



# Keg Reference Guide

# Keg Reference Guide

Publication Date: 01 Aug 2024

<https://documentation.suse.com> 

# Contents

## Preface **v**

### **1 Overview 1**

- 1.1 Conceptual overview 1
- 1.2 Working with keg 2

### **2 Installation 4**

- 2.1 Installation from openSUSE Cloud:Tools repository 4
- 2.2 Installation from PyPI 4

### **3 Command Line 5**

- 3.1 keg 5
  - SYNOPSIS 5 • DESCRIPTION 5 • ARGUMENTS 5 • OPTIONS 5 • EXAMPLE 7
- 3.2 generate\_recipes\_changelog 7
  - SYNOPSIS 7 • DESCRIPTION 7 • ARGUMENTS 8 • OPTIONS 8 • EXAMPLE 8

### **4 Recipes basics 10**

- 4.1 Recipes data layout 10
- 4.2 Source data format and processing 11

### **5 Image definition 12**

- 5.1 Image definition structure 12
  - image 13 • config 16 • setup 17 • archive 17
- 5.2 The `_include` statement 18
- 5.3 Additional configuration directives 19
  - include-paths 19 • image-config-comments 19 • xmlfiles 20 • schema 20

<b>6</b>	<b>Data modules</b>	<b>21</b>
6.1	Image definition modules	21
6.2	Image configuration scriptlets	23
6.3	Overlay files	24
<b>7</b>	<b>Generating change logs</b>	<b>25</b>
7.1	Source info tracking	25
7.2	Change log generator	25
7.3	Integration in OBS source service	26
<b>8</b>	<b>Keg OBS source service</b>	<b>27</b>

# Preface



## Note

Template-Based KIWI Description Builder


- [GitHub Sources \(https://github.com/SUSE-Enceladus/keg\)](https://github.com/SUSE-Enceladus/keg) ↗

# 1 Overview



## Note


### Abstract


This document provides a conceptual overview about the steps of creating an image description with `keg` which can be used to build an appliance with the `KIWI` (<https://osinside.github.io/kiwi/>)  appliance builder.



## Note


Copyright © 2022 SUSE LLC and contributors. All rights reserved.

Except where otherwise noted, this document is licensed under Creative Commons Attribution-ShareAlike 4.0 International (CC-BY-SA 4.0): <https://creativecommons.org/licenses/by-sa/4.0/legalcode> .

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

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

## 1.1 Conceptual overview

`Keg` is a tool which helps to create and manage image descriptions for use with the `KIWI` (<https://osinside.github.io/kiwi/>)  appliance builder. A `KIWI` image description consists of a single XML document that specifies type, configuration, and content of the image to build. Optionally there can be configuration scripts and overlay archives added to an image description, which allow for further configuration and additional content.

Since `KIWI` image descriptions are monolithic, maintaining a number of image descriptions that have considerable overlap with respect to content and setup can be cumbersome and error-prone. `Keg` attempts to alleviate that by allowing image descriptions to be broken into

modules. Those modules can be composed in different ways in so called image definitions, and modules can inherit from parent modules which allows for fine-tuning for specific image setups. Configuration scripts and overlay archives can also be generated in a modular fashion.

The collection of source data required for `keg` to produce image descriptions is called `recipes`. `Keg recipes` are typically kept in a `git` repository, and `keg` has support for producing change logs from `git` commit history, but this is not a requirement. A `recipes` repository provides `keg` with the information how an image description is to be composed as well as the content of the components.

The basic principle of operation is that when `keg` is executed, it is pointed to a directory within the `recipes` repository and it reads any YAML files in that directory and any parent directories and merges their contents into a dictionary. How the image definition data is structured and composed is not relevant, as long as the resulting dictionary represents a valid image definition. This allows for a lot of flexibility in the layout of a `recipes` repository. The [SUSE Public Cloud Keg Recipes repository \(https://github.com/SUSE-Enceladus/keg-recipes\)](https://github.com/SUSE-Enceladus/keg-recipes) <sup>↗</sup> provides an example of a highly modular one with strong use of inheritance.

For more details on what constitutes a `recipes` repository, see section [Chapter 4, Recipes basics](#) (ff).

