GNU Guix Reference Manual

Using the GNU Guix Functional Package Manager

The GNU Guix Developers

Edition 0.13.0
1 August 2017

Table of Contents

GNU Guix

This document describes GNU Guix version 0.13.0, a functional package management tool written for the GNU system.

1 Introduction

GNU Guix[1] is a package management tool for the GNU system. Guix makes it easy for unprivileged users to install, upgrade, or remove packages, to roll back to a previous package set, to build packages from source, and generally assists with the creation and maintenance of software environments.

Guix provides a command-line package management interface (see Section 3.2 [Invoking guix package], page 18), a set of command-line utilities (see Chapter 5 [Utilities], page 64), as well as Scheme programming interfaces (see Chapter 4 [Programming Interface], page 35). Its *build daemon* is responsible for building packages on behalf of users (see Section 2.4 [Setting Up the Daemon], page 7) and for downloading pre-built binaries from authorized sources (see Section 3.3 [Substitutes], page 25).

Guix includes package definitions for many GNU and non-GNU packages, all of which respect the user's computing freedom (`https://www.gnu.org/philosophy/free-sw.html`). It is *extensible*: users can write their own package definitions (see Section 4.1 [Defining Packages], page 35) and make them available as independent package modules (see Section 6.6 [Package Modules], page 231). It is also *customizable*: users can *derive* specialized package definitions from existing ones, including from the command line (see Section 5.1.2 [Package Transformation Options], page 66).

You can install GNU Guix on top of an existing GNU/Linux system where it complements the available tools without interference (see Chapter 2 [Installation], page 3), or you can use it as part of the standalone *Guix System Distribution* or GuixSD (see Chapter 6 [GNU Distribution], page 96). With GNU GuixSD, you *declare* all aspects of the operating system configuration and Guix takes care of instantiating the configuration in a transactional, reproducible, and stateless fashion (see Section 6.2 [System Configuration], page 103).

Under the hood, Guix implements the *functional package management* discipline pioneered by Nix (see Chapter 8 [Acknowledgments], page 248). In Guix, the package build and installation process is seen as a *function*, in the mathematical sense. That function takes inputs, such as build scripts, a compiler, and libraries, and returns an installed package. As a pure function, its result depends solely on its inputs—for instance, it cannot refer to software or scripts that were not explicitly passed as inputs. A build function always produces the same result when passed a given set of inputs. It cannot alter the environment of the running system in any way; for instance, it cannot create, modify, or delete files outside of its build and installation directories. This is achieved by running build processes in isolated environments (or *containers*), where only their explicit inputs are visible.

The result of package build functions is *cached* in the file system, in a special directory called *the store* (see Section 4.3 [The Store], page 48). Each package is installed in a directory of its own in the store—by default under `/gnu/store`. The directory name contains a hash of all the inputs used to build that package; thus, changing an input yields a different directory name.

This approach is the foundation for the salient features of Guix: support for transactional package upgrade and rollback, per-user installation, and garbage collection of packages (see Section 3.1 [Features], page 17).

[1] "Guix" is pronounced like "geeks", or "iks" using the international phonetic alphabet (IPA).

2 Installation

GNU Guix is available for download from its website at `http://www.gnu.org/software/guix/`. This section describes the software requirements of Guix, as well as how to install it and get ready to use it.

Note that this section is concerned with the installation of the package manager, which can be done on top of a running GNU/Linux system. If, instead, you want to install the complete GNU operating system, see Section 6.1 [System Installation], page 96.

When installed on a running GNU/Linux system—thereafter called a *foreign distro*—GNU Guix complements the available tools without interference. Its data lives exclusively in two directories, usually `/gnu/store` and `/var/guix`; other files on your system, such as `/etc`, are left untouched.

Once installed, Guix can be updated by running `guix pull` (see Section 3.6 [Invoking guix pull], page 29).

2.1 Binary Installation

This section describes how to install Guix on an arbitrary system from a self-contained tarball providing binaries for Guix and for all its dependencies. This is often quicker than installing from source, which is described in the next sections. The only requirement is to have GNU tar and Xz.

Installing goes along these lines:

1. Download the binary tarball from '`ftp://alpha.gnu.org/gnu/guix/guix-binary-0.13.0.`*system*`.tar.`
   where *system* is `x86_64-linux` for an `x86_64` machine already running the kernel Linux, and so on.

   Make sure to download the associated `.sig` file and to verify the authenticity of the tarball against it, along these lines:

   ```
   $ wget ftp://alpha.gnu.org/gnu/guix/guix-binary-0.13.0.system.tar.xz.sig
   $ gpg --verify guix-binary-0.13.0.system.tar.xz.sig
   ```

   If that command fails because you do not have the required public key, then run this command to import it:

   ```
   $ gpg --keyserver pgp.mit.edu --recv-keys 3CE464558A84FDC69DB40CFB090B11993D9AEBB5
   ```

   and rerun the `gpg --verify` command.

2. As `root`, run:

   ```
   # cd /tmp
   # tar --warning=no-timestamp -xf \
        guix-binary-0.13.0.system.tar.xz
   # mv var/guix /var/ && mv gnu /
   ```

   This creates `/gnu/store` (see Section 4.3 [The Store], page 48) and `/var/guix`. The latter contains a ready-to-use profile for `root` (see next step.)

   Do *not* unpack the tarball on a working Guix system since that would overwrite its own essential files.

   The `--warning=no-timestamp` option makes sure GNU tar does not emit warnings about "implausibly old time stamps" (such warnings were triggered by GNU tar 1.26

and older; recent versions are fine.) They stem from the fact that all the files in the archive have their modification time set to zero (which means January 1st, 1970.) This is done on purpose to make sure the archive content is independent of its creation time, thus making it reproducible.

3. Make `root`'s profile available under `~/.guix-profile`:

```
# ln -sf /var/guix/profiles/per-user/root/guix-profile \
         ~root/.guix-profile
```

   Source `etc/profile` to augment `PATH` and other relevant environment variables:

```
# GUIX_PROFILE=$HOME/.guix-profile \
  source $GUIX_PROFILE/etc/profile
```

4. Create the group and user accounts for build users as explained below (see Section 2.4.1 [Build Environment Setup], page 7).

5. Run the daemon, and set it to automatically start on boot.

   If your host distro uses the systemd init system, this can be achieved with these commands:

```
# cp ~root/.guix-profile/lib/systemd/system/guix-daemon.service \
       /etc/systemd/system/
# systemctl start guix-daemon && systemctl enable guix-daemon
```

   If your host distro uses the Upstart init system:

```
# initctl reload-configuration
# cp ~root/.guix-profile/lib/upstart/system/guix-daemon.conf /etc/init/
# start guix-daemon
```

   Otherwise, you can still start the daemon manually with:

```
# ~root/.guix-profile/bin/guix-daemon --build-users-group=guixbuild
```

6. Make the `guix` command available to other users on the machine, for instance with:

```
# mkdir -p /usr/local/bin
# cd /usr/local/bin
# ln -s /var/guix/profiles/per-user/root/guix-profile/bin/guix
```

   It is also a good idea to make the Info version of this manual available there:

```
# mkdir -p /usr/local/share/info
# cd /usr/local/share/info
# for i in /var/guix/profiles/per-user/root/guix-profile/share/info/* ;
  do ln -s $i ; done
```

   That way, assuming `/usr/local/share/info` is in the search path, running `info guix` will open this manual (see Section "Other Info Directories" in *GNU Texinfo*, for more details on changing the Info search path.)

7. To use substitutes from `hydra.gnu.org` or one of its mirrors (see Section 3.3 [Substitutes], page 25), authorize them:

```
# guix archive --authorize < ~root/.guix-profile/share/guix/hydra.gnu.org.pub
```

8. Each user may need to perform a few additional steps to make their Guix environment ready for use, see Section 2.6 [Application Setup], page 14.

Voilà, the installation is complete!

You can confirm that Guix is working by installing a sample package into the root profile:

```
# guix package -i hello
```

The `guix` package must remain available in `root`'s profile, or it would become subject to garbage collection—in which case you would find yourself badly handicapped by the lack of the `guix` command. In other words, do not remove `guix` by running `guix package -r guix`.

The binary installation tarball can be (re)produced and verified simply by running the following command in the Guix source tree:

```
make guix-binary.system.tar.xz
```

... which, in turn, runs:

```
guix pack -s system --localstatedir guix
```

See Section 3.7 [Invoking guix pack], page 30, for more info on this handy tool.

2.2 Requirements

This section lists requirements when building Guix from source. The build procedure for Guix is the same as for other GNU software, and is not covered here. Please see the files `README` and `INSTALL` in the Guix source tree for additional details.

GNU Guix depends on the following packages:

- GNU Guile (`http://gnu.org/software/guile/`), version 2.0.9 or later, including 2.2.x;
- GNU libgcrypt (`http://gnupg.org/`);
- GnuTLS (`http://gnutls.org/`), specifically its Guile bindings (see Section "Guile Preparations" in *GnuTLS-Guile*);
- GNU Make (`http://www.gnu.org/software/make/`).

The following dependencies are optional:

- Installing Guile-JSON (`http://savannah.nongnu.org/projects/guile-json/`) will allow you to use the `guix import pypi` command (see Section 5.5 [Invoking guix import], page 73). It is of interest primarily for developers and not for casual users.
- Support for build offloading (see Section 2.4.2 [Daemon Offload Setup], page 8) and `guix copy` (see Section 5.13 [Invoking guix copy], page 93) depends on Guile-SSH (`https://github.com/artyom-poptsov/guile-ssh`), version 0.10.2 or later.
- When zlib (`http://zlib.net`) is available, `guix publish` can compress build byproducts (see Section 5.11 [Invoking guix publish], page 89).

Unless `--disable-daemon` was passed to `configure`, the following packages are also needed:

- SQLite 3 (`http://sqlite.org`);
- libbz2 (`http://www.bzip.org`);
- GCC's g++ (`http://gcc.gnu.org`), with support for the C++11 standard.

When configuring Guix on a system that already has a Guix installation, be sure to specify the same state directory as the existing installation using the `--localstatedir` option of the `configure` script (see Section "Directory Variables" in *GNU Coding Standards*).

The `configure` script protects against unintended misconfiguration of *localstatedir* so you do not inadvertently corrupt your store (see Section 4.3 [The Store], page 48).

When a working installation of the Nix package manager (`http://nixos.org/nix/`) is available, you can instead configure Guix with `--disable-daemon`. In that case, Nix replaces the three dependencies above.

Guix is compatible with Nix, so it is possible to share the same store between both. To do so, you must pass `configure` not only the same `--with-store-dir` value, but also the same `--localstatedir` value. The latter is essential because it specifies where the database that stores metadata about the store is located, among other things. The default values for Nix are `--with-store-dir=/nix/store` and `--localstatedir=/nix/var`. Note that `--disable-daemon` is not required if your goal is to share the store with Nix.

2.3 Running the Test Suite

After a successful `configure` and `make` run, it is a good idea to run the test suite. It can help catch issues with the setup or environment, or bugs in Guix itself—and really, reporting test failures is a good way to help improve the software. To run the test suite, type:

```
make check
```

Test cases can run in parallel: you can use the `-j` option of GNU make to speed things up. The first run may take a few minutes on a recent machine; subsequent runs will be faster because the store that is created for test purposes will already have various things in cache.

It is also possible to run a subset of the tests by defining the `TESTS` makefile variable as in this example:

```
make check TESTS="tests/store.scm tests/cpio.scm"
```

By default, tests results are displayed at a file level. In order to see the details of every individual test cases, it is possible to define the `SCM_LOG_DRIVER_FLAGS` makefile variable as in this example:

```
make check TESTS="tests/base64.scm" SCM_LOG_DRIVER_FLAGS="--brief=no"
```

Upon failure, please email `bug-guix@gnu.org` and attach the `test-suite.log` file. Please specify the Guix version being used as well as version numbers of the dependencies (see Section 2.2 [Requirements], page 5) in your message.

Guix also comes with a whole-system test suite that tests complete GuixSD operating system instances. It can only run on systems where Guix is already installed, using:

```
make check-system
```

or, again, by defining `TESTS` to select a subset of tests to run:

```
make check-system TESTS="basic mcron"
```

These system tests are defined in the `(gnu tests ...)` modules. They work by running the operating systems under test with lightweight instrumentation in a virtual machine (VM). They can be computationally intensive or rather cheap, depending on whether substitutes are available for their dependencies (see Section 3.3 [Substitutes], page 25). Some of them require a lot of storage space to hold VM images.

Again in case of test failures, please send `bug-guix@gnu.org` all the details.

2.4 Setting Up the Daemon

Operations such as building a package or running the garbage collector are all performed by a specialized process, the *build daemon*, on behalf of clients. Only the daemon may access the store and its associated database. Thus, any operation that manipulates the store goes through the daemon. For instance, command-line tools such as `guix package` and `guix build` communicate with the daemon (*via* remote procedure calls) to instruct it what to do.

The following sections explain how to prepare the build daemon's environment. See also Section 3.3 [Substitutes], page 25, for information on how to allow the daemon to download pre-built binaries.

2.4.1 Build Environment Setup

In a standard multi-user setup, Guix and its daemon—the `guix-daemon` program—are installed by the system administrator; `/gnu/store` is owned by `root` and `guix-daemon` runs as `root`. Unprivileged users may use Guix tools to build packages or otherwise access the store, and the daemon will do it on their behalf, ensuring that the store is kept in a consistent state, and allowing built packages to be shared among users.

