapter in another manual.

- Commands that must be run with `root` privileges. Often you can also prefix these commands with the **`sudo`** command to run them as non-privileged user.

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

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

```
geeko > command
```

- Notices

Warning: Warning Notice

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

Important: Important Notice

Important information you should be aware of before proceeding.

Note: Note Notice

Additional information, for example about differences in software versions.

Tip: Tip Notice

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

4 Contributing to the Documentation

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

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

This guide is written in DocBook XML which can be converted to HTML or PDF documentation.

To clone the source of this guide, use Git:

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

To learn how to validate and generate the OBS documentation, see the file `README`.

To submit changes, use GitHub pull requests:

1. Fork your own copy of the repository.
2. Commit your changes into the forked repository.
3. Create a pull request. This can be done at https://github.com/openSUSE/obs-docu.

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

I Concepts

1 Supported Build Recipes and Package Formats

1.1 About Formats

OBS differentiates between the format of the build recipes and the format of the installed packages. For example, the spec recipe format is used to build RPM packages by calling `rpmbuild`.

In most cases, the build result format is the same as the package format used for setting up the build environment, but sometimes the format is different. An example is the KIWI build recipe format, which can build ISOs, but uses RPM packages to set up the build process.

OBS currently supports the following build recipe formats and packages:

SUPPORTED PACKAGE FORMATS

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

SUPPORTED BUILD RECIPE FORMATS

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

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

1.2 RPM: Spec

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

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

`.changes`
: the file which contains the changelog.

1.3 Debian: Dsc

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

`debian.control`
: The file contains the meta information for the package like the build dependencies or some description.

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

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

1.4 Arch: pkg

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

`PKGBUILD`

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

1.5 KIWI Appliance

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

For building an image in KIWI you need:

`my_image.kiwi`
: Contains the image configuration in XML format. Full XML schema documentation can be found https://suse.github.io/kiwi/development/schema.html.

`config.sh` **(optional)**
: configuration script that runs at the end of the installation, but before package scripts have run.

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

Note

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

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

1.6 SimpleImage

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

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

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

EXAMPLE 1.1: SIMPLEIMAGE FILE (`simpleimage`)

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

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

# Enable ssh
systemctl enable sshd
```

1.7 AppImage

II Setup

2 osc, the Command Line Tool

2.1 Installing and Configuring

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

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

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

For other systems, use your preferred package manager.

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

2.2 Configuring osc

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

Some useful options in the `~/.oscrc` file are described in the following list (all under the `general` section):

`apiurl` (string)

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

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

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

`extra-pkgs` **(list)**

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

`build_repository` **(string)**

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

`exclude_glob` **(list)**

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

`checkout_no_colon` **(bool)**

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

`use_keyring` **(bool)**

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

2.3 Usage

2.3.1 Getting Help

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

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

2.3.2 Using **osc** for the First Time

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

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

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

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

2.3.3 Overview of Brief Examples

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

List Existing Content on the Server

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

`osc ls` shows you a list of projects on OBS. Which OBS instance it shows depends on the option `apiurl` in the configuration file. By default, the openSUSE Build Server is used. If you need another server, use the `-A` option as shown in *Section 2.2, "Configuring osc"*.

Checkout Content

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

Update a Working Ddirectory

```
osc up
osc up [directory]
osc up *            # from within a project dir, update all packages
```

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

Upload Changed Content

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

Check the Commit Log

```
osc log
```

Show the status (which files have been changed locally)

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

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

Mark files to be Added or Removed on the Next Checkin

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

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

```
osc addremove
```

Generate a diff to view the changes

```
osc diff [file]
```

Show the Build Results of the Package

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

Show the Log File of a Package

(you need to be inside a package directory)

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

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

```
osc repourls [dir]
```

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

```
osc rebuildpac [dir]
```

Build a Package on Your Computer

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

Show Configured Platforms/Build Targets

```
osc platforms [project]
```

Show Possible Build Targets for Your Project

```
osc repos
```

Show Metadata

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

Edit Meta Information

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

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

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

Update Package Metadata on OBS with Metadata Taken from Spec File

```
osc updatepacmetafromspec [dir]
```

3 Project Configuration

3.1 About the Project Configuration

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

- Handle compatibility layers.
- Switch on or off certain features during the build.
- Decide which package is installed during build if there are circular dependencies.
- Handle user decisions like package providing the same or special macros, packages, or flags in the build environment.

To view the project configuration, use one of the following methods

- **With `osc`.** Use **`osc meta prjconf`** in your working directory of your project.
- **In the OBS Web UI.** Via the *Project Config* tab.
- **From the Local Build System.** Open one of the files in `/usr/lib/build/configs/*.conf` to see one of the default configurations that is being used.
- **With the OBS API.** Reachable via the `/source/PROJECT/_config` path.

3.2 Configuration File Syntax

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