## 1.2 Working with keg

To create an image description, `keg` needs to be installed, as well as `KIWI`, as the latter is used by `keg` to validate the final image description. See [Chapter 2, Installation](#) for information about how to install `keg`, and [KIWI Installation \(https://osinside.github.io/kiwi/installation.html\)](https://osinside.github.io/kiwi/installation.html) <sup>↗</sup> about how to install `KIWI`.

Additionally, a `recipes` repository is required. The following example uses the aforementioned SUSE Public Cloud `keg` recipes:

```
$ git clone https://github.com/SUSE-Enceladus/keg-recipes.git
$ mkdir sles15-sp4-byos
$ keg --recipes-root keg-recipes --dest-dir sles15-sp4-byos \
    cross-cloud/sles/byos/15-sp4
```

After the `keg` command completes the destination directory specified with `--dest-dir` contains a description for a SUSE Linux Enterprise Server 15 SP4 image for use in the Public Clouds. It can be processed with KIWI to build an image. For more details about KIWI image descriptions, see [https://osinside.github.io/kiwi/image\\_description.html](https://osinside.github.io/kiwi/image_description.html).

`Recipes` used to generate an image description can be spread over multiple repositories. For that purpose, the `--recipes-root` command line argument may be given multiple times, with each one specifying a different `recipes` repository. Repositories will be searched in the order they are specified, and for any dictionary key, config scriptlet, or overlay archive module that exists in multiple repositories, the one that is read last will be used.

Using multiple repositories for `recipes` can be useful in some situations. For example, if some parts of `recipes` data are public and some private, they can be kept in different repositories. It could also be used to base `recipes` on an upstream repository and only maintain additional image definitions or modifications in a separate repository.

`Keg` also provides support for producing image descriptions for use with the [Open Build Service](https://openbuildservice.org/help/manuals/obs-user-guide/). It can generate `_multi-build` files that are required by `OBS` for image descriptions with multiple profiles, and it comes with an `OBS Source Service` plug-in for automating generating image descriptions. See [Chapter 8, Keg OBS source service](#) for details.



## 2 Installation



### Note

This document describes how to install Keg. Currently `keg` is available from PyPi (<https://pypi.org/project/kiwi-keg/>) and from the openSUSE Build Service (<https://build.opensuse.org/package/show/Cloud:Tools/python-kiwi-keg/>) for several openSUSE distributions. It is included in openSUSE Tumbleweed.

### 2.1 Installation from openSUSE Cloud:Tools repository

`Keg` is available for various openSUSE distributions from the `Cloud:Tools` repository (<https://download.opensuse.org/repositories/Cloud:/Tools/>) in the openSUSE Build Service. Package name is `python[py_version]-kiwi-keg`.

### 2.2 Installation from PyPi

`Keg` can be obtained from the Python Package Index (PyPi) via Python's package manager `pip`:

```
$ pip install kiwi_keg
```

## 3 Command Line

### 3.1 keg

#### 3.1.1 SYNOPSIS

**keg** [*options*] <*source*>

#### 3.1.2 DESCRIPTION

keg is a tool which helps to create and manage image descriptions suitable for the KIWI (<https://osinside.github.io/kiwi/>)<sup>↗</sup> appliance builder. While keg can be used to manage a single image definition the tool provides no considerable advantage in such a use case. The primary use case for keg are situations where many image descriptions must be managed and the image descriptions have considerable overlap with respect to content and setup.

keg requires source data called recipes which provides all information necessary for keg to create KIWI image descriptions. See *Chapter 4, Recipes basics* for more information about recipes.

The recipes used for generating SUSE Public Cloud image descriptions can be found in the Public Cloud Keg Recipes (<https://github.com/SUSE-Enceladus/keg-recipes>)<sup>↗</sup> repository.

#### 3.1.3 ARGUMENTS

source

Path to image source under RECIPES\_ROOT/images

#### 3.1.4 OPTIONS

-r --recipes-root

Root directory of keg recipes. Can be used more than once. Elements from later roots may overwrite earlier one.

-d --dest-dir

Destination directory for generated description [default: .]

--disable-multibuild

Option to disable creation of OBS \_multibuild file (for image definitions with multiple profiles). [default: false]

--disable-root-tar

Option to disable the creation of root.tar.gz in destination directory. If present, an overlay tree will be created instead. [default: false]

--dump-dict

Dump generated data dictionary to stdout instead of generating an image description. Useful for debugging.