When `guix-daemon` runs as `root`, you may not want package build processes themselves to run as `root` too, for obvious security reasons. To avoid that, a special pool of *build users* should be created for use by build processes started by the daemon. These build users need not have a shell and a home directory: they will just be used when the daemon drops `root` privileges in build processes. Having several such users allows the daemon to launch distinct build processes under separate UIDs, which guarantees that they do not interfere with each other—an essential feature since builds are regarded as pure functions (see Chapter 1 [Introduction], page 2).

On a GNU/Linux system, a build user pool may be created like this (using Bash syntax and the `shadow` commands):

```
# groupadd --system guixbuild
# for i in `seq -w 1 10`;
  do
    useradd -g guixbuild -G guixbuild           \
            -d /var/empty -s `which nologin`    \
            -c "Guix build user $i" --system    \
            guixbuilder$i;
  done
```

The number of build users determines how many build jobs may run in parallel, as specified by the `--max-jobs` option (see Section 2.5 [Invoking guix-daemon], page 11). To use `guix system vm` and related commands, you may need to add the build users to the `kvm` group so they can access `/dev/kvm`, using `-G guixbuild,kvm` instead of `-G guixbuild` (see Section 6.2.13 [Invoking guix system], page 214).

The `guix-daemon` program may then be run as `root` with the following command[1]:

[1] If your machine uses the systemd init system, dropping the *prefix*`/lib/systemd/system/guix-daemon.service` file in `/etc/systemd/system` will ensure that `guix-daemon` is automatically started. Similarly, if your machine uses the Upstart init system, drop the *prefix*`/lib/upstart/system/guix-daemon.conf` file in `/etc/init`.

```
# guix-daemon --build-users-group=guixbuild
```

This way, the daemon starts build processes in a chroot, under one of the `guixbuilder` users. On GNU/Linux, by default, the chroot environment contains nothing but:

- a minimal `/dev` directory, created mostly independently from the host `/dev`[2];
- the `/proc` directory; it only shows the processes of the container since a separate PID name space is used;
- `/etc/passwd` with an entry for the current user and an entry for user `nobody`;
- `/etc/group` with an entry for the user's group;
- `/etc/hosts` with an entry that maps `localhost` to `127.0.0.1`;
- a writable `/tmp` directory.

You can influence the directory where the daemon stores build trees *via* the `TMPDIR` environment variable. However, the build tree within the chroot is always called `/tmp/guix-build-`*name*`.drv-0`, where *name* is the derivation name—e.g., `coreutils-8.24`. This way, the value of `TMPDIR` does not leak inside build environments, which avoids discrepancies in cases where build processes capture the name of their build tree.

The daemon also honors the `http_proxy` environment variable for HTTP downloads it performs, be it for fixed-output derivations (see Section 4.4 [Derivations], page 50) or for substitutes (see Section 3.3 [Substitutes], page 25).

If you are installing Guix as an unprivileged user, it is still possible to run `guix-daemon` provided you pass `--disable-chroot`. However, build processes will not be isolated from one another, and not from the rest of the system. Thus, build processes may interfere with each other, and may access programs, libraries, and other files available on the system—making it much harder to view them as *pure* functions.

2.4.2 Using the Offload Facility

When desired, the build daemon can *offload* derivation builds to other machines running Guix, using the `offload` *build hook*[3]. When that feature is enabled, a list of user-specified build machines is read from `/etc/guix/machines.scm`; every time a build is requested, for instance via `guix build`, the daemon attempts to offload it to one of the machines that satisfy the constraints of the derivation, in particular its system type—e.g., `x86_64-linux`. Missing prerequisites for the build are copied over SSH to the target machine, which then proceeds with the build; upon success the output(s) of the build are copied back to the initial machine.

The `/etc/guix/machines.scm` file typically looks like this:

```
(list (build-machine
        (name "eightysix.example.org")
        (system "x86_64-linux")
        (host-key "ssh-ed25519 AAAAC3Nza...")
```

[2] "Mostly", because while the set of files that appear in the chroot's `/dev` is fixed, most of these files can only be created if the host has them.

[3] This feature is available only when Guile-SSH (`https://github.com/artyom-poptsov/guile-ssh`) is present.

```
  (user "bob")
  (speed 2.))     ;incredibly fast!

(build-machine
  (name "meeps.example.org")
  (system "mips64el-linux")
  (host-key "ssh-rsa AAAAB3Nza...")
  (user "alice")
  (private-key
   (string-append (getenv "HOME")
                  "/.ssh/identity-for-guix"))))
```

In the example above we specify a list of two build machines, one for the `x86_64` architecture and one for the `mips64el` architecture.

In fact, this file is—not surprisingly!—a Scheme file that is evaluated when the `offload` hook is started. Its return value must be a list of `build-machine` objects. While this example shows a fixed list of build machines, one could imagine, say, using DNS-SD to return a list of potential build machines discovered in the local network (see Section "Introduction" in *Using Avahi in Guile Scheme Programs*). The `build-machine` data type is detailed below.

`build-machine` [Data Type]

This data type represents build machines to which the daemon may offload builds. The important fields are:

`name` The host name of the remote machine.

`system` The system type of the remote machine—e.g., `"x86_64-linux"`.

`user` The user account to use when connecting to the remote machine over SSH. Note that the SSH key pair must *not* be passphrase-protected, to allow non-interactive logins.

`host-key` This must be the machine's SSH *public host key* in OpenSSH format. This is used to authenticate the machine when we connect to it. It is a long string that looks like this:

```
ssh-ed25519 AAAAC3NzaC...mde+UhL hint@example.org
```

If the machine is running the OpenSSH daemon, `sshd`, the host key can be found in a file such as `/etc/ssh/ssh_host_ed25519_key.pub`.

If the machine is running the SSH daemon of GNU lsh, `lshd`, the host key is in `/etc/lsh/host-key.pub` or a similar file. It can be converted to the OpenSSH format using `lsh-export-key` (see Section "Converting keys" in *LSH Manual*):

```
$ lsh-export-key --openssh < /etc/lsh/host-key.pub
ssh-rsa AAAAB3NzaC1yc2EAAAAEOp8FoQAAAQEAs1eB46LV...
```

A number of optional fields may be specified:

`port` (default: `22`)

Port number of SSH server on the machine.

`private-key` (default: `~/.ssh/id_rsa`)
: The SSH private key file to use when connecting to the machine, in OpenSSH format.

`compression` (default: `"zlib@openssh.com,zlib"`)
`compression-level` (default: `3`)
: The SSH-level compression methods and compression level requested.

  Note that offloading relies on SSH compression to reduce bandwidth usage when transferring files to and from build machines.

`daemon-socket` (default: `"/var/guix/daemon-socket/socket"`)
: File name of the Unix-domain socket `guix-daemon` is listening to on that machine.

`parallel-builds` (default: `1`)
: The number of builds that may run in parallel on the machine.

`speed` (default: `1.0`)
: A "relative speed factor". The offload scheduler will tend to prefer machines with a higher speed factor.

`features` (default: `'()`)
: A list of strings denoting specific features supported by the machine. An example is `"kvm"` for machines that have the KVM Linux modules and corresponding hardware support. Derivations can request features by name, and they will be scheduled on matching build machines.

The `guile` command must be in the search path on the build machines. In addition, the Guix modules must be in `$GUILE_LOAD_PATH` on the build machine—you can check whether this is the case by running:

```
ssh build-machine guile -c "'(use-modules (guix config))'"
```

There is one last thing to do once `machines.scm` is in place. As explained above, when offloading, files are transferred back and forth between the machine stores. For this to work, you first need to generate a key pair on each machine to allow the daemon to export signed archives of files from the store (see Section 3.8 [Invoking guix archive], page 32):

```
# guix archive --generate-key
```

Each build machine must authorize the key of the master machine so that it accepts store items it receives from the master:

```
# guix archive --authorize < master-public-key.txt
```

Likewise, the master machine must authorize the key of each build machine.

All the fuss with keys is here to express pairwise mutual trust relations between the master and the build machines. Concretely, when the master receives files from a build machine (and *vice versa*), its build daemon can make sure they are genuine, have not been tampered with, and that they are signed by an authorized key.

To test whether your setup is operational, run this command on the master node:

```
# guix offload test
```

This will attempt to connect to each of the build machines specified in `/etc/guix/machines.scm`, make sure Guile and the Guix modules are available on each

machine, attempt to export to the machine and import from it, and report any error in the process.

If you want to test a different machine file, just specify it on the command line:

```
# guix offload test machines-qualif.scm
```

Last, you can test the subset of the machines whose name matches a regular expression like this:

```
# guix offload test machines.scm '\.gnu\.org$'
```

2.5 Invoking `guix-daemon`

The `guix-daemon` program implements all the functionality to access the store. This includes launching build processes, running the garbage collector, querying the availability of a build result, etc. It is normally run as `root` like this:

```
# guix-daemon --build-users-group=guixbuild
```

For details on how to set it up, see Section 2.4 [Setting Up the Daemon], page 7.

By default, `guix-daemon` launches build processes under different UIDs, taken from the build group specified with `--build-users-group`. In addition, each build process is run in a chroot environment that only contains the subset of the store that the build process depends on, as specified by its derivation (see Chapter 4 [Programming Interface], page 35), plus a set of specific system directories. By default, the latter contains `/dev` and `/dev/pts`. Furthermore, on GNU/Linux, the build environment is a *container*: in addition to having its own file system tree, it has a separate mount name space, its own PID name space, network name space, etc. This helps achieve reproducible builds (see Section 3.1 [Features], page 17).

When the daemon performs a build on behalf of the user, it creates a build directory under `/tmp` or under the directory specified by its `TMPDIR` environment variable; this directory is shared with the container for the duration of the build. Be aware that using a directory other than `/tmp` can affect build results—for example, with a longer directory name, a build process that uses Unix-domain sockets might hit the name length limitation for `sun_path`, which it would otherwise not hit.

The build directory is automatically deleted upon completion, unless the build failed and the client specified `--keep-failed` (see Section 5.1 [Invoking guix build], page 64).

The following command-line options are supported:

`--build-users-group=`*group*
: Take users from *group* to run build processes (see Section 2.4 [Setting Up the Daemon], page 7).

`--no-substitutes`
: Do not use substitutes for build products. That is, always build things locally instead of allowing downloads of pre-built binaries (see Section 3.3 [Substitutes], page 25).

  By default substitutes are used, unless the client—such as the `guix package` command—is explicitly invoked with `--no-substitutes`.

  When the daemon runs with `--no-substitutes`, clients can still explicitly enable substitution *via* the `set-build-options` remote procedure call (see Section 4.3 [The Store], page 48).

`--substitute-urls=`*urls*
: Consider *urls* the default whitespace-separated list of substitute source URLs. When this option is omitted, '`https://mirror.hydra.gnu.org https://hydra.gnu.org`' is used (`mirror.hydra.gnu.org` is a mirror of `hydra.gnu.org`).

  This means that substitutes may be downloaded from *urls*, as long as they are signed by a trusted signature (see Section 3.3 [Substitutes], page 25).

`--no-build-hook`
: Do not use the *build hook*.

  The build hook is a helper program that the daemon can start and to which it submits build requests. This mechanism is used to offload builds to other machines (see Section 2.4.2 [Daemon Offload Setup], page 8).

`--cache-failures`
: Cache build failures. By default, only successful builds are cached.

  When this option is used, `guix gc --list-failures` can be used to query the set of store items marked as failed; `guix gc --clear-failures` removes store items from the set of cached failures. See Section 3.5 [Invoking guix gc], page 27.

`--cores=`*n*
`-c` *n*
: Use *n* CPU cores to build each derivation; `0` means as many as available.

  The default value is `0`, but it may be overridden by clients, such as the `--cores` option of `guix build` (see Section 5.1 [Invoking guix build], page 64).

  The effect is to define the `NIX_BUILD_CORES` environment variable in the build process, which can then use it to exploit internal parallelism—for instance, by running `make -j$NIX_BUILD_CORES`.

`--max-jobs=`*n*
`-M` *n*
: Allow at most *n* build jobs in parallel. The default value is `1`. Setting it to `0` means that no builds will be performed locally; instead, the daemon will offload builds (see Section 2.4.2 [Daemon Offload Setup], page 8), or simply fail.

`--rounds=`*N*
: Build each derivation *n* times in a row, and raise an error if consecutive build results are not bit-for-bit identical. Note that this setting can be overridden by clients such as `guix build` (see Section 5.1 [Invoking guix build], page 64).

  When used in conjunction with `--keep-failed`, the differing output is kept in the store, under `/gnu/store/...-check`. This makes it easy to look for differences between the two results.

`--debug`
: Produce debugging output.

  This is useful to debug daemon start-up issues, but then it may be overridden by clients, for example the `--verbosity` option of `guix build` (see Section 5.1 [Invoking guix build], page 64).

`--chroot-directory=`*dir*
: Add *dir* to the build chroot.

  Doing this may change the result of build processes—for instance if they use optional dependencies found in *dir* when it is available, and not otherwise. For

that reason, it is not recommended to do so. Instead, make sure that each derivation declares all the inputs that it needs.

`--disable-chroot`
: Disable chroot builds.

  Using this option is not recommended since, again, it would allow build processes to gain access to undeclared dependencies. It is necessary, though, when `guix-daemon` is running under an unprivileged user account.

`--disable-log-compression`
: Disable compression of the build logs.

  Unless `--lose-logs` is used, all the build logs are kept in the *localstatedir*. To save space, the daemon automatically compresses them with bzip2 by default. This option disables that.

`--disable-deduplication`
: Disable automatic file "deduplication" in the store.