```
keyword: arguments
```

In the following list, the placeholder *PACKAGES* indicates a package base name. For example, as a package name you need the base name like `gcc` but not the full name as in `gcc-1.2.3.i386.rpm`.

The following list contains a list of allowed keywords in the project configuration:

AVAILABLE KEYWORDS IN PROJECT CONFIGURATION

`BinaryType: TYPE` **(OBS 2.4 or later)**

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 recipe 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.

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

`BuildEngine: ENGINE`

Use an alternative build engine. This is still chained inside of the build script for security reasons. Alternatives are mock (for Fedora and Red Hat) 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 engine will build deb files out of a spec file description. It can be used by the following definition inside of the project build config:

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

`BuildFlags: FLAG:VALUE`

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

`vmfstype:`

Defines a specific file system when building inside of a VM.

`kiwiprofile:`

builds the selected profile in KIWI appliance builds.

`logidlelimit:`

Build jobs which don't create any output get aborted after some time. This flag can be used to modify the limit. Specify the seconds behind flag.

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

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

`ExportFilter: REGEX ARCH`

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

`FileProvides: FILE PACKAGES`

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

`HostArch: HOST_ARCH`

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

`Ignore: PACKAGES`

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

`Ignore: PACKAGE_A:PACKAGES`

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

`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 to overwrite this behavior. It is usually needed for packages like `make` that are used to build itself. Preinstalled packages are automatically kept, as the package installation program needs to work all the time.

`OptFlags: TARGET_ARCH FLAGS` **(RPM only)**

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

`Order: PACKAG_A:PACKAGE_B`

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

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

`Patterntype: TYPES`

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

`Prefer: PACKAGES`

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

`Prefer: 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.

`Preinstall: PACKAGES`

Are needed to run the package installation program. These packages 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.

`PublishFilter: REGEXP [REGEXP]`

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

`Repotype: TYPE[:OPTIONS]`

Defines the repository format for published 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:`*`REPOSITORY_SUFFIX`* can be appended, e.g.

```
Repotype: rpm-md splitdebug:-debuginfo
```

(the repository format may even be omitted to use the default type). This results in a debuginfo package repository being created in parallel to the package repository.

`Required:` *`PACKAGES`*

Contain one or more packages that always get installed for package builds. A change in one of these packages triggers a new build.

`Runscripts:` *`PACKAGES`*

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

`Substitute:` *`PACKAGE_A PACKAGES`*

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

`Support:` *`PACKAGES`*

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`.

`Target:` *`TARGET_ARCH`* **(RPM only)**

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

`Type:` *`TYPE`*

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

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

`VMInstall:` *`PACKAGES`*

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.

3.3 Macro Section

Macros are defined at the end of the project configuration. The macro section is only used on RPM builds.

The project configuration knows two possible definitions:

- **%define Macro Definitions.** Starting with a `%define` line and are used in the project configuration only. These definitions are *not* available inside the build root.
- **Other Macro Definitions.** Starting after the `Macros:` line and are exported into the `.rpmmacros` file of the build root. As such, these macro definitions can be used in a spec file.

3.3.1 Macros Used in Project Configuration Only

Inside the project configuration use `%define` to define your macros. You can use all the features that RPM supports except you cannot define functions or call external commands.

For example, you can define

```
%define _with_pulseaudio 1
```

3.3.2 Macros Used in Spec Files Only

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

EXAMPLE 3.1: STRUCTURE OF A MACRO DEFINITION

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

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

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

```
%
```

III Usage

4 Basic OBS Workflow

4.1 Setting Up Your Home Project

This section shows how to set up your home project with the command line tool `osc`. For more information about setting up your home project with the Web UI, see *Article "Beginner's Guide", Section 6 "Setting Up Your Home Project for the First Time"*.

This chapter is based on the following assumptions:

- You have an account on an Open Build Service instance. To create an account, use the Web UI.
- You have installed `osc` as described in *Section 2.1, "Installing and Configuring"*.
- You have configured `osc` as described in *Section 2.3.2, "Using `osc` for the First Time"*.

PROCEDURE 4.1: SETTING UP YOUR HOME PROJECT

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

   ```
   geeko > osc ls /
   ```

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

2. Configure your build targets with:

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

   The previous command shows a XML structure like this:

   EXAMPLE 4.1: XML STRUCTURE OF BUILD SERVICE METADATA

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

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

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

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

4. Add more `repository` elements as needed. Insert the information from *Step 1* into the `project` attribute.
   On openSUSE Build Service, you can normally use the attribute `repository` with the value `standard`. For example, to add openSUSE Leap as a build target, create an entry like:

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

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

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

4.2 Creating a New Package

This section covers how to create packages from a project hosted on GitHub (the “upstream project”). We assume that this project contains source code which you want to package for different SUSE distributions. We assume the setup of your home project in your OBS instance is already done. If not, refer to *Section 4.1, “Setting Up Your Home Project”*.

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

PROCEDURE 4.2: GENERAL PROCEDURE TO BUILD A RPM PACKAGE

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

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

      This will create `home:obsgeeko` in the current directory.

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

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

3. Download the source of the upstream project and save it in `home:obsgeeko/YOUR_PROJECT`.
4. Create a *spec file* which contains metadata and build instructions. For more information about spec files, see https://rpm-guide.readthedocs.io/en/latest/.
5. Create a new changelog and add your changes:
   a. To create a new changelog file or to update an existing changelog file with **`osc`**, use:

      ```
      geeko > osc vc
      ```

      The command will open an editor with the following content:

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

   b. Enter your changes in the editor.

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

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

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

6. Add all the files to your working directory:

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

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

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

   If you encounter problems, see *Section 4.3, "Investigating the Local Build Process"*.

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