-l --list-recipes

List available images that can be created with the current recipes

-f --force

Force mode (ignore errors, overwrite files)

--format-yaml

Format/Update Keg written image description to installed KIWI schema and write the result description in YAML markup



## Note

Currently no translation of comment blocks from the Keg generated KIWI description to the YAML markup will be performed.

--format-xml

Format/Update Keg written image description to installed KIWI schema and write the result description in XML markup



## Note

Currently only top-level header comments from the Keg written image description will be preserved into the formatted/updated KIWI XML file. Inline comments will not be preserved.

-i --image-version

Set image version

-a

Generate image description for architecture ARCH (can be used multiple times)

-s --write-source-info

Write a file per profile containing a list of all used source locations. The files can be used to generate a change log from the recipes repository commit log.

-v --verbose

Enable verbose output

--version

Print version

### 3.1.5 EXAMPLE

```
git clone https://github.com/SUSE-Enceladus/keg-recipes.git
keg --recipes-root keg-recipes --dest-dir leap_description leap/jeos/15.2
```

## 3.2 generate\_recipes\_changelog

### 3.2.1 SYNOPSIS

**generate\_recipes\_changelog** [*options*] <logfile>

### 3.2.2 DESCRIPTION

`generate_recipes_changelog` generates a change log from the git commit history of one or more `keg-recipes` repositories. The input file is a source info log that is generated by `keg` when run with source tracking enabled ( `-s` command line switch).

The exit status is 0 in case a change log was generated successfully, 1 in case an error occurred, or 2 in case no error occurred but generated change log is empty.

### 3.2.3 ARGUMENTS

logfile

A source info log file produced by `keg`.

### 3.2.4 OPTIONS

`-o`

Write output to `OUTPUT_FILE` (stdout if omitted)

`-r PATH:REV`

Set git revision range to `REV` for repo at `PATH`



#### Note

This limits the applicable set of commits to the given revision spec. This can be used to select only newer changes from a previous change log generator run. See EXAMPLE below.

`-f`

Output format, 'text' or 'yaml' [default: yaml]

`-m`

Format spec for commit messages (see 'format: <string>' in 'man git-log') [default: - %s] (only used with text format)

`-t`

Use `ROOT_TAG` for yaml output (e.g. image version)

### 3.2.5 EXAMPLE

```
generate_recipes_changelog -r /path/to/repo:12345678.. -t 1.1.8 log_sources
```

This will produce a YAML change log, namespaced with `1.1.8`, intended as a version number, only considering commits after hash `12345678`. If `12345678` was the hash of the HEAD commit in the checked out branch at `/path/to/repo` at the previous run of `generate_recipes_changelog` on the same data set, this would limit the change log to only contain newer changes. This method can be used to produce incremental change logs.

The change log will have the following format:

```
1.1.8:  
- change: git message subject  
  date: git commit UTC timestamp  
  details: |-  
    git message body  
...
```

## 4 Recipes basics

To produce image descriptions, `keg` must be provided with source data, also called `keg recipes`. Unlike `KIWI` descriptions, `keg recipes` can be composed of an arbitrary number of files, which allows for creating building blocks for image descriptions. `keg` does not mandate a specific structure of the recipes data, with the exception that it expects certain types of source data in specific directories.

This document describes the fundamental `keg recipes` structure and how `keg` processes input data to generate an image definition.

### 4.1 Recipes data layout

Essentially, a `keg recipes` repository consists of three top-level directories which contain different types of configuration data. Those three are:

1. Image Definitions: `images`

The `images` directory contains all image definitions. An image definition specifies the properties and content of the image description to generate. Include statements in the image definition allow to reference chunks of content from the data modules. Image definitions are specified in YAML format, can be modular and support data inheritance. See [Chapter 5, Image definition](#) for details.

2. Data Modules: `data`

The `data` directory contains different bits of configuration and content data that can be used to compose an image description. There are three different types of data modules:

- 2.1 Image Definition Modules

Any directory in `data` that is not `file:scripts` or `file:overlayfiles` is considered a module, or module tree, for image definition data. Those modules can be referenced in the image definitions using `_include` statements. The data is in YAML format and supports inheritance.

- 2.2 Image Configuration Scriptlets

Scriptlets can be used to compose optional configuration shell scripts that `KIWI` can run during the build process. The scriptlets are located in `data/scripts`.