  By default, files added to the store are automatically "deduplicated": if a newly added file is identical to another one found in the store, the daemon makes the new file a hard link to the other file. This can noticeably reduce disk usage, at the expense of slightly increased input/output load at the end of a build process. This option disables this optimization.

`--gc-keep-outputs[=yes|no]`
: Tell whether the garbage collector (GC) must keep outputs of live derivations.

  When set to "yes", the GC will keep the outputs of any live derivation available in the store—the `.drv` files. The default is "no", meaning that derivation outputs are kept only if they are GC roots.

`--gc-keep-derivations[=yes|no]`
: Tell whether the garbage collector (GC) must keep derivations corresponding to live outputs.

  When set to "yes", as is the case by default, the GC keeps derivations—i.e., `.drv` files—as long as at least one of their outputs is live. This allows users to keep track of the origins of items in their store. Setting it to "no" saves a bit of disk space.

  Note that when both `--gc-keep-derivations` and `--gc-keep-outputs` are used, the effect is to keep all the build prerequisites (the sources, compiler, libraries, and other build-time tools) of live objects in the store, regardless of whether these prerequisites are live. This is convenient for developers since it saves rebuilds or downloads.

`--impersonate-linux-2.6`
: On Linux-based systems, impersonate Linux 2.6. This means that the kernel's `uname` system call will report 2.6 as the release number.

  This might be helpful to build programs that (usually wrongfully) depend on the kernel version number.

`--lose-logs`
: Do not keep build logs. By default they are kept under *`localstatedir`*`/guix/log`.

`--system=`*system*
: Assume *system* as the current system type. By default it is the architecture/kernel pair found at configure time, such as `x86_64-linux`.

`--listen=`*socket*
: Listen for connections on *socket*, the file name of a Unix-domain socket. The default socket is *`localstatedir`*`/daemon-socket/socket`. This option is only useful in exceptional circumstances, such as if you need to run several daemons on the same machine.

2.6 Application Setup

When using Guix on top of GNU/Linux distribution other than GuixSD—a so-called *foreign distro*—a few additional steps are needed to get everything in place. Here are some of them.

2.6.1 Locales

Packages installed *via* Guix will not use the locale data of the host system. Instead, you must first install one of the locale packages available with Guix and then define the `GUIX_LOCPATH` environment variable:

```
$ guix package -i glibc-locales
$ export GUIX_LOCPATH=$HOME/.guix-profile/lib/locale
```

Note that the `glibc-locales` package contains data for all the locales supported by the GNU libc and weighs in at around 110 MiB. Alternatively, the `glibc-utf8-locales` is smaller but limited to a few UTF-8 locales.

The `GUIX_LOCPATH` variable plays a role similar to `LOCPATH` (see Section "Locale Names" in *The GNU C Library Reference Manual*). There are two important differences though:

1. `GUIX_LOCPATH` is honored only by the libc in Guix, and not by the libc provided by foreign distros. Thus, using `GUIX_LOCPATH` allows you to make sure the programs of the foreign distro will not end up loading incompatible locale data.
2. libc suffixes each entry of `GUIX_LOCPATH` with `/X.Y`, where `X.Y` is the libc version—e.g., `2.22`. This means that, should your Guix profile contain a mixture of programs linked against different libc version, each libc version will only try to load locale data in the right format.

This is important because the locale data format used by different libc versions may be incompatible.

2.6.2 Name Service Switch

When using Guix on a foreign distro, we *strongly recommend* that the system run the GNU C library's *name service cache daemon*, `nscd`, which should be listening on the `/var/run/nscd/socket` socket. Failing to do that, applications installed with Guix may fail to look up host names or user accounts, or may even crash. The next paragraphs explain why.

The GNU C library implements a *name service switch* (NSS), which is an extensible mechanism for "name lookups" in general: host name resolution, user accounts, and more (see Section "Name Service Switch" in *The GNU C Library Reference Manual*).

Being extensible, the NSS supports *plugins*, which provide new name lookup implementations: for example, the `nss-mdns` plugin allow resolution of `.local` host names, the `nis` plugin allows user account lookup using the Network information service (NIS), and so on. These extra "lookup services" are configured system-wide in `/etc/nsswitch.conf`, and all the programs running on the system honor those settings (see Section "NSS Configuration File" in *The GNU C Reference Manual*).

When they perform a name lookup—for instance by calling the `getaddrinfo` function in C—applications first try to connect to the nscd; on success, nscd performs name lookups on their behalf. If the nscd is not running, then they perform the name lookup by themselves, by loading the name lookup services into their own address space and running it. These name lookup services—the `libnss_*.so` files—are `dlopen`'d, but they may come from the host system's C library, rather than from the C library the application is linked against (the C library coming from Guix).

And this is where the problem is: if your application is linked against Guix's C library (say, glibc 2.24) and tries to load NSS plugins from another C library (say, `libnss_mdns.so` for glibc 2.22), it will likely crash or have its name lookups fail unexpectedly.

Running `nscd` on the system, among other advantages, eliminates this binary incompatibility problem because those `libnss_*.so` files are loaded in the `nscd` process, not in applications themselves.

2.6.3 X11 Fonts

The majority of graphical applications use Fontconfig to locate and load fonts and perform X11-client-side rendering. The `fontconfig` package in Guix looks for fonts in `$HOME/.guix-profile` by default. Thus, to allow graphical applications installed with Guix to display fonts, you have to install fonts with Guix as well. Essential font packages include `gs-fonts`, `font-dejavu`, and `font-gnu-freefont-ttf`.

To display text written in Chinese languages, Japanese, or Korean in graphical applications, consider installing `font-adobe-source-han-sans` or `font-wqy-zenhei`. The former has multiple outputs, one per language family (see Section 3.4 [Packages with Multiple Outputs], page 27). For instance, the following command installs fonts for Chinese languages:

```
guix package -i font-adobe-source-han-sans:cn
```

Older programs such as `xterm` do not use Fontconfig and instead rely on server-side font rendering. Such programs require to specify a full name of a font using XLFD (X Logical Font Description), like this:

```
-*-dejavu sans-medium-r-normal-*-*-100-*-*-*-*-*-1
```

To be able to use such full names for the TrueType fonts installed in your Guix profile, you need to extend the font path of the X server:

```
xset +fp ~/.guix-profile/share/fonts/truetype
```

After that, you can run `xlsfonts` (from `xlsfonts` package) to make sure your TrueType fonts are listed there.

2.6.4 X.509 Certificates

The `nss-certs` package provides X.509 certificates, which allow programs to authenticate Web servers accessed over HTTPS.

When using Guix on a foreign distro, you can install this package and define the relevant environment variables so that packages know where to look for certificates. See Section 6.2.9 [X.509 Certificates], page 206, for detailed information.

2.6.5 Emacs Packages

When you install Emacs packages with Guix, the elisp files may be placed either in `$HOME/.guix-profile/share/emacs/site-lisp/` or in sub-directories of `$HOME/.guix-profile/share/emacs/site-lisp/guix.d/`. The latter directory exists because potentially there may exist thousands of Emacs packages and storing all their files in a single directory may be not reliable (because of name conflicts). So we think using a separate directory for each package is a good idea. It is very similar to how the Emacs package system organizes the file structure (see Section "Package Files" in *The GNU Emacs Manual*).

By default, Emacs (installed with Guix) "knows" where these packages are placed, so you do not need to perform any configuration. If, for some reason, you want to avoid auto-loading Emacs packages installed with Guix, you can do so by running Emacs with `--no-site-file` option (see Section "Init File" in *The GNU Emacs Manual*).

3 Package Management

The purpose of GNU Guix is to allow users to easily install, upgrade, and remove software packages, without having to know about their build procedures or dependencies. Guix also goes beyond this obvious set of features.

This chapter describes the main features of Guix, as well as the package management tools it provides. Along with the command-line interface described below (see Section 3.2 [Invoking guix package], page 18), you may also use Emacs Interface (see *The Emacs-Guix Reference Manual*), after installing `emacs-guix` package (run `M-x guix-help` command to start with it):

```
guix package -i emacs-guix
```

3.1 Features

When using Guix, each package ends up in the *package store*, in its own directory—something that resembles `/gnu/store/xxx-package-1.2`, where `xxx` is a base32 string.

Instead of referring to these directories, users have their own *profile*, which points to the packages that they actually want to use. These profiles are stored within each user's home directory, at `$HOME/.guix-profile`.

For example, `alice` installs GCC 4.7.2. As a result, `/home/alice/.guix-profile/bin/gcc` points to `/gnu/store/...-gcc-4.7.2/bin/gcc`. Now, on the same machine, `bob` had already installed GCC 4.8.0. The profile of `bob` simply continues to point to `/gnu/store/...-gcc-4.8.0/bin/gcc`—i.e., both versions of GCC coexist on the same system without any interference.

The `guix package` command is the central tool to manage packages (see Section 3.2 [Invoking guix package], page 18). It operates on the per-user profiles, and can be used *with normal user privileges*.

The command provides the obvious install, remove, and upgrade operations. Each invocation is actually a *transaction*: either the specified operation succeeds, or nothing happens. Thus, if the `guix package` process is terminated during the transaction, or if a power outage occurs during the transaction, then the user's profile remains in its previous state, and remains usable.

In addition, any package transaction may be *rolled back*. So, if, for example, an upgrade installs a new version of a package that turns out to have a serious bug, users may roll back to the previous instance of their profile, which was known to work well. Similarly, the global system configuration on GuixSD is subject to transactional upgrades and roll-back (see Section 6.2.1 [Using the Configuration System], page 103).

All packages in the package store may be *garbage-collected*. Guix can determine which packages are still referenced by user profiles, and remove those that are provably no longer referenced (see Section 3.5 [Invoking guix gc], page 27). Users may also explicitly remove old generations of their profile so that the packages they refer to can be collected.

Finally, Guix takes a *purely functional* approach to package management, as described in the introduction (see Chapter 1 [Introduction], page 2). Each `/gnu/store` package directory name contains a hash of all the inputs that were used to build that package—compiler, libraries, build scripts, etc. This direct correspondence allows users to make sure a given

package installation matches the current state of their distribution. It also helps maximize *build reproducibility*: thanks to the isolated build environments that are used, a given build is likely to yield bit-identical files when performed on different machines (see Section 2.5 [Invoking guix-daemon], page 11).

This foundation allows Guix to support *transparent binary/source deployment*. When a pre-built binary for a `/gnu/store` item is available from an external source—a *substitute*, Guix just downloads it and unpacks it; otherwise, it builds the package from source, locally (see Section 3.3 [Substitutes], page 25). Because build results are usually bit-for-bit reproducible, users do not have to trust servers that provide substitutes: they can force a local build and *challenge* providers (see Section 5.12 [Invoking guix challenge], page 92).

Control over the build environment is a feature that is also useful for developers. The `guix environment` command allows developers of a package to quickly set up the right development environment for their package, without having to manually install the dependencies of the package into their profile (see Section 5.10 [Invoking guix environment], page 86).

3.2 Invoking `guix package`

The `guix package` command is the tool that allows users to install, upgrade, and remove packages, as well as rolling back to previous configurations. It operates only on the user's own profile, and works with normal user privileges (see Section 3.1 [Features], page 17). Its syntax is:

```
guix package options
```

Primarily, *options* specifies the operations to be performed during the transaction. Upon completion, a new profile is created, but previous *generations* of the profile remain available, should the user want to roll back.

For example, to remove `lua` and install `guile` and `guile-cairo` in a single transaction:

```
guix package -r lua -i guile guile-cairo
```

`guix package` also supports a *declarative approach* whereby the user specifies the exact set of packages to be available and passes it *via* the `--manifest` option (see [profile-manifest], page 20).

For each user, a symlink to the user's default profile is automatically created in `$HOME/.guix-profile`. This symlink always points to the current generation of the user's default profile. Thus, users can add `$HOME/.guix-profile/bin` to their `PATH` environment variable, and so on. If you are not using the Guix System Distribution, consider adding the following lines to your `~/.bash_profile` (see Section "Bash Startup Files" in *The GNU Bash Reference Manual*) so that newly-spawned shells get all the right environment variable definitions:

```
GUIX_PROFILE="$HOME/.guix-profile" \
source "$HOME/.guix-profile/etc/profile"
```

In a multi-user setup, user profiles are stored in a place registered as a *garbage-collector root*, which `$HOME/.guix-profile` points to (see Section 3.5 [Invoking guix gc], page 27). That directory is normally *`localstatedir`*`/profiles/per-user/`*`user`*, where *localstatedir* is the value passed to `configure` as `--localstatedir`, and *user* is the user name. The

`per-user` directory is created when `guix-daemon` is started, and the *user* sub-directory is created by `guix package`.

The *options* can be among the following:

`--install=`*package* `...`
`-i` *package* `...`
: Install the specified *package*s.

  Each *package* may specify either a simple package name, such as `guile`, or a package name followed by an at-sign and version number, such as `guile@1.8.8` or simply `guile@1.8` (in the latter case, the newest version prefixed by `1.8` is selected.)

  If no version number is specified, the newest available version will be selected. In addition, *package* may contain a colon, followed by the name of one of the outputs of the package, as in `gcc:doc` or `binutils@2.22:lib` (see Section 3.4 [Packages with Multiple Outputs], page 27). Packages with a corresponding name (and optionally version) are searched for among the GNU distribution modules (see Section 6.6 [Package Modules], page 231).

  Sometimes packages have *propagated inputs*: these are dependencies that automatically get installed along with the required package (see [package-propagated-inputs], page 39, for information about propagated inputs in package definitions).