```
geeko > osc commit
```

4.3 Investigating the Local Build Process

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

- *Section 4.3.1, "Build Log"*
- *Section 4.3.2, "Local Build Root Directory"*

4.3.1 Build Log

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

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

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

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

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

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

`.build.log`
: Contains the log.

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

`.build.packages`
: Contains the path to all object files.

4.3.2 Local Build Root Directory

If you build a package locally and you get a build error, investigate the problems in the build root directory directly. This is sometimes easier and more effective than only looking at the build log.

By default, the directory `/var/tmp/build-root/` is used as the *build root*. This is defined in the configuration file `~/.oscrc` using the key `build-root`.

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

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

When prompted, enter the `root` password.

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

The build root contains the following structure:

EXAMPLE 4.2: DIRECTORY STRUCTURE OF A BUILD ROOT (`/var/tmp/build-root/`)

```
/home/abuild/
└── rpmbuild
    ├── BUILD 1
    ├── BUILDROOT  2
    ├── OTHER  3
    ├── RPMS  4
    │   ├── i386
    │   ├── noarch
    │   └── x86_64
    ├── SOURCES  5
    ├── SPECS  6
    └── SRPMS  7
```

1. Contains directory named after the package name. In spec files, the name of the package directory is referenced using the `%buildroot` macro.
2. If the build process was unable to create a package, this directory contains all files and directories which are installed in the target system through the `%install` section of the spec file.
   If the package has been successfully built, this directory will be emptied.
3. Usually contains the file `rpmlint.log`.
4. If the build was successful, stores binary RPMs into subdirectories of architecture (for example, `noarch` or `x86_64`).
5. All source files from the working copy will be copied here.
6.
7. Stores source RPMs into this directory.

4.4 Adding Dependencies to Your Project

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

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

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

- *Section 4.4.1, "Adding Dependencies to Your Build Recipes"*
- *Section 4.4.2, "Associating Other Repositories with Your Repository"* (*layering*)
- *Section 4.4.3, "Reusing Packages in Your Project"* (*linking* and *aggregating*)

4.4.1 Adding Dependencies to Your Build Recipes

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

EXAMPLE 4.3: EXCERPT OF BUILD AND INSTALLATION REQUIREMENTS

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

4.4.2 Associating Other Repositories with Your Repository

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

When a package is needed, it can be installed from another other repository (see the note below).

To add another repository that can be used as build or installation requirements, do the following:

1. Open a terminal.

2. Edit the project metadata:

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

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

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

Note: Order Is Important

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

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

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

4. Add more `path` elements under the same `repository` element.

5. If necessary, repeat *Step 3* and *Step 4* to add `path` elements to `repository` elements of other distributions or releases.

4.4.3 Reusing Packages in Your Project

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

4.4.3.1 Linking a Package

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

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

The general syntax of the **`linkpac`** command is:

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

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

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

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

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

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

5 Building Package Formats

5.1 From Spec Files (RPM)

To create an RPM package, you need:

- the spec file which is the build recipe
- a file with the extension `.changes` that can be used to document the package history
- the original source archive
- optional patches which changes the original source code to fix problems regarding security, the build process, or other issues
- other files which do not fall into one of the previous categories

For existing packages, this is already the case. To build an existing package, the general procedure is as follows:

1. If you have not done so yet, set up your project as shown in *Section 4.1, "Setting Up Your Home Project"*.
2. In the terminal, choose or create a directory on a local partition that has enough space to hold the package sources.
3. Check out the project that contains the package:

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

   This creates a *PROJECT*:*PACKAGE* directory in the current directory.
4. Change the directory:

   ```
   geeko > cd PROJECT:PACKAGE
   ```
5. Decide for which build target (for example openSUSE Tumbleweed for x86_64), you want to create the RPM package and execute:

   ```
   geeko > osc build openSUSE:Tumbleweed x86_64 *.spec
   ```

6. Inspect the build process.

   Successful Build

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

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

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

   Unsuccessful Build

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

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

   A successful build ends always with the creation of the RPM and SRPM files.

7. For a detailed log, see the file `/var/tmp/build-root/openSUSE_Tumbleweed-x86_64/.build.log`.

6 Using Source Services

6.1 About Source Service

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

Source services allow:

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

For using source services you need (refer to *Example 6.1, "Structure of a `_service` File"*):

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

EXAMPLE 6.1: STRUCTURE OF A `_service` FILE