- 2.3 Overlay Files

Image description may include overlay files that get copied into the target image. `keg` can create overlay archives from overlay data directories. Overlay files trees are located in `data/overlayfiles`.

See *Chapter 6, Data modules* for details on data modules.

- Schema Templates: `schemas`

`keg` uses Jinja2 templates to produce the headers for `config.sh` and `images.sh`. Both are optional and `keg` will write a fallback header if they are missing. Additionally, a Jinja2 template can be used to generate `config.kiwi` instead of using the internal XML generator.

## 4.2 Source data format and processing

This section contains some general information about how `keg` handles its source data.

An image description is internally represented by a data dictionary with a certain structure. This dictionary gets composed by parsing source image definition and data files referenced by the image definition and merging them into a dictionary.

Image definitions as well as data modules are used by referencing a directory (under `images` or `data` respectively), which may be several layers of directories under the root directory. When parsing those, `keg` will also read any `.yaml` file that is in a directory above the referenced one, and merge all source data into one dictionary, with the lower (i.e. more specific) layers taking precedence over upper (i.e. more generic) ones. This inheritance mechanism is intended to reduce data duplication.

`keg` uses namespaces in the image definition to group certain bits of information (for instance, a list of packages) which can be overwritten in derived modules, allowing for creating specialized versions of data modules for specific use case or different image description versions.

Once everything is merged, the resulting dictionary is validated against the image definition schema, to ensure its structure is correct and all required keys are present. If that is the case, `keg` runs the image dictionary through its XML generator to produce a `config.kiwi` file. In case the image definition contains configuration scripts or overlay archives specifications, `keg` will generate those as well.



## 5 Image definition

In `keg` terminology, an image definition is the data set that specifies the KIWI image description that should be generated. `keg` reads image definition from the `images` directory in the `recipes` root directory.

`keg` considers all leaf directories in `images` to be image definitions. This means by parsing any YAML file from those directories and all YAML files in any parent directory and merging their data into a dictionary, a complete image definition needs to be available in the resulting dictionary. There is no specific hierarchy required in `images`. Any level of sub directories can be used to create multiple levels of inheritance, or simply just to group image definitions. Example directory layout:

```
images/  
  opensuse/  
    defaults.yaml  
    leap/  
      content.yaml  
      15.2/  
        image.yaml  
      15.3/  
        image.yaml
```

This example layout defines two images, `opensuse/leap/15.2` and `opensuse/leap/15.3`. It uses inheritance to define a common content definition for both image definitions, and to set some `opensuse` specific defaults. Running `keg -d output_dir opensuse/leap/15.3` would merge data from the following files in the show order:

```
images/opensuse/defaults.yaml  
images/opensuse/leap/content.yaml  
images/opensuse/leap/15.3/image.yaml
```

All keys from the individual YAML files that are in the given tree will be merged into a dictionary that defines the image to be generated.

### 5.1 Image definition structure

An image definition dictionary is composed of several parts that define different parts of the image. The actual image description, configuration scripts, overlay archives. All parts are defined under a top-level key in the dictionary. There are additional top-level keys that affect data parsing and generator selection.

The top-level keys are as follows:

### 5.1.1 image

The image dictionary. This is the only mandatory top-level key. It defines the content of the `config.kiwi` file `keg` should generate and is essentially a YAML version of `KIWI`'s image description (typically in XML). It contains all image configuration properties, package lists, and references to overlay archives. There is a number of special keys that influence how `keg` constructs the dictionary and generates the XML output. The basic structure is as follows:

```
image:
  _attributes:
    schemaversion: "<schema_maj>.<schema_min>"
    name: <image_name>
    displayname: <image_boot_title>
  description:
    _attributes:
      type: <system_type>
      author: <author_name>
      contact: <author_email>
  preferences:
    - version: <version_string>
    - _attributes:
        profiles:
          - <profile_name>
          ...
        type:
          _attributes:
            image: <image_type>
            kernelcmdline:
              <kernel_param>: <kernel_param_value>
            ...
          ...
        size:
          _attributes:
            unit: <size_unit>
            _text: <disk_size>
        ...
  users:
    user:
      - _attributes:
          name: <user_name>
          groups: <user_groups>
          home: <user_home>
```

```

    password: <user_password>
    ...
packages:
  - _attributes:
      type: image|bootstrap
      profiles:
        - <profile>
        ...
    archive:
      _attributes:
        name: <archive_filename>
    <namespace>:
      package:
        - _attributes:
            name: <package_name>
            arch: <package_arch>
        ...
    ...
  ...
profiles:
  profile:
    - _attributes:
        name: <profile_name>
        description: <profile_description>
    ...