  An example is the GNU MPC library: its C header files refer to those of the GNU MPFR library, which in turn refer to those of the GMP library. Thus, when installing MPC, the MPFR and GMP libraries also get installed in the profile; removing MPC also removes MPFR and GMP—unless they had also been explicitly installed by the user.

  Besides, packages sometimes rely on the definition of environment variables for their search paths (see explanation of `--search-paths` below). Any missing or possibly incorrect environment variable definitions are reported here.

`--install-from-expression=`*exp*
`-e` *exp*
: Install the package *exp* evaluates to.

  exp must be a Scheme expression that evaluates to a `<package>` object. This option is notably useful to disambiguate between same-named variants of a package, with expressions such as `(@ (gnu packages base) guile-final)`.

  Note that this option installs the first output of the specified package, which may be insufficient when needing a specific output of a multiple-output package.

`--install-from-file=`*file*
`-f` *file*
: Install the package that the code within *file* evaluates to.

  As an example, *file* might contain a definition like this (see Section 4.1 [Defining Packages], page 35):

```
(use-modules (guix)
             (guix build-system gnu)
             (guix licenses))
```

```
(package
  (name "hello")
  (version "2.10")
  (source (origin
            (method url-fetch)
            (uri (string-append "mirror://gnu/hello/hello-" version
                                ".tar.gz"))
            (sha256
             (base32
              "0ssi1wpaf7plaswqqjwigppsg5fyh99vdlb9kzl7c9lng89ndq1i"))))
  (build-system gnu-build-system)
  (synopsis "Hello, GNU world:  An example GNU package")
  (description "Guess what GNU Hello prints!")
  (home-page "http://www.gnu.org/software/hello/")
  (license gpl3+))
```

Developers may find it useful to include such a `guix.scm` file in the root of their project source tree that can be used to test development snapshots and create reproducible development environments (see Section 5.10 [Invoking guix environment], page 86).

`--remove=`*package* `...`
`-r` *package* `...`

Remove the specified *package*s.

As for `--install`, each *package* may specify a version number and/or output name in addition to the package name. For instance, `-r glibc:debug` would remove the `debug` output of `glibc`.

`--upgrade[=`*regexp* `...]`
`-u [`*regexp* `...]`

Upgrade all the installed packages. If one or more *regexp*s are specified, upgrade only installed packages whose name matches a *regexp*. Also see the `--do-not-upgrade` option below.

Note that this upgrades package to the latest version of packages found in the distribution currently installed. To update your distribution, you should regularly run `guix pull` (see Section 3.6 [Invoking guix pull], page 29).

`--do-not-upgrade[=`*regexp* `...]`

When used together with the `--upgrade` option, do *not* upgrade any packages whose name matches a *regexp*. For example, to upgrade all packages in the current profile except those containing the substring "emacs":

```
$ guix package --upgrade . --do-not-upgrade emacs
```

`--manifest=`*file*
`-m` *file*

Create a new generation of the profile from the manifest object returned by the Scheme code in *file*.

This allows you to *declare* the profile's contents rather than constructing it through a sequence of `--install` and similar commands. The advantage is that *file* can be put under version control, copied to different machines to reproduce the same profile, and so on.

file must return a *manifest* object, which is roughly a list of packages:

```
(use-package-modules guile emacs)

(packages->manifest
 (list emacs
       guile-2.0
       ;; Use a specific package output.
       (list guile-2.0 "debug")))
```

In this example we have to know which modules define the `emacs` and `guile-2.0` variables to provide the right `use-package-modules` line, which can be cumbersome. We can instead provide regular package specifications and let `specification->package-output` look up the corresponding package objects, like this:

```
(packages->manifest
 (map (compose list specification->package+output)
      '("emacs" "guile@2.0" "guile@2.0:debug")))
```

`--roll-back`

Roll back to the previous *generation* of the profile—i.e., undo the last transaction.

When combined with options such as `--install`, roll back occurs before any other actions.

When rolling back from the first generation that actually contains installed packages, the profile is made to point to the *zeroth generation*, which contains no files apart from its own metadata.

After having rolled back, installing, removing, or upgrading packages overwrites previous future generations. Thus, the history of the generations in a profile is always linear.

`--switch-generation=`*pattern*
`-S` *pattern*

Switch to a particular generation defined by *pattern*.

pattern may be either a generation number or a number prefixed with "+" or "-". The latter means: move forward/backward by a specified number of generations. For example, if you want to return to the latest generation after `--roll-back`, use `--switch-generation=+1`.

The difference between `--roll-back` and `--switch-generation=-1` is that `--switch-generation` will not make a zeroth generation, so if a specified generation does not exist, the current generation will not be changed.

`--search-paths[=`*kind*`]`

Report environment variable definitions, in Bash syntax, that may be needed in order to use the set of installed packages. These environment variables are used to specify *search paths* for files used by some of the installed packages.

For example, GCC needs the `CPATH` and `LIBRARY_PATH` environment variables to be defined so it can look for headers and libraries in the user's profile (see Section "Environment Variables" in *Using the GNU Compiler Collection (GCC)*). If

GCC and, say, the C library are installed in the profile, then `--search-paths` will suggest setting these variables to *profile*`/include` and *profile*`/lib`, respectively.

The typical use case is to define these environment variables in the shell:

```
$ eval `guix package --search-paths`
```

kind may be one of `exact`, `prefix`, or `suffix`, meaning that the returned environment variable definitions will either be exact settings, or prefixes or suffixes of the current value of these variables. When omitted, *kind* defaults to `exact`.

This option can also be used to compute the *combined* search paths of several profiles. Consider this example:

```
$ guix package -p foo -i guile
$ guix package -p bar -i guile-json
$ guix package -p foo -p bar --search-paths
```

The last command above reports about the `GUILE_LOAD_PATH` variable, even though, taken individually, neither `foo` nor `bar` would lead to that recommendation.

`--profile=`*profile*
`-p` *profile*
: Use *profile* instead of the user's default profile.

`--verbose`
: Produce verbose output. In particular, emit the build log of the environment on the standard error port.

`--bootstrap`
: Use the bootstrap Guile to build the profile. This option is only useful to distribution developers.

In addition to these actions, `guix package` supports the following options to query the current state of a profile, or the availability of packages:

`--search=`*regexp*
`-s` *regexp*
: List the available packages whose name, synopsis, or description matches *regexp*. Print all the metadata of matching packages in `recutils` format (see *GNU recutils manual*).

This allows specific fields to be extracted using the `recsel` command, for instance:

```
$ guix package -s malloc | recsel -p name,version
name: glibc
version: 2.17

name: libgc
version: 7.2alpha6
```

Similarly, to show the name of all the packages available under the terms of the GNU LGPL version 3:

```
$ guix package -s "" | recsel -p name -e 'license ~ "LGPL 3"'
```

```
name: elfutils

name: gmp
...
```

It is also possible to refine search results using several `-s` flags. For example, the following command returns a list of board games:

```
$ guix package -s '\<board\>' -s game | recsel -p name
name: gnubg
...
```

If we were to omit `-s game`, we would also get software packages that deal with printed circuit boards; removing the angle brackets around `board` would further add packages that have to do with keyboards.

And now for a more elaborate example. The following command searches for cryptographic libraries, filters out Haskell, Perl, Python, and Ruby libraries, and prints the name and synopsis of the matching packages:

```
$ guix package -s crypto -s library | \
    recsel -e '! (name ~ "^(ghc|perl|python|ruby)")' -p name,synopsis
```

See Section "Selection Expressions" in *GNU recutils manual*, for more information on *selection expressions* for `recsel -e`.

`--show=`*package*

Show details about *package*, taken from the list of available packages, in `recutils` format (see *GNU recutils manual*).

```
$ guix package --show=python | recsel -p name,version
name: python
version: 2.7.6

name: python
version: 3.3.5
```

You may also specify the full name of a package to only get details about a specific version of it:

```
$ guix package --show=python@3.4 | recsel -p name,version
name: python
version: 3.4.3
```

`--list-installed[=`*regexp*`]`
`-I [`*regexp*`]`

List the currently installed packages in the specified profile, with the most recently installed packages shown last. When *regexp* is specified, list only installed packages whose name matches *regexp*.

For each installed package, print the following items, separated by tabs: the package name, its version string, the part of the package that is installed (for instance, `out` for the default output, `include` for its headers, etc.), and the path of this package in the store.

`--list-available[=`*regexp*`]`
`-A [`*regexp*`]`

List packages currently available in the distribution for this system (see Chapter 6 [GNU Distribution], page 96). When *regexp* is specified, list only installed packages whose name matches *regexp*.

For each package, print the following items separated by tabs: its name, its version string, the parts of the package (see Section 3.4 [Packages with Multiple Outputs], page 27), and the source location of its definition.

`--list-generations[=`*pattern*`]`
`-l [`*pattern*`]`

Return a list of generations along with their creation dates; for each generation, show the installed packages, with the most recently installed packages shown last. Note that the zeroth generation is never shown.

For each installed package, print the following items, separated by tabs: the name of a package, its version string, the part of the package that is installed (see Section 3.4 [Packages with Multiple Outputs], page 27), and the location of this package in the store.

When *pattern* is used, the command returns only matching generations. Valid patterns include:

- *Integers and comma-separated integers.* Both patterns denote generation numbers. For instance, `--list-generations=1` returns the first one.

  And `--list-generations=1,8,2` outputs three generations in the specified order. Neither spaces nor trailing commas are allowed.

- *Ranges.* `--list-generations=2..9` prints the specified generations and everything in between. Note that the start of a range must be smaller than its end.

  It is also possible to omit the endpoint. For example, `--list-generations=2..`, returns all generations starting from the second one.

- *Durations.* You can also get the last *N* days, weeks, or months by passing an integer along with the first letter of the duration. For example, `--list-generations=20d` lists generations that are up to 20 days old.

`--delete-generations[=`*pattern*`]`
`-d [`*pattern*`]`

When *pattern* is omitted, delete all generations except the current one.

This command accepts the same patterns as `--list-generations`. When *pattern* is specified, delete the matching generations. When *pattern* specifies a duration, generations *older* than the specified duration match. For instance, `--delete-generations=1m` deletes generations that are more than one month old.

If the current generation matches, it is *not* deleted. Also, the zeroth generation is never deleted.

Note that deleting generations prevents rolling back to them. Consequently, this command must be used with care.

Finally, since `guix package` may actually start build processes, it supports all the common build options (see Section 5.1.1 [Common Build Options], page 64). It also supports package transformation options, such as `--with-source` (see Section 5.1.2 [Package Transformation Options], page 66). However, note that package transformations are lost when upgrading; to preserve transformations across upgrades, you should define your own package variant in a Guile module and add it to `GUIX_PACKAGE_PATH` (see Section 4.1 [Defining Packages], page 35).

3.3 Substitutes

Guix supports transparent source/binary deployment, which means that it can either build things locally, or download pre-built items from a server. We call these pre-built items *substitutes*—they are substitutes for local build results. In many cases, downloading a substitute is much faster than building things locally.

Substitutes can be anything resulting from a derivation build (see Section 4.4 [Derivations], page 50). Of course, in the common case, they are pre-built package binaries, but source tarballs, for instance, which also result from derivation builds, can be available as substitutes.

The `hydra.gnu.org` server is a front-end to a build farm that builds packages from the GNU distribution continuously for some architectures, and makes them available as substitutes. This is the default source of substitutes; it can be overridden by passing the `--substitute-urls` option either to `guix-daemon` (see [`guix-daemon --substitute-urls`], page 12) or to client tools such as `guix package` (see [client `--substitute-urls` option], page 65).

Substitute URLs can be either HTTP or HTTPS. HTTPS is recommended because communications are encrypted; conversely, using HTTP makes all communications visible to an eavesdropper, who could use the information gathered to determine, for instance, whether your system has unpatched security vulnerabilities.

To allow Guix to download substitutes from `hydra.gnu.org` or a mirror thereof, you must add its public key to the access control list (ACL) of archive imports, using the `guix archive` command (see Section 3.8 [Invoking guix archive], page 32). Doing so implies that you trust `hydra.gnu.org` to not be compromised and to serve genuine substitutes.

This public key is installed along with Guix, in *prefix*`/share/guix/hydra.gnu.org.pub`, where *prefix* is the installation prefix of Guix. If you installed Guix from source, make sure you checked the GPG signature of `guix-0.13.0.tar.gz`, which contains this public key file. Then, you can run something like this:

```
# guix archive --authorize < hydra.gnu.org.pub
```

Once this is in place, the output of a command like `guix build` should change from something like:

```
$ guix build emacs --dry-run
The following derivations would be built:
   /gnu/store/yr7bnx8xwcayd6j95r2clmkdl1qh688w-emacs-24.3.drv
   /gnu/store/x8qsh1hlhgjx6cwsjyvybnfv2i37z23w-dbus-1.6.4.tar.gz.drv
   /gnu/store/1ixwp12fl950d15h2cj11c73733jay0z-alsa-lib-1.0.27.1.tar.bz2.drv
   /gnu/store/nlma1pw0p603fpfiqy7kn4zm105r5dmw-util-linux-2.21.drv
```

```
...
```

to something like:

```
$ guix build emacs --dry-run
The following files would be downloaded:
   /gnu/store/pk3n22lbq6ydamyymqkkz7i69wiwjiwi-emacs-24.3
   /gnu/store/2ygn4ncnhrpr61rssa6z0d9x22si0va3-libjpeg-8d
   /gnu/store/71yz6lgx4dazma9dwn2mcjxaah9w77jq-cairo-1.12.16
   /gnu/store/7zdhgp0n1518lvfn8mb96sxqfmvqrl7v-libxrender-0.9.7
...
```

This indicates that substitutes from `hydra.gnu.org` are usable and will be downloaded, when possible, for future builds.