```
<services> ❶
```

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

❶ The root element of a `_service` file.

❷ The service name. The service is a script that is stored in the `/usr/lib/obs/service` directory.

❸ Mode of the service, see *Section 6.2, "Modes of Services"*.

❹ One or more parameters which are passed to the script defined in ❷.

The example above will execute the script:

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

6.2 Modes of Services

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

Default Mode

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

`trylocal` Mode

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

`localonly` Mode

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

`serveronly` Mode

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

`buildtime` Mode

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

`disabled` Mode

The disabled mode is neither running the service locally nor 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 runall`** The result will get committed as standard files again.

6.3 Defining Services for Validation

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

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

You can validate sources using either of two methods:

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

6.4 Creating Source Service Definitions

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

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

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

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

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

1. Downloads the file via the `download_files` service using the URL from the Spec file. When using osc, the downloaded file gets committed as part of the commit.
2. Compares the downloaded file (`krabber-1.0.tar.gz`) against the SHA256 checksum.
3. When **`osc service runall`** is run manually, update the source archive from an online source. In all other cases, ignore this part of the `_service` file.

6.5 Removing a Source Service

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

6.6 Trigger a service run via a webhook

You may want to update sources in Open Build Service whenever they change in a SCM system. You can create a token which allows to trigger a specific package update and use it via a web-hook. It is recommended to create a token for a specific package and not a wildcard token. Read *Book “Reference Guide”, Chapter 14 “Authorization”* to learn how to create a token.

6.6.1 Using it on gitlab

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

```
https://$your_instance/trigger/runservice
https://api.opensuse.org/trigger/runservice as real life example.
```

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

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

6.6.2 Using it on github.com

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

```
https://$your_instance/trigger/webhook?id=$TOKEN_ID.
https://api.opensuse.org/trigger/webhook?id=42 as real life example.
```

and fill the secret box with your token string. Please not that github requires that you must also define the token id as part of the webhook string. All other settings can stay untouched and just hit the "Add webhook" button.

7 Staging Workflow

7.1 Working with Staging Projects

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

7.1.1 Overview of All Staging Projects

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

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

Which will return the following XML:

```
<staging_projects>
  <staging_project name="home:Admin:Staging:A" state="empty">
    <staged_requests count="0"/>
    <untracked_requests count="0"/>
    <requests_to_review count="0"/>
    <obsolete_requests count="0"/>
    <missing_reviews count="0"/>
    <broken_packages count="0"/>
    <checks count="0"/>
    <missing_checks count="0"/>
  </staging_project>
  <staging_project name="home:Admin:Staging:B" state="empty">
    <staged_requests count="0"/>
    <untracked_requests count="0"/>
    <requests_to_review count="0"/>
    <obsolete_requests count="0"/>
    <missing_reviews count="0"/>
    <broken_packages count="0"/>
    <checks count="0"/>
    <missing_checks count="0"/>
  </staging_project>
</staging_projects>
```

7.1.2 Overview of All Staging Projects

This endpoint provides an overview of a single staging project.

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

Which will return the following XML:

```
<staging_project name="home:Admin:Staging:A" state="empty">
  <staged_requests count="0"/>
  <untracked_requests count="0"/>
  <requests_to_review count="0"/>
  <obsolete_requests count="0"/>
  <missing_reviews count="0"/>
  <broken_packages count="0"/>
  <checks count="0"/>
  <missing_checks count="0"/>
</staging_project>
```

7.1.3 Copy a Staging Project

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

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

7.2 Working with Requests

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

7.2.1 Assign Requests into a Staging Project

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

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

7.2.2 Remove Requests from a Staging Project

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

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

7.2.3 List Requests of a Staging Project

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

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

Which will return the following XML:

```
<staged_requests>
  <request id="6" creator="user_1" state="review" package="ctris"/>
</staged_requests>
```

7.2.4 Exclude Requests for a Staging Workflow

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

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

7.2.5 Remove Requests from a Staging Workflow

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

```
geeko > osc api -X DELETE '/staging/openSUSE:Factory/excluded_requests' -d
 '<requests><number>3</number><number>4</number></requests>'
```