```

This only outlines the structure and includes some of the configuration keys that KIWI supports. See KIWI Image Description (<https://documentation.suse.com/kiwi/9/single-html/kiwi/index.html#image-description>) for full details.

For the purpose of generating the KIWI XML image description, any key in the image dictionary that is not a plain data type is converted to an XML element in the KIWI image description, with the tag name being the key name. Any key that starts with an \_ has a special meaning. The following are supported:

#### \_attributes

If a key contains a sub key called \_attributes, it instructs the XML generator to produce an attribute for the XML element with the given key name and value as its name-value pair. If value is not a plain data type, it is converted to a string, which allows for complex attributes being split over different files and also for redefinition on lower levels. For example:

```

type:
  _attributes:
    image: vmx
    kernelcmdline:
      console: ttyS0

```

```
debug: []
```

Would generate the following XML element:

```
<type image="vmx" kernelcmdline="console=ttyS0 debug"/>
```

The empty list used as value for `debug` means the attribute parameter is valueless (i.e. a flag).

`_text`

If a key contains a key called `_text`, its value is considered the element's content string.

`_namespace[_name]`

Any key that start with `_namespace` does not produce an XML element in the output. Namespaces are used to group data and allow for an inheritance and overwrite mechanism. Namespaces produce comments in the XML output that states which namespace the enclosed data was part of.

`_map_attribute`

If a key contains a key `_map_attribute`, which needs to be a string type, any `_attribute` key under the key that is a simple list instead of the actually required mapping, is automatically converted to a mapping with the attribute key equal to `_map_attribute` value. For example:

```
packages:
  _map_attribute: name
  _namespace_some_pkgs:
    package:
      - pkg1
      - pkg2
```

Is automatically converted to:

```
packages:
  _namespace_some_pkgs:
    package:
      - _attribute:
          name: pkg1
      - _attribute:
          name: pkg1
    archive:
      - _attributes:
          name: archive1.tar.gz
```

This allows for making lists of elements that all have the same attribute (which package lists typically have) more compact and readable.

\_\_comment[\_\_name]

Any key that has a key that starts with \_\_comment will have a comment above it in the XML output, reading the value of the \_\_comment key (needs to be a string).

## 5.1.2 config

The config dictionary defines the content of the config.sh file keg should generate. config.sh is a script that KIWI runs during the image prepare step and can be used to modify the image's configuration. The config dictionary structure is as follows:

```
config:
- profiles:
  - <profile_name>
  ...
files:
  <namespace>:
  - path: <file>
    append: bool (defaults to False if missing)
    content: string
  ...
  ...
scripts:
  <namespace>:
  - <script>
  ...
  ...
services:
  <namespace>:
  - <service_name>
  - name: <service_name>
    enable: bool
  ...
  ...
sysconfig:
  <namespace>:
  - file: <sysconfig_file>
    name: <sysconfig_variable>
    value: string
  ...
  ...
...
```

Each list item in `config` produces a section in `config.sh`, with the optional `profiles` key defining for which image profile that section should apply. Each item can have the following keys (all are optional, but there has to be at least one):

`files` defines files that should be created (or overwritten if existing) with the given `content` or have `content` appended to in `config.sh`.

`scripts` defines which scriptlets should be included. `<script>` refers to a file `data/scripts/<script>.sh` in the recipes tree.

`services` defines which systemd services and timers should be enabled or disabled in the image. The short version (just a string) means the string is the service name and it should be enabled.

`vars` defines which existing sysconfig variables should be altered.



## Note

`<namespace>` defines a namespace with the same purpose as in the `image` dictionary, but `config` namespaces don't have to start with `_`, but are allowed to.

### 5.1.3 setup

The `config` dictionary defines the content of the `images.sh` file `keg` should generate. This script is run by `KIWI` during the image create step. Its structure is identical to `config`.