Guix ignores substitutes that are not signed, or that are not signed by one of the keys listed in the ACL. It also detects and raises an error when attempting to use a substitute that has been tampered with.

Substitutes are downloaded over HTTP or HTTPS. The `http_proxy` environment variable can be set in the environment of `guix-daemon` and is honored for downloads of substitutes. Note that the value of `http_proxy` in the environment where `guix build`, `guix package`, and other client commands are run has *absolutely no effect*.

When using HTTPS, the server's X.509 certificate is *not* validated (in other words, the server is not authenticated), contrary to what HTTPS clients such as Web browsers usually do. This is because Guix authenticates substitute information itself, as explained above, which is what we care about (whereas X.509 certificates are about authenticating bindings between domain names and public keys.)

The substitute mechanism can be disabled globally by running `guix-daemon` with `--no-substitutes` (see Section 2.5 [Invoking guix-daemon], page 11). It can also be disabled temporarily by passing the `--no-substitutes` option to `guix package`, `guix build`, and other command-line tools.

On Trusting Binaries

Today, each individual's control over their own computing is at the mercy of institutions, corporations, and groups with enough power and determination to subvert the computing infrastructure and exploit its weaknesses. While using `hydra.gnu.org` substitutes can be convenient, we encourage users to also build on their own, or even run their own build farm, such that `hydra.gnu.org` is less of an interesting target. One way to help is by publishing the software you build using `guix publish` so that others have one more choice of server to download substitutes from (see Section 5.11 [Invoking guix publish], page 89).

Guix has the foundations to maximize build reproducibility (see Section 3.1 [Features], page 17). In most cases, independent builds of a given package or derivation should yield bit-identical results. Thus, through a diverse set of independent package builds, we can strengthen the integrity of our systems. The `guix challenge` command aims to help users assess substitute servers, and to assist developers in finding out about non-deterministic package builds (see Section 5.12 [Invoking guix challenge], page 92). Similarly, the `--check` option of `guix build` allows users to check whether previously-installed substitutes are genuine by rebuilding them locally (see [build-check], page 69).

In the future, we want Guix to have support to publish and retrieve binaries to/from other users, in a peer-to-peer fashion. If you would like to discuss this project, join us on `guix-devel@gnu.org`.

3.4 Packages with Multiple Outputs

Often, packages defined in Guix have a single *output*—i.e., the source package leads to exactly one directory in the store. When running `guix package -i glibc`, one installs the default output of the GNU libc package; the default output is called `out`, but its name can be omitted as shown in this command. In this particular case, the default output of `glibc` contains all the C header files, shared libraries, static libraries, Info documentation, and other supporting files.

Sometimes it is more appropriate to separate the various types of files produced from a single source package into separate outputs. For instance, the GLib C library (used by GTK+ and related packages) installs more than 20 MiB of reference documentation as HTML pages. To save space for users who do not need it, the documentation goes to a separate output, called `doc`. To install the main GLib output, which contains everything but the documentation, one would run:

```
guix package -i glib
```

The command to install its documentation is:

```
guix package -i glib:doc
```

Some packages install programs with different "dependency footprints". For instance, the WordNet package installs both command-line tools and graphical user interfaces (GUIs). The former depend solely on the C library, whereas the latter depend on Tcl/Tk and the underlying X libraries. In this case, we leave the command-line tools in the default output, whereas the GUIs are in a separate output. This allows users who do not need the GUIs to save space. The `guix size` command can help find out about such situations (see Section 5.8 [Invoking guix size], page 81). `guix graph` can also be helpful (see Section 5.9 [Invoking guix graph], page 82).

There are several such multiple-output packages in the GNU distribution. Other conventional output names include `lib` for libraries and possibly header files, `bin` for stand-alone programs, and `debug` for debugging information (see Section 6.4 [Installing Debugging Files], page 228). The outputs of a packages are listed in the third column of the output of `guix package --list-available` (see Section 3.2 [Invoking guix package], page 18).

3.5 Invoking `guix gc`

Packages that are installed, but not used, may be *garbage-collected*. The `guix gc` command allows users to explicitly run the garbage collector to reclaim space from the `/gnu/store` directory. It is the *only* way to remove files from `/gnu/store`—removing files or directories manually may break it beyond repair!

The garbage collector has a set of known *roots*: any file under `/gnu/store` reachable from a root is considered *live* and cannot be deleted; any other file is considered *dead* and may be deleted. The set of garbage collector roots includes default user profiles, and may be augmented with `guix build --root`, for example (see Section 5.1 [Invoking guix build], page 64).

Prior to running `guix gc --collect-garbage` to make space, it is often useful to remove old generations from user profiles; that way, old package builds referenced by those generations can be reclaimed. This is achieved by running `guix package --delete-generations` (see Section 3.2 [Invoking guix package], page 18).

The `guix gc` command has three modes of operation: it can be used to garbage-collect any dead files (the default), to delete specific files (the `--delete` option), to print garbage-collector information, or for more advanced queries. The garbage collection options are as follows:

`--collect-garbage[=`*min*`]`
`-C [`*min*`]` Collect garbage—i.e., unreachable `/gnu/store` files and sub-directories. This is the default operation when no option is specified.

When *min* is given, stop once *min* bytes have been collected. *min* may be a number of bytes, or it may include a unit as a suffix, such as `MiB` for mebibytes and `GB` for gigabytes (see Section "Block size" in *GNU Coreutils*).

When *min* is omitted, collect all the garbage.

`--free-space=`*free*
`-F` *free* Collect garbage until *free* space is available under `/gnu/store`, if possible; *free* denotes storage space, such as `500MiB`, as described above.

When *free* or more is already available in `/gnu/store`, do nothing and exit immediately.

`--delete`
`-d` Attempt to delete all the store files and directories specified as arguments. This fails if some of the files are not in the store, or if they are still live.

`--list-failures`
List store items corresponding to cached build failures.

This prints nothing unless the daemon was started with `--cache-failures` (see Section 2.5 [Invoking guix-daemon], page 11).

`--clear-failures`
Remove the specified store items from the failed-build cache.

Again, this option only makes sense when the daemon is started with `--cache-failures`. Otherwise, it does nothing.

`--list-dead`
Show the list of dead files and directories still present in the store—i.e., files and directories no longer reachable from any root.

`--list-live`
Show the list of live store files and directories.

In addition, the references among existing store files can be queried:

`--references`
`--referrers`
List the references (respectively, the referrers) of store files given as arguments.

`--requisites`
`-R`
: List the requisites of the store files passed as arguments. Requisites include the store files themselves, their references, and the references of these, recursively. In other words, the returned list is the *transitive closure* of the store files.

  See Section 5.8 [Invoking guix size], page 81, for a tool to profile the size of the closure of an element. See Section 5.9 [Invoking guix graph], page 82, for a tool to visualize the graph of references.

Lastly, the following options allow you to check the integrity of the store and to control disk usage.

`--verify[=`*options*`]`
: Verify the integrity of the store.

  By default, make sure that all the store items marked as valid in the database of the daemon actually exist in `/gnu/store`.

  When provided, *options* must be a comma-separated list containing one or more of `contents` and `repair`.

  When passing `--verify=contents`, the daemon computes the content hash of each store item and compares it against its hash in the database. Hash mismatches are reported as data corruptions. Because it traverses *all the files in the store*, this command can take a long time, especially on systems with a slow disk drive.

  Using `--verify=repair` or `--verify=contents,repair` causes the daemon to try to repair corrupt store items by fetching substitutes for them (see Section 3.3 [Substitutes], page 25). Because repairing is not atomic, and thus potentially dangerous, it is available only to the system administrator. A lightweight alternative, when you know exactly which items in the store are corrupt, is `guix build --repair` (see Section 5.1 [Invoking guix build], page 64).

`--optimize`
: Optimize the store by hard-linking identical files—this is *deduplication*.

  The daemon performs deduplication after each successful build or archive import, unless it was started with `--disable-deduplication` (see Section 2.5 [Invoking guix-daemon], page 11). Thus, this option is primarily useful when the daemon was running with `--disable-deduplication`.

3.6 Invoking `guix pull`

Packages are installed or upgraded to the latest version available in the distribution currently available on your local machine. To update that distribution, along with the Guix tools, you must run `guix pull`: the command downloads the latest Guix source code and package descriptions, and deploys it.

On completion, `guix package` will use packages and package versions from this just-retrieved copy of Guix. Not only that, but all the Guix commands and Scheme modules will also be taken from that latest version. New `guix` sub-commands added by the update also become available.

Any user can update their Guix copy using `guix pull`, and the effect is limited to the user who run `guix pull`. For instance, when user `root` runs `guix pull`, this has no effect on the version of Guix that user `alice` sees, and vice versa[1].

The `guix pull` command is usually invoked with no arguments, but it supports the following options:

`--verbose`
: Produce verbose output, writing build logs to the standard error output.

`--url=`*url*
: Download the source tarball of Guix from *url*.

  By default, the tarball is taken from its canonical address at `gnu.org`, for the stable branch of Guix.

  With some Git servers, this can be used to deploy any version of Guix. For example, to download and deploy version 0.12.0 of Guix from the canonical Git repo:

  ```
  guix pull --url=https://git.savannah.gnu.org/cgit/guix.git/snapshot/v0.12.0.
  ```

  It can also be used to deploy arbitrary Git revisions:

  ```
  guix pull --url=https://git.savannah.gnu.org/cgit/guix.git/snapshot/74d862e8
  ```

`--bootstrap`
: Use the bootstrap Guile to build the latest Guix. This option is only useful to Guix developers.

In addition, `guix pull` supports all the common build options (see Section 5.1.1 [Common Build Options], page 64).

3.7 Invoking `guix pack`

Occasionally you want to pass software to people who are not (yet!) lucky enough to be using Guix. You'd tell them to run `guix package -i something`, but that's not possible in this case. This is where `guix pack` comes in.

The `guix pack` command creates a shrink-wrapped *pack* or *software bundle*: it creates a tarball or some other archive containing the binaries of the software you're interested in, and all its dependencies. The resulting archive can be used on any machine that does not have Guix, and people can run the exact same binaries as those you have with Guix. The pack itself is created in a bit-reproducible fashion, so anyone can verify that it really contains the build results that you pretend to be shipping.

For example, to create a bundle containing Guile, Emacs, Geiser, and all their dependencies, you can run:

```
$ guix pack guile emacs geiser
...
/gnu/store/...-pack.tar.gz
```

[1] Under the hood, `guix pull` updates the `~/.config/guix/latest` symbolic link to point to the latest Guix, and the `guix` command loads code from there. Currently, the only way to roll back an invocation of `guix pull` is to manually update this symlink to point to the previous Guix.

The result here is a tarball containing a `/gnu/store` directory with all the relevant packages. The resulting tarball contains a *profile* with the three packages of interest; the profile is the same as would be created by `guix package -i`. It is this mechanism that is used to create Guix's own standalone binary tarball (see Section 2.1 [Binary Installation], page 3).

Users of this pack would have to run `/gnu/store/...-profile/bin/guile` to run Guile, which you may find inconvenient. To work around it, you can create, say, a `/opt/gnu/bin` symlink to the profile:

```
guix pack -S /opt/gnu/bin=bin guile emacs geiser
```

That way, users can happily type `/opt/gnu/bin/guile` and enjoy.

Alternatively, you can produce a pack in the Docker image format using the following command:

```
guix pack -f docker guile emacs geiser
```

The result is a tarball that can be passed to the `docker load` command. See the Docker documentation (`https://docs.docker.com/engine/reference/commandline/load/`) for more information.

Several command-line options allow you to customize your pack:

`--format=`*format*
`-f` *format* Produce a pack in the given *format*.

The available formats are:

`tarball` This is the default format. It produces a tarball containing all the specifies binaries and symlinks.

`docker` This produces a tarball that follows the Docker Image Specification (`https://github.com/docker/docker/blob/master/image/spec/v1.2.md`).

`--expression=`*expr*
`-e` *expr* Consider the package *expr* evaluates to.

This has the same purpose as the same-named option in `guix build` (see Section 5.1.3 [Additional Build Options], page 67).

`--system=`*system*
`-s` *system* Attempt to build for *system*—e.g., `i686-linux`—instead of the system type of the build host.

`--target=`*triplet*
Cross-build for *triplet*, which must be a valid GNU triplet, such as `"mips64el-linux-gnu"` (see Section "Specifying target triplets" in *Autoconf*).

`--compression=`*tool*
`-C` *tool* Compress the resulting tarball using *tool*—one of `gzip`, `bzip2`, `xz`, or `lzip`.

`--symlink=`*spec*
`-S` *spec* Add the symlinks specified by *spec* to the pack. This option can appear several times.

spec has the form *source*`=`*target*, where *source* is the symlink that will be created and *target* is the symlink target.

For instance, `-S /opt/gnu/bin=bin` creates a `/opt/gnu/bin` symlink pointing to the `bin` sub-directory of the profile.

`--localstatedir`
: Include the "local state directory", `/var/guix`, in the resulting pack.

  `/var/guix` contains the store database (see Section 4.3 [The Store], page 48) as well as garbage-collector roots (see Section 3.5 [Invoking guix gc], page 27). Providing it in the pack means that the store is "complete" and manageable by Guix; not providing it pack means that the store is "dead": items cannot be added to it or removed from it after extraction of the pack.

  One use case for this is the Guix self-contained binary tarball (see Section 2.1 [Binary Installation], page 3).