See [User defined scripts \(https://documentation.suse.com/kiwi/9/single-html/kiwi/index.html#working-with-kiwi-user-defined-scripts\)](https://documentation.suse.com/kiwi/9/single-html/kiwi/index.html#working-with-kiwi-user-defined-scripts) in the `KIWI` documentation for more details on user scripts.

### 5.1.4 archive

The `archive` dictionary defines the content of overlay tar archives, that can be included in the image via the `archive` sub-section of the `packages` section of the `image` dictionary. The structure is as follows:

```
archive:
- name: <archive_filename>
  <namespace>:
    _include_overlays:
      - <overlay_module>
```

```
...  
...
```

When generating the image description, `keg` will produce a tar archive for each entry in `archive` with the given file name, with its contents being composed of all files that are in the listed overlay modules. Each module references a directory in `data/overlayfiles`.

`Keg` automatically compresses the archive based on the file name extension. Supported are `gz`, `xz`, or no extension for uncompressed archive.



## Note

The archive name `root.tar` (regardless of compression extension) is automatically included in all profiles (if there are any) by `KIWI`. It is not necessary to include it explicitly in the image definition.

## 5.2 The `_include` statement

`Keg` supports importing parts of the image definition from other directory trees within the recipes to allow for modularization. For that purpose, a key in the image dictionary may have a sub-key called `_include`. Its value is a list of strings, each of which points to a directory in the `data` sub-directory of the recipes root. To process the instruction, `keg` generates another dictionary from all YAML files in the referenced directory trees (the same mechanism as when parsing the `images` tree applies). It then looks up the key in that dictionary that is equal to the parent key of the `_include` key, and replaces the `_include` key with its contents. That means, if the `_include` statement is below a key called `packages`, only data under `packages` in the include dictionary will be copied into the image definition dictionary. This allows for having different types of configuration data in the same directory and including them in different places in the image definition. See [Chapter 6, Data modules](#) for details on data modules.

## 5.3 Additional configuration directives

There are three additional optional top-level image definition sections that affect how the image definition dictionary is composed and the image description is generated:

### 5.3.1 `include-paths`

The `include-paths` key defines a list of search paths that get appended when `__include` statements are processed. This allows for having different versions of data modules and still share the most of an image definition between different versions. See [Chapter 6, Data modules](#) for details.

### 5.3.2 `image-config-comments`

This section allows to add top-level comments in the produced `KIWI` file. The format is as follows:

```
image-config-comments:  
  <comment_name>: <comment>  
  ...
```

`<comment_name>` is just a name and is not included in the generated output. Comments can be used to include arbitrary information in the image description. Some comments have a special meaning for processing image descriptions by the Open Build Service, for instance the `OBS-Profiles` directive that is required to process multi-profile image descriptions. See [https://osinside.github.io/kiwi/working\\_with\\_images/build\\_in\\_buildservice.html](https://osinside.github.io/kiwi/working_with_images/build_in_buildservice.html) for details.



#### Note

Keg generates some comments automatically. In case the image definition has multiple profiles and the `--disable-multibuild` command line switch is not set, it will add an `OBS-Profiles: @BUILD_FLAVOR@` comment. In case the image description is generated for one or more specific architectures via the `-a` command line option, the appropriate `OBS-ExclusiveArch` comment is added.



### 5.3.3 xmlfiles

This optional section allows generating additional custom XML files. The format is as follows:

```
xmlfiles:
- name: <filename>
  content:
    <content_dictionary>
  ...
```

For each list item in this section, an XML file named `<filename>` will be created, with the content being generated from the `<content_dictionary>`. For this dictionary the same rules about formatting, including, namespacing, etc., apply as for the image dictionary.

Custom XML files can be useful when generating image descriptions for use in the Open Build Service, which accepts build configuration directives via XML source files, like the `_constraints` file. See [https://openbuildservice.org/help/manuals/obs-user-guide/cha.obs.build\\_job\\_constraints.html](https://openbuildservice.org/help/manuals/obs-user-guide/cha.obs.build_job_constraints.html) for details.

### 5.3.4 schema

`keg` starting with version 2.0.0 has an internal XML generator to produce `KIWI` image descriptions. Previously, a Jinja2 template was used to convert the image dictionary that `keg` constructed into a `KIWI` image description. Using a Jinja2 template is still supported and can be configured as follows in the image definition:

```
schema: <template>
```

In this case, instead of running the XML generator, `keg` would read the file `<template>.kiwi.templ` from the `schemas` directory in the recipes root directory and run it through the Jinja2 engine.



#### Note

While using a Jinja2 template would in theory allow to operate on different input data structures, the internal schema validator requires the image definition to comply with what `keg` expects.

## 6 Data modules

Data modules are essentially directories in the `data` tree. There are three different kinds of data modules:

- Image Definition Modules

Any part of the image definition can be in a data module that is included by the `__include` statement from the main image definition.

- Image Configuration Scriptlets

Configuration scriptlets are stored in `data/scripts`. Those scriptlets can be used to compose an image configuration script or image setup script `config` and `setup` key in the image definition.

- Overlay Files

Files that can be directly included in the image description and will be copied into the image's file system by `KIWI` during the build process. Overlay files are stored under `data/overlayfiles`.

### 6.1 Image definition modules

Any directory under `data` that is not `scripts` or `overlayfiles` is considered an image definition data module and may be included in the main image definition using the `__include` statement.

Inheritance rules apply similarly to the image definition tree, but additionally, `keg` supports sub-versions of data modules. This can be used for instance to create slightly different versions of modules for use with different image versions while still sharing most of the image definition between those versions.

For this purpose, `keg` supports the `include-paths` directive in the image definition. Include paths are paths that get appended to any source path and those get scanned for input files as well. See the following image definition as an example:

```
include-paths:
  leap15/1
  leap15/2
image:
  preferences:
    - __include:
```

```

- base/common
packages:
- __include:
- base/common
config:
- __include:
- base/common

```

This tells `keg`, when adding data from directory `data/base/common` to the image data dictionary, to also look into sub directories `leap15/2`, `leap15/1`, and `leap15` (through inheritance). This would lead to the following directories being scanned:

```

data
data/common
data/common/base
data/common/base/leap15
data/common/base/leap15/1
data/common/base/leap15/2

```

This allows for example to put generic configuration bits in `data/common/base`, Leap 15 specific configuration in `data/common/base/leap15`, and adjust the configuration for minor versions, if necessary.

When merging the included dictionaries into the main dictionary, `keg` only copies the dictionary under the top level key that matches the key under which the `__include` statement is. That means, assuming the YAML files collected from the above trees resulted in the following data structure:

```

preferences:
  locale: en_US
  timezone: UTC
  type:
    __attributes:
      firmware: efi
      image: vmx
packages:
  __namespace_base_packages:
    package:
      - bash
      - glibc
      - kernel-default
config:
  __namespace_base_services:
    services:
      - sshd

```

Would result in a data structure like this:

```
include-paths:
  leap15/1
  leap15/2
image:
  preferences:
    locale: en_US
    timezone: UTC
    type:
      _attributes:
        firmware: efi
        image: vmx
  packages:
    _namespace_base_packages:
      package:
        - bash
        - glibc
        - kernel-default
config:
  _namespace_base_services:
    services:
      - sshd
```

Merging based on the parent key allows for grouping of different types of configuration data in one data module.

## 6.2 Image configuration scriptlets

Configuration scriptlets are individual script snippets that can be used to generate image configuration scripts. KIWI runs those scripts at certain points in the image build process. They can be used to do changes to the system's configuration.

The scriptlets are located in data/scripts and are required to have a .sh suffix. These are referenced in the scripts lists of the config or setup sections in the image definition (without the .sh suffix). See [Section 5.1.2, "config"](#) for details on the config section.

## 6.3 Overlay files

KIWI image descriptions can contain optional overlay archives, which will be extracted into the system's root directory before the image is created. Overlay files are located in sub-directories in `data/overlayfiles`, with each sub-directory representing an overlay files module. Any directory structure under the module's top directory is preserved.

Overlay files modules can be referenced in the `archive` section of the image definition using the `_include_overlays` directive. See [Section 5.1.4, "archive"](#) for details.

## 7 Generating change logs

`keg` comes with a separate tool that can be used to produce a change log for a generated image description from the git commit history of the used `keg_recipes` tree(s). This obviously requires these `keg_recipes` to be stored in git repositories.

To produce a change log for an image description, the description needs to be generated with source info tracking enabled in `keg` (`-s` command line switch).