In addition, `guix pack` supports all the common build options (see Section 5.1.1 [Common Build Options], page 64) and all the package transformation options (see Section 5.1.2 [Package Transformation Options], page 66).

3.8 Invoking `guix archive`

The `guix archive` command allows users to *export* files from the store into a single archive, and to later *import* them. In particular, it allows store files to be transferred from one machine to the store on another machine.

To export store files as an archive to standard output, run:

```
guix archive --export options specifications...
```

specifications may be either store file names or package specifications, as for `guix package` (see Section 3.2 [Invoking guix package], page 18). For instance, the following command creates an archive containing the `gui` output of the `git` package and the main output of `emacs`:

```
guix archive --export git:gui /gnu/store/...-emacs-24.3 > great.nar
```

If the specified packages are not built yet, `guix archive` automatically builds them. The build process may be controlled with the common build options (see Section 5.1.1 [Common Build Options], page 64).

To transfer the `emacs` package to a machine connected over SSH, one would run:

```
guix archive --export -r emacs | ssh the-machine guix archive --import
```

Similarly, a complete user profile may be transferred from one machine to another like this:

```
guix archive --export -r $(readlink -f ~/.guix-profile) | \
  ssh the-machine guix-archive --import
```

However, note that, in both examples, all of `emacs` and the profile as well as all of their dependencies are transferred (due to `-r`), regardless of what is already available in the store on the target machine. The `--missing` option can help figure out which items are missing from the target store. The `guix copy` command simplifies and optimizes this whole process, so this is probably what you should use in this case (see Section 5.13 [Invoking guix copy], page 93).

Archives are stored in the "normalized archive" or "nar" format, which is comparable in spirit to 'tar', but with differences that make it more appropriate for our purposes. First,

rather than recording all Unix metadata for each file, the nar format only mentions the file type (regular, directory, or symbolic link); Unix permissions and owner/group are dismissed. Second, the order in which directory entries are stored always follows the order of file names according to the C locale collation order. This makes archive production fully deterministic.

When exporting, the daemon digitally signs the contents of the archive, and that digital signature is appended. When importing, the daemon verifies the signature and rejects the import in case of an invalid signature or if the signing key is not authorized.

The main options are:

`--export` Export the specified store files or packages (see below.) Write the resulting archive to the standard output.

Dependencies are *not* included in the output, unless `--recursive` is passed.

`-r`
`--recursive`
When combined with `--export`, this instructs `guix archive` to include dependencies of the given items in the archive. Thus, the resulting archive is self-contained: it contains the closure of the exported store items.

`--import` Read an archive from the standard input, and import the files listed therein into the store. Abort if the archive has an invalid digital signature, or if it is signed by a public key not among the authorized keys (see `--authorize` below.)

`--missing`
Read a list of store file names from the standard input, one per line, and write on the standard output the subset of these files missing from the store.

`--generate-key[=`*parameters*`]`
Generate a new key pair for the daemon. This is a prerequisite before archives can be exported with `--export`. Note that this operation usually takes time, because it needs to gather enough entropy to generate the key pair.

The generated key pair is typically stored under `/etc/guix`, in `signing-key.pub` (public key) and `signing-key.sec` (private key, which must be kept secret.) When *parameters* is omitted, an ECDSA key using the Ed25519 curve is generated, or, for Libgcrypt versions before 1.6.0, it is a 4096-bit RSA key. Alternatively, *parameters* can specify `genkey` parameters suitable for Libgcrypt (see Section "General public-key related Functions" in *The Libgcrypt Reference Manual*).

`--authorize`
Authorize imports signed by the public key passed on standard input. The public key must be in "s-expression advanced format"—i.e., the same format as the `signing-key.pub` file.

The list of authorized keys is kept in the human-editable file `/etc/guix/acl`. The file contains "advanced-format s-expressions" (`http://people.csail.mit.edu/rivest/Sexp.txt`) and is structured as an access-control list in the Simple Public-Key Infrastructure (SPKI) (`http://theworld.com/~cme/spki.txt`).

`--extract=`*directory*
`-x` *directory*

Read a single-item archive as served by substitute servers (see Section 3.3 [Substitutes], page 25) and extract it to *directory*. This is a low-level operation needed in only very narrow use cases; see below.

For example, the following command extracts the substitute for Emacs served by `hydra.gnu.org` to `/tmp/emacs`:

```
$ wget -O - \
  https://hydra.gnu.org/nar/...-emacs-24.5 \
  | bunzip2 | guix archive -x /tmp/emacs
```

Single-item archives are different from multiple-item archives produced by `guix archive --export`; they contain a single store item, and they do *not* embed a signature. Thus this operation does *no* signature verification and its output should be considered unsafe.

The primary purpose of this operation is to facilitate inspection of archive contents coming from possibly untrusted substitute servers.

4 Programming Interface

GNU Guix provides several Scheme programming interfaces (APIs) to define, build, and query packages. The first interface allows users to write high-level package definitions. These definitions refer to familiar packaging concepts, such as the name and version of a package, its build system, and its dependencies. These definitions can then be turned into concrete build actions.

Build actions are performed by the Guix daemon, on behalf of users. In a standard setup, the daemon has write access to the store—the `/gnu/store` directory—whereas users do not. The recommended setup also has the daemon perform builds in chroots, under a specific build users, to minimize interference with the rest of the system.

Lower-level APIs are available to interact with the daemon and the store. To instruct the daemon to perform a build action, users actually provide it with a *derivation*. A derivation is a low-level representation of the build actions to be taken, and the environment in which they should occur—derivations are to package definitions what assembly is to C programs. The term "derivation" comes from the fact that build results *derive* from them.

This chapter describes all these APIs in turn, starting from high-level package definitions.

4.1 Defining Packages

The high-level interface to package definitions is implemented in the `(guix packages)` and `(guix build-system)` modules. As an example, the package definition, or *recipe*, for the GNU Hello package looks like this:

```
(define-module (gnu packages hello)
  #:use-module (guix packages)
  #:use-module (guix download)
  #:use-module (guix build-system gnu)
  #:use-module (guix licenses)
  #:use-module (gnu packages gawk))

(define-public hello
  (package
    (name "hello")
    (version "2.10")
    (source (origin
              (method url-fetch)
              (uri (string-append "mirror://gnu/hello/hello-" version
                                  ".tar.gz"))
              (sha256
               (base32
                "0ssi1wpaf7plaswqqjwigppsg5fyh99vdlb9kzl7c9lng89ndq1i"))))
    (build-system gnu-build-system)
    (arguments '(#:configure-flags '("--enable-silent-rules")))
    (inputs `(("gawk" ,gawk)))
    (synopsis "Hello, GNU world: An example GNU package")
    (description "Guess what GNU Hello prints!")
```

```
        (home-page "http://www.gnu.org/software/hello/")
        (license gpl3+)))
```

Without being a Scheme expert, the reader may have guessed the meaning of the various fields here. This expression binds the variable `hello` to a `<package>` object, which is essentially a record (see Section "SRFI-9" in *GNU Guile Reference Manual*). This package object can be inspected using procedures found in the `(guix packages)` module; for instance, `(package-name hello)` returns—surprise!—`"hello"`.

With luck, you may be able to import part or all of the definition of the package you are interested in from another repository, using the `guix import` command (see Section 5.5 [Invoking guix import], page 73).

In the example above, *hello* is defined in a module of its own, `(gnu packages hello)`. Technically, this is not strictly necessary, but it is convenient to do so: all the packages defined in modules under `(gnu packages ...)` are automatically known to the command-line tools (see Section 6.6 [Package Modules], page 231).

There are a few points worth noting in the above package definition:

- The `source` field of the package is an `<origin>` object (see Section 4.1.2 [origin Reference], page 40, for the complete reference). Here, the `url-fetch` method from `(guix download)` is used, meaning that the source is a file to be downloaded over FTP or HTTP.

  The `mirror://gnu` prefix instructs `url-fetch` to use one of the GNU mirrors defined in `(guix download)`.

  The `sha256` field specifies the expected SHA256 hash of the file being downloaded. It is mandatory, and allows Guix to check the integrity of the file. The `(base32 ...)` form introduces the base32 representation of the hash. You can obtain this information with `guix download` (see Section 5.3 [Invoking guix download], page 72) and `guix hash` (see Section 5.4 [Invoking guix hash], page 72).

  When needed, the `origin` form can also have a `patches` field listing patches to be applied, and a `snippet` field giving a Scheme expression to modify the source code.

- The `build-system` field specifies the procedure to build the package (see Section 4.2 [Build Systems], page 41). Here, *gnu-build-system* represents the familiar GNU Build System, where packages may be configured, built, and installed with the usual `./configure && make && make check && make install` command sequence.

- The `arguments` field specifies options for the build system (see Section 4.2 [Build Systems], page 41). Here it is interpreted by *gnu-build-system* as a request run `configure` with the `--enable-silent-rules` flag.

  What about these quote (`'`) characters? They are Scheme syntax to introduce a literal list; `'` is synonymous with `quote`. See Section "Expression Syntax" in *GNU Guile Reference Manual*, for details. Here the value of the `arguments` field is a list of arguments passed to the build system down the road, as with `apply` (see Section "Fly Evaluation" in *GNU Guile Reference Manual*).

  The hash-colon (`#:`) sequence defines a Scheme *keyword* (see Section "Keywords" in *GNU Guile Reference Manual*), and `#:configure-flags` is a keyword used to pass a keyword argument to the build system (see Section "Coding With Keywords" in *GNU Guile Reference Manual*).

- The `inputs` field specifies inputs to the build process—i.e., build-time or run-time dependencies of the package. Here, we define an input called `"gawk"` whose value is that of the *gawk* variable; *gawk* is itself bound to a `<package>` object.

  Again, ‘ (a backquote, synonymous with `quasiquote`) allows us to introduce a literal list in the `inputs` field, while `,` (a comma, synonymous with `unquote`) allows us to insert a value in that list (see Section “Expression Syntax” in *GNU Guile Reference Manual*).

  Note that GCC, Coreutils, Bash, and other essential tools do not need to be specified as inputs here. Instead, *gnu-build-system* takes care of ensuring that they are present (see Section 4.2 [Build Systems], page 41).

  However, any other dependencies need to be specified in the `inputs` field. Any dependency not specified here will simply be unavailable to the build process, possibly leading to a build failure.

See Section 4.1.1 [package Reference], page 38, for a full description of possible fields.

Once a package definition is in place, the package may actually be built using the `guix build` command-line tool (see Section 5.1 [Invoking guix build], page 64), troubleshooting any build failures you encounter (see Section 5.1.4 [Debugging Build Failures], page 70). You can easily jump back to the package definition using the `guix edit` command (see Section 5.2 [Invoking guix edit], page 71). See Section 6.7 [Packaging Guidelines], page 231, for more information on how to test package definitions, and Section 5.7 [Invoking guix lint], page 80, for information on how to check a definition for style conformance. Lastly, see Section 6.6 [Package Modules], page 231, for information on how to extend the distribution by adding your own package definitions to `GUIX_PACKAGE_PATH`.

Finally, updating the package definition to a new upstream version can be partly automated by the `guix refresh` command (see Section 5.6 [Invoking guix refresh], page 77).

Behind the scenes, a derivation corresponding to the `<package>` object is first computed by the `package-derivation` procedure. That derivation is stored in a `.drv` file under `/gnu/store`. The build actions it prescribes may then be realized by using the `build-derivations` procedure (see Section 4.3 [The Store], page 48).

`package-derivation` *store package* [*system*] [Scheme Procedure]

Return the `<derivation>` object of *package* for *system* (see Section 4.4 [Derivations], page 50).

package must be a valid `<package>` object, and *system* must be a string denoting the target system type—e.g., `"x86_64-linux"` for an x86_64 Linux-based GNU system. *store* must be a connection to the daemon, which operates on the store (see Section 4.3 [The Store], page 48).

Similarly, it is possible to compute a derivation that cross-builds a package for some other system:

`package-cross-derivation` *store package target* [*system*] [Scheme Procedure]

Return the `<derivation>` object of *package* cross-built from *system* to *target*.

target must be a valid GNU triplet denoting the target hardware and operating system, such as `"mips64el-linux-gnu"` (see Section “Configuration Names” in *GNU Configure and Build System*).

Packages can be manipulated in arbitrary ways. An example of a useful transformation is *input rewriting*, whereby the dependency tree of a package is rewritten by replacing specific inputs by others:

`package-input-rewriting` *replacements* [*rewrite-name*] [Scheme Procedure]
Return a procedure that, when passed a package, replaces its direct and indirect dependencies (but not its implicit inputs) according to *replacements*. *replacements* is a list of package pairs; the first element of each pair is the package to replace, and the second one is the replacement.

Optionally, *rewrite-name* is a one-argument procedure that takes the name of a package and returns its new name after rewrite.

Consider this example:

```
(define libressl-instead-of-openssl
  ;; This is a procedure to replace OPENSSL by LIBRESSL,
  ;; recursively.
  (package-input-rewriting `((,openssl . ,libressl))))

(define git-with-libressl
  (libressl-instead-of-openssl git))
```

Here we first define a rewriting procedure that replaces *openssl* with *libressl*. Then we use it to define a *variant* of the *git* package that uses *libressl* instead of *openssl*. This is exactly what the `--with-input` command-line option does (see Section 5.1.2 [Package Transformation Options], page 66).

A more generic procedure to rewrite a package dependency graph is `package-mapping`: it supports arbitrary changes to nodes in the graph.

`package-mapping` *proc* [*cut?*] [Scheme Procedure]
Return a procedure that, given a package, applies *proc* to all the packages depended on and returns the resulting package. The procedure stops recursion when *cut?* returns true for a given package.

4.1.1 `package` Reference

This section summarizes all the options available in `package` declarations (see Section 4.1 [Defining Packages], page 35).

`package` [Data Type]
This is the data type representing a package recipe.

`name` The name of the package, as a string.

`version` The version of the package, as a string.

`source` An object telling how the source code for the package should be acquired. Most of the time, this is an `origin` object, which denotes a file fetched from the Internet (see Section 4.1.2 [origin Reference], page 40). It can also be any other "file-like" object such as a `local-file`, which denotes a file from the local file system (see Section 4.6 [G-Expressions], page 56).

`build-system`
: The build system that should be used to build the package (see Section 4.2 [Build Systems], page 41).

`arguments` (default: `'()`)
: The arguments that should be passed to the build system. This is a list, typically containing sequential keyword-value pairs.

`inputs` (default: `'()`)
`native-inputs` (default: `'()`)
`propagated-inputs` (default: `'()`)
: These fields list dependencies of the package. Each one is a list of tuples, where each tuple has a label for the input (a string) as its first element, a package, origin, or derivation as its second element, and optionally the name of the output thereof that should be used, which defaults to `"out"` (see Section 3.4 [Packages with Multiple Outputs], page 27, for more on package outputs). For example, the list below specifies three inputs:

```
`(("libffi" ,libffi)
  ("libunistring" ,libunistring)
  ("glib:bin" ,glib "bin"))  ;the "bin" output of Glib
```

The distinction between `native-inputs` and `inputs` is necessary when considering cross-compilation. When cross-compiling, dependencies listed in `inputs` are built for the *target* architecture; conversely, dependencies listed in `native-inputs` are built for the architecture of the *build* machine.

`native-inputs` is typically used to list tools needed at build time, but not at run time, such as Autoconf, Automake, pkg-config, Gettext, or Bison. `guix lint` can report likely mistakes in this area (see Section 5.7 [Invoking guix lint], page 80).

Lastly, `propagated-inputs` is similar to `inputs`, but the specified packages will be automatically installed alongside the package they belong to (see [package-cmd-propagated-inputs], page 19, for information on how `guix package` deals with propagated inputs.)

For example this is necessary when a C/C++ library needs headers of another library to compile, or when a pkg-config file refers to another one *via* its `Requires` field.

Another example where `propagated-inputs` is useful is for languages that lack a facility to record the run-time search path akin to the `RUNPATH` of ELF files; this includes Guile, Python, Perl, and more. To ensure that libraries written in those languages can find library code they depend on at run time, run-time dependencies must be listed in `propagated-inputs` rather than `inputs`.

`self-native-input?` (default: `#f`)
: This is a Boolean field telling whether the package should use itself as a native input when cross-compiling.

`outputs` (default: `'("out")`)
: The list of output names of the package. See Section 3.4 [Packages with Multiple Outputs], page 27, for typical uses of additional outputs.

`native-search-paths` (default: `'()`)
`search-paths` (default: `'()`)
: A list of `search-path-specification` objects describing search-path environment variables honored by the package.

`replacement` (default: `#f`)
: This must be either `#f` or a package object that will be used as a *replacement* for this package. See Section 6.5 [Security Updates], page 229, for details.

`synopsis` A one-line description of the package.

`description`
: A more elaborate description of the package.

`license` The license of the package; a value from `(guix licenses)`, or a list of such values.

`home-page`
: The URL to the home-page of the package, as a string.

`supported-systems` (default: *%supported-systems*)
: The list of systems supported by the package, as strings of the form `architecture-kernel`, for example `"x86_64-linux"`.

`maintainers` (default: `'()`)
: The list of maintainers of the package, as `maintainer` objects.

`location` (default: source location of the `package` form)
: The source location of the package. It is useful to override this when inheriting from another package, in which case this field is not automatically corrected.

4.1.2 `origin` Reference

This section summarizes all the options available in `origin` declarations (see Section 4.1 [Defining Packages], page 35).

`origin` [Data Type]

This is the data type representing a source code origin.

`uri` An object containing the URI of the source. The object type depends on the `method` (see below). For example, when using the *url-fetch* method of `(guix download)`, the valid `uri` values are: a URL represented as a string, or a list thereof.

`method` A procedure that handles the URI.

Examples include:

url-fetch from `(guix download)`
: download a file from the HTTP, HTTPS, or FTP URL specified in the `uri` field;

git-fetch from `(guix git-download)`
clone the Git version control repository, and check out the revision specified in the `uri` field as a `git-reference` object; a `git-reference` looks like this:

```
(git-reference
  (url "git://git.debian.org/git/pkg-shadow/shadow")
  (commit "v4.1.5.1"))
```

`sha256` A bytevector containing the SHA-256 hash of the source. Typically the `base32` form is used here to generate the bytevector from a base-32 string.

You can obtain this information using `guix download` (see Section 5.3 [Invoking guix download], page 72) or `guix hash` (see Section 5.4 [Invoking guix hash], page 72).

`file-name` (default: `#f`)
The file name under which the source code should be saved. When this is `#f`, a sensible default value will be used in most cases. In case the source is fetched from a URL, the file name from the URL will be used. For version control checkouts, it is recommended to provide the file name explicitly because the default is not very descriptive.

`patches` (default: `'()`)
A list of file names containing patches to be applied to the source.

This list of patches must be unconditional. In particular, it cannot depend on the value of `%current-system` or `%current-target-system`.

`snippet` (default: `#f`)
A G-expression (see Section 4.6 [G-Expressions], page 56) or S-expression that will be run in the source directory. This is a convenient way to modify the source, sometimes more convenient than a patch.

`patch-flags` (default: `'("-p1")`)
A list of command-line flags that should be passed to the `patch` command.

`patch-inputs` (default: `#f`)
Input packages or derivations to the patching process. When this is `#f`, the usual set of inputs necessary for patching are provided, such as GNU Patch.

`modules` (default: `'()`)
A list of Guile modules that should be loaded during the patching process and while running the code in the `snippet` field.

`patch-guile` (default: `#f`)
The Guile package that should be used in the patching process. When this is `#f`, a sensible default is used.

4.2 Build Systems

Each package definition specifies a *build system* and arguments for that build system (see Section 4.1 [Defining Packages], page 35). This `build-system` field represents the build procedure of the package, as well as implicit dependencies of that build procedure.

Build systems are `<build-system>` objects. The interface to create and manipulate them is provided by the `(guix build-system)` module, and actual build systems are exported by specific modules.

Under the hood, build systems first compile package objects to *bags*. A *bag* is like a package, but with less ornamentation—in other words, a bag is a lower-level representation of a package, which includes all the inputs of that package, including some that were implicitly added by the build system. This intermediate representation is then compiled to a derivation (see Section 4.4 [Derivations], page 50).

Build systems accept an optional list of *arguments*. In package definitions, these are passed *via* the `arguments` field (see Section 4.1 [Defining Packages], page 35). They are typically keyword arguments (see Section "Optional Arguments" in *GNU Guile Reference Manual*). The value of these arguments is usually evaluated in the *build stratum*—i.e., by a Guile process launched by the daemon (see Section 4.4 [Derivations], page 50).

The main build system is *gnu-build-system*, which implements the standard build procedure for GNU and many other packages. It is provided by the `(guix build-system gnu)` module.

`gnu-build-system` [Scheme Variable]

gnu-build-system represents the GNU Build System, and variants thereof (see Section "Configuration" in *GNU Coding Standards*).

In a nutshell, packages using it are configured, built, and installed with the usual `./configure && make && make check && make install` command sequence. In practice, a few additional steps are often needed. All these steps are split up in separate *phases*, notably[1]:

`unpack` Unpack the source tarball, and change the current directory to the extracted source tree. If the source is actually a directory, copy it to the build tree, and enter that directory.

`patch-source-shebangs`
Patch shebangs encountered in source files so they refer to the right store file names. For instance, this changes `#!/bin/sh` to `#!/gnu/store/...-bash-4.3/bin/sh`.

`configure`
Run the `configure` script with a number of default options, such as `--prefix=/gnu/store/...`, as well as the options specified by the `#:configure-flags` argument.

`build` Run `make` with the list of flags specified with `#:make-flags`. If the `#:parallel-build?` argument is true (the default), build with `make -j`.

`check` Run `make check`, or some other target specified with `#:test-target`, unless `#:tests? #f` is passed. If the `#:parallel-tests?` argument is true (the default), run `make check -j`.

`install` Run `make install` with the flags listed in `#:make-flags`.

[1] Please see the `(guix build gnu-build-system)` modules for more details about the build phases.

`patch-shebangs`
: Patch shebangs on the installed executable files.

`strip`
: Strip debugging symbols from ELF files (unless `#:strip-binaries?` is false), copying them to the `debug` output when available (see Section 6.4 [Installing Debugging Files], page 228).

The build-side module `(guix build gnu-build-system)` defines *%standard-phases* as the default list of build phases. *%standard-phases* is a list of symbol/procedure pairs, where the procedure implements the actual phase.

The list of phases used for a particular package can be changed with the `#:phases` parameter. For instance, passing:

```
#:phases (modify-phases %standard-phases (delete 'configure))
```

means that all the phases described above will be used, except the `configure` phase.

In addition, this build system ensures that the "standard" environment for GNU packages is available. This includes tools such as GCC, libc, Coreutils, Bash, Make, Diffutils, grep, and sed (see the `(guix build-system gnu)` module for a complete list). We call these the *implicit inputs* of a package, because package definitions do not have to mention them.

Other `<build-system>` objects are defined to support other conventions and tools used by free software packages. They inherit most of *gnu-build-system*, and differ mainly in the set of inputs implicitly added to the build process, and in the list of phases executed. Some of these build systems are listed below.

`ant-build-system` [Scheme Variable]
: This variable is exported by `(guix build-system ant)`. It implements the build procedure for Java packages that can be built with Ant build tool (`http://ant.apache.org/`).

  It adds both `ant` and the *Java Development Kit* (JDK) as provided by the `icedtea` package to the set of inputs. Different packages can be specified with the `#:ant` and `#:jdk` parameters, respectively.

  When the original package does not provide a suitable Ant build file, the parameter `#:jar-name` can be used to generate a minimal Ant build file `build.xml` with tasks to build the specified jar archive. In this case the parameter `#:source-dir` can be used to specify the source sub-directory, defaulting to "src".

  The parameter `#:build-target` can be used to specify the Ant task that should be run during the `build` phase. By default the "jar" task will be run.

`asdf-build-system/source` [Scheme Variable]
`asdf-build-system/sbcl` [Scheme Variable]
`asdf-build-system/ecl` [Scheme Variable]
: These variables, exported by `(guix build-system asdf)`, implement build procedures for Common Lisp packages using "ASDF" (`https://common-lisp.net/project/asdf/`). ASDF is a system definition facility for Common Lisp programs and libraries.

  The `asdf-build-system/source` system installs the packages in source form, and can be loaded using any common lisp implementation, via ASDF. The others, such

as `asdf-build-system/sbcl`, install binary systems in the format which a particular implementation understands. These build systems can also be used to produce executable programs, or lisp images which contain a set of packages pre-loaded.

The build system uses naming conventions. For binary packages, the package itself as well as its run-time dependencies should begin their name with the lisp implementation, such as `sbcl-` for `asdf-build-system/sbcl`. Beginning the input name with this prefix will allow the build system to encode its location into the resulting library, so that the input can be found at run-time.

If dependencies are used only for tests, it is convenient to use a different prefix in order to avoid having a run-time dependency on such systems. For example,

```
(define-public sbcl-bordeaux-threads
  (package
    ...
    (native-inputs `(("tests:cl-fiveam" ,sbcl-fiveam)))
    ...))
```

Additionally, the corresponding source package should be labeled using the same convention as python packages (see Section 6.7.5 [Python Modules], page 235), using the `cl-` prefix.

For binary packages, each system should be defined as a Guix package. If one package `origin` contains several systems, package variants can be created in order to build all the systems. Source packages, which use `asdf-build-system/source`, may contain several systems.

In order to create executable programs and images, the build-side procedures `build-program` and `build-image` can be used. They should be called in a build phase after the `create-symlinks` phase, so that the system which was just built can be used within the resulting image. `build-program` requires a list of Common Lisp expressions to be passed as the `#:entry-program` argument.

If the system is not defined within its own `.asd` file of the same name, then the `#:asd-file` parameter should be used to specify which file the system is defined in.

`cargo-build-system` [Scheme Variable]

This variable is exported by `(guix build-system cargo)`. It supports builds of packages using Cargo, the build tool of the Rust programming language (`https://www.rust-lang.org`).

In its `configure` phase, this build system replaces dependencies specified in the `Carto.toml` file with inputs to the Guix package. The `install` phase installs the binaries, and it also installs the source code and `Cargo.toml` file.

`cmake-build-system` [Scheme Variable]

This variable is exported by `(guix build-system cmake)`. It implements the build procedure for packages using the CMake build tool (`http://www.cmake.org`).

It automatically adds the `cmake` package to the set of inputs. Which package is used can be specified with the `#:cmake` parameter.

The `#:configure-flags` parameter is taken as a list of flags passed to the `cmake` command. The `#:build-type` parameter specifies in abstract terms the flags passed

to the compiler; it defaults to `"RelWithDebInfo"` (short for "release mode with debugging information"), which roughly means that code is compiled with `-O2 -g`, as is the case for Autoconf-based packages by default.

`glib-or-gtk-build-system` [Scheme Variable]

This variable is exported by `(guix build-system glib-or-gtk)`. It is intended for use with packages making use of GLib or GTK+.