### 7.1 Source info tracking

With source info tracking enabled, `keg` will write one or more source info files in addition to the image description in the output directory. In case the image description at hand is single-build, a single file `log_sources` is written, in case it is multi-build, a file `log_sources_PROFILE` is written for each profile. This allows for generating individual change logs for the resulting image binaries.

The source info logs contain detailed information about which bits from the `keg-recipes` tree was used to generate the image description. The source info log files will contain several lines of the following format:

```
root:/path/to/repository
range:start:end:/path/to/repository/file
/path/to/repository/file_or_dir
```

The first line specifies the repository location. There will be one for each `keg-recipes` directory given to `keg`. Lines starting with `range:` specify a part of a file in a repository. This is used to track the source location of each key that was in the final image dictionary. The third line format simply specifies a file or a directory in the repository that was used in the image description, and is used for configuration script snippets and overlay files.

This enables the change log generator to produce a change log using the git commit history, selecting only commits that apply to the generated image description.

### 7.2 Change log generator

The generated source info log files, together with the `keg-recipes` in the place and state they were used to generate the image description, can be used to generate change logs. The `keg` distribution contains a tool `generate_recipes_changelog` for that purpose. When called with

a source log file as argument, `generate_recipes_changelog` will use the source information to select matching git messages and produce a change log in chronological order. There are parameters to narrow down the applicable commit range as well as some formatting options. Refer to [Section 3.2, “generate\\_recipes\\_changelog”](#) command overview for details.

## 7.3 Integration in OBS source service

The `keg` distribution contains a module for integrating with the Open Build Service, an implementation of a so-called [OBS Source Service](https://openbuildservice.org/help/manuals/obs-user-guide/cha.obs.source_service.html) ([https://openbuildservice.org/help/manuals/obs-user-guide/cha.obs.source\\_service.html](https://openbuildservice.org/help/manuals/obs-user-guide/cha.obs.source_service.html))<sup>7</sup>. It supports automatic handling of change log generation. See [Chapter 8, Keg OBS source service](#) for details.

## 8 Keg OBS source service

The OBS Source Service for keg provides a mechanism to produce kiwi image descriptions for use with the Open Build Service (<https://openbuildservice.org/help/manuals/obs-user-guide/>) in an automated fashion. The OBS Source Service, named compose\_kiwi\_description, checks out any given keg-recipes repositories, runs keg to produce the specified image description, and optionally produces change log files and stores the HEAD commit hashed of the keg-recipes repositories to be used for the next source service run.

To set up an OBS package as a keg source service package, simply create a file named \_service in your package directory. The contents of the file should look like the following:

```
<services>
  <service name="compose_kiwi_description">
    <param name="git-recipes">https://github.com/SUSE-Enceladus/keg-recipes.git</
param>
    <param name="git-branch">released</param>
    <param name="image-source">cross-cloud/sles/byos/15-sp3</param>
  </service>
</services>
```

In this example, the released branch of the public keg-recipes repository for SUSE Linux Enterprise images hosted on github is used as source and the selected image source is cross-cloud/sles/byos/15-sp3. Running the source service will produce a description for a SUSE Linux Enterprise Server 15 SP3 BYOS image for several cloud service provider frameworks.

The parameters <git-recipes> and <git-branch> may be used multiple times if the image description should be composed from more than one repository.

There are a few additional optional parameters:

- arch (string)

Set build target architecture. Can be used multiple times.

- image-version (string)

Set image version. If no version is given, the version number of the existing image description will be used with the patch level increased by one.



- version-bump (true|false)

Whether the patch version number should be incremented. Ignored if `--image-version` is set. If set to `false` and `--image-version` is not set, the image version defined in the recipes will be used. If no image version is defined, image description generation will fail. Default is `true`.

- update-changelogs (true|false)

Whether `changes.yaml` files should be updated. Default is `true`.

- update-revisions (true|false)

Whether `_keg_revisions` (used for storing current commit IDs) should be updated. Default is `true`.

- force (true|false)

If true, refresh image description even if there are no new commits. Default is `false`.

The system the source service is run on needs to have `keg` and `obs-service-keg` installed. Refer to the [Using Source Services \(https://openbuildservice.org/help/manuals/obs-user-guide/cha.obs.source\\_service.html\)](https://openbuildservice.org/help/manuals/obs-user-guide/cha.obs.source_service.html) section of the OBS manual about details on how to run the source service and which operating modes are available.