This build system adds the following two phases to the ones defined by *gnu-build-system*:

`glib-or-gtk-wrap`

The phase `glib-or-gtk-wrap` ensures that programs in `bin/` are able to find GLib "schemas" and GTK+ modules (`https://developer.gnome.org/gtk3/stable/gtk-running.html`). This is achieved by wrapping the programs in launch scripts that appropriately set the `XDG_DATA_DIRS` and `GTK_PATH` environment variables.

It is possible to exclude specific package outputs from that wrapping process by listing their names in the `#:glib-or-gtk-wrap-excluded-outputs` parameter. This is useful when an output is known not to contain any GLib or GTK+ binaries, and where wrapping would gratuitously add a dependency of that output on GLib and GTK+.

`glib-or-gtk-compile-schemas`

The phase `glib-or-gtk-compile-schemas` makes sure that all GSettings schemas (`https://developer.gnome.org/gio/stable/glib-compile-schemas.html`) of GLib are compiled. Compilation is performed by the `glib-compile-schemas` program. It is provided by the package `glib:bin` which is automatically imported by the build system. The `glib` package providing `glib-compile-schemas` can be specified with the `#:glib` parameter.

Both phases are executed after the `install` phase.

`ocaml-build-system` [Scheme Variable]

This variable is exported by `(guix build-system ocaml)`. It implements a build procedure for OCaml (`https://ocaml.org`) packages, which consists of choosing the correct set of commands to run for each package. OCaml packages can expect many different commands to be run. This build system will try some of them.

When the package has a `setup.ml` file present at the top-level, it will run `ocaml setup.ml -configure`, `ocaml setup.ml -build` and `ocaml setup.ml -install`. The build system will assume that this file was generated by OASIS (`http://oasis.forge.ocamlcore.org/`) and will take care of setting the prefix and enabling tests if they are not disabled. You can pass configure and build flags with the `#:configure-flags` and `#:build-flags`. The `#:test-flags` key can be passed to change the set of flags used to enable tests. The `#:use-make?` key can be used to bypass this system in the build and install phases.

When the package has a `configure` file, it is assumed that it is a hand-made configure script that requires a different argument format than in the `gnu-build-system`. You can add more flags with the `#:configure-flags` key.

When the package has a `Makefile` file (or `#:use-make?` is `#t`), it will be used and more flags can be passed to the build and install phases with the `#:make-flags` key.

Finally, some packages do not have these files and use a somewhat standard location for its build system. In that case, the build system will run `ocaml pkg/pkg.ml` or `ocaml pkg/build.ml` and take care of providing the path to the required findlib module. Additional flags can be passed via the `#:build-flags` key. Install is taken care of by `opam-installer`. In this case, the `opam` package must be added to the `native-inputs` field of the package definition.

Note that most OCaml packages assume they will be installed in the same directory as OCaml, which is not what we want in guix. In particular, they will install `.so` files in their module's directory, which is usually fine because it is in the OCaml compiler directory. In guix though, these libraries cannot be found and we use `CAML_LD_LIBRARY_PATH`. This variable points to `lib/ocaml/site-lib/stubslibs` and this is where `.so` libraries should be installed.

`python-build-system` [Scheme Variable]

This variable is exported by `(guix build-system python)`. It implements the more or less standard build procedure used by Python packages, which consists in running `python setup.py build` and then `python setup.py install --prefix=/gnu/store/...`.

For packages that install stand-alone Python programs under `bin/`, it takes care of wrapping these programs so that their `PYTHONPATH` environment variable points to all the Python libraries they depend on.

Which Python package is used to perform the build can be specified with the `#:python` parameter. This is a useful way to force a package to be built for a specific version of the Python interpreter, which might be necessary if the package is only compatible with a single interpreter version.

By default guix calls `setup.py` under control of `setuptools`, much like `pip` does. Some packages are not compatible with setuptools (and pip), thus you can disable this by setting the `#:use-setuptools` parameter to `#f`.

`perl-build-system` [Scheme Variable]

This variable is exported by `(guix build-system perl)`. It implements the standard build procedure for Perl packages, which either consists in running `perl Build.PL --prefix=/gnu/store/...`, followed by `Build` and `Build install`; or in running `perl Makefile.PL PREFIX=/gnu/store/...`, followed by `make` and `make install`, depending on which of `Build.PL` or `Makefile.PL` is present in the package distribution. Preference is given to the former if both `Build.PL` and `Makefile.PL` exist in the package distribution. This preference can be reversed by specifying `#t` for the `#:make-maker?` parameter.

The initial `perl Makefile.PL` or `perl Build.PL` invocation passes flags specified by the `#:make-maker-flags` or `#:module-build-flags` parameter, respectively.

Which Perl package is used can be specified with `#:perl`.

`r-build-system` [Scheme Variable]

This variable is exported by `(guix build-system r)`. It implements the build procedure used by R (`http://r-project.org`) packages, which essentially is little more

than running `R CMD INSTALL --library=/gnu/store/...` in an environment where `R_LIBS_SITE` contains the paths to all R package inputs. Tests are run after installation using the R function `tools::testInstalledPackage`.

`ruby-build-system` [Scheme Variable]

This variable is exported by `(guix build-system ruby)`. It implements the RubyGems build procedure used by Ruby packages, which involves running `gem build` followed by `gem install`.

The `source` field of a package that uses this build system typically references a gem archive, since this is the format that Ruby developers use when releasing their software. The build system unpacks the gem archive, potentially patches the source, runs the test suite, repackages the gem, and installs it. Additionally, directories and tarballs may be referenced to allow building unreleased gems from Git or a traditional source release tarball.

Which Ruby package is used can be specified with the `#:ruby` parameter. A list of additional flags to be passed to the `gem` command can be specified with the `#:gem-flags` parameter.

`waf-build-system` [Scheme Variable]

This variable is exported by `(guix build-system waf)`. It implements a build procedure around the `waf` script. The common phases—`configure`, `build`, and `install`—are implemented by passing their names as arguments to the `waf` script.

The `waf` script is executed by the Python interpreter. Which Python package is used to run the script can be specified with the `#:python` parameter.

`haskell-build-system` [Scheme Variable]

This variable is exported by `(guix build-system haskell)`. It implements the Cabal build procedure used by Haskell packages, which involves running `runhaskell Setup.hs configure --prefix=/gnu/store/...` and `runhaskell Setup.hs build`. Instead of installing the package by running `runhaskell Setup.hs install`, to avoid trying to register libraries in the read-only compiler store directory, the build system uses `runhaskell Setup.hs copy`, followed by `runhaskell Setup.hs register`. In addition, the build system generates the package documentation by running `runhaskell Setup.hs haddock`, unless `#:haddock? #f` is passed. Optional Haddock parameters can be passed with the help of the `#:haddock-flags` parameter. If the file `Setup.hs` is not found, the build system looks for `Setup.lhs` instead.

Which Haskell compiler is used can be specified with the `#:haskell` parameter which defaults to `ghc`.

`dub-build-system` [Scheme Variable]

This variable is exported by `(guix build-system dub)`. It implements the Dub build procedure used by D packages, which involves running `dub build` and `dub run`. Installation is done by copying the files manually.

Which D compiler is used can be specified with the `#:ldc` parameter which defaults to `ldc`.

`emacs-build-system` [Scheme Variable]

This variable is exported by `(guix build-system emacs)`. It implements an installation procedure similar to the packaging system of Emacs itself (see Section "Packages" in *The GNU Emacs Manual*).

It first creates the *package*`-autoloads.el` file, then it byte compiles all Emacs Lisp files. Differently from the Emacs packaging system, the Info documentation files are moved to the standard documentation directory and the `dir` file is deleted. Each package is installed in its own directory under `share/emacs/site-lisp/guix.d`.

Lastly, for packages that do not need anything as sophisticated, a "trivial" build system is provided. It is trivial in the sense that it provides basically no support: it does not pull any implicit inputs, and does not have a notion of build phases.

`trivial-build-system` [Scheme Variable]

This variable is exported by `(guix build-system trivial)`.

This build system requires a `#:builder` argument. This argument must be a Scheme expression that builds the package output(s)—as with `build-expression->derivation` (see Section 4.4 [Derivations], page 50).

4.3 The Store

Conceptually, the *store* is the place where derivations that have been built successfully are stored—by default, `/gnu/store`. Sub-directories in the store are referred to as *store items* or sometimes *store paths*. The store has an associated database that contains information such as the store paths referred to by each store path, and the list of *valid* store items—results of successful builds. This database resides in *`localstatedir`*`/guix/db`, where *localstatedir* is the state directory specified *via* `--localstatedir` at configure time, usually `/var`.

The store is *always* accessed by the daemon on behalf of its clients (see Section 2.5 [Invoking guix-daemon], page 11). To manipulate the store, clients connect to the daemon over a Unix-domain socket, send requests to it, and read the result—these are remote procedure calls, or RPCs.

> **Note:** Users must *never* modify files under `/gnu/store` directly. This would lead to inconsistencies and break the immutability assumptions of Guix's functional model (see Chapter 1 [Introduction], page 2).
>
> See Section 3.5 [Invoking guix gc], page 27, for information on how to check the integrity of the store and attempt recovery from accidental modifications.

The `(guix store)` module provides procedures to connect to the daemon, and to perform RPCs. These are described below. By default, `open-connection`, and thus all the `guix` commands, connect to the local daemon or to the URI specified by the `GUIX_DAEMON_SOCKET` environment variable.

`GUIX_DAEMON_SOCKET` [Environment Variable]

When set, the value of this variable should be a file name or a URI designating the daemon endpoint. When it is a file name, it denotes a Unix-domain socket to connect to. In addition to file names, the supported URI schemes are:

`file`
`unix` These are for Unix-domain sockets. `file:///var/guix/daemon-socket/socket` is equivalent to `/var/guix/daemon-socket/socket`.

`guix` These URIs denote connections over TCP/IP, without encryption nor authentication of the remote host. The URI must always specify both the host name and port number:

```
guix://master.guix.example.org:1234
```

This setup is suitable on local networks, such as clusters, where only trusted nodes may connect to the build daemon at `master.guix.example.org`.

`ssh` These URIs allow you to connect to a remote daemon over SSH[2]. A typical URL might look like this:

```
ssh://charlie@guix.example.org:22
```

As for `guix copy`, the usual OpenSSH client configuration files are honored (see Section 5.13 [Invoking guix copy], page 93).

Additional URI schemes may be supported in the future.

Note: The ability to connect to remote build daemons is considered experimental as of 0.13.0. Please get in touch with us to share any problems or suggestions you may have (see Chapter 7 [Contributing], page 242).

`open-connection` [*uri*] [*#:reserve-space? #t*] [Scheme Procedure]
Connect to the daemon over the Unix-domain socket at *uri* (a string). When *reserve-space?* is true, instruct it to reserve a little bit of extra space on the file system so that the garbage collector can still operate should the disk become full. Return a server object.

file defaults to *%default-socket-path*, which is the normal location given the options that were passed to `configure`.

`close-connection` *server* [Scheme Procedure]
Close the connection to *server*.

`current-build-output-port` [Scheme Variable]
This variable is bound to a SRFI-39 parameter, which refers to the port where build and error logs sent by the daemon should be written.

Procedures that make RPCs all take a server object as their first argument.

`valid-path?` *server path* [Scheme Procedure]
Return `#t` when *path* designates a valid store item and `#f` otherwise (an invalid item may exist on disk but still be invalid, for instance because it is the result of an aborted or failed build.)

A `&nix-protocol-error` condition is raised if *path* is not prefixed by the store directory (`/gnu/store`).

[2] This feature requires Guile-SSH (see Section 2.2 [Requirements], page 5).

`add-text-to-store` *server name text* [*references*] [Scheme Procedure]
Add *text* under file *name* in the store, and return its store path. *references* is the list of store paths referred to by the resulting store path.

`build-derivations` *server derivations* [Scheme Procedure]
Build *derivations* (a list of `<derivation>` objects or derivation paths), and return when the worker is done building them. Return `#t` on success.

Note that the `(guix monads)` module provides a monad as well as monadic versions of the above procedures, with the goal of making it more convenient to work with code that accesses the store (see Section 4.5 [The Store Monad], page 52).

This section is currently incomplete.

4.4 Derivations

Low-level build actions and the environment in which they are performed are represented by *derivations*. A derivation contains the following pieces of information:

- The outputs of the derivation—derivations produce at least one file or directory in the store, but may produce more.
- The inputs of the derivations, which may be other derivations or plain files in the store (patches, build scripts, etc.)
- The system type targeted by the derivation—e.g., `x86_64-linux`.
- The file name of a build script in the store, along with the arguments to be passed.
- A list of environment variables to be defined.

Derivations allow clients of the daemon to communicate build actions to the store. They exist in two forms: as an in-memory representation, both on the client- and daemon-side, and as files in the store whose name end in `.drv`—these files are referred to as *derivation paths*. Derivations paths can be passed to the `build-derivations` procedure to perform the build actions they prescribe (see Section 4.3 [The Store], page 48).

The `(guix derivations)` module provides a representation of derivations as Scheme objects, along with procedures to create and otherwise manipulate derivations. The lowest-level primitive to create a derivation is the `derivation` procedure:

`derivation` *store name builder args* [*#:outputs '("out")*] [Scheme Procedure]
[*#:hash #f*] [*#:hash-algo #f*] [*#:recursive? #f*] [*#:inputs '()*] [*#:env-vars '()*] [*#:system (%current-system)*] [*#:references-graphs #f*] [*#:allowed-references #f*] [*#:disallowed-references #f*] [*#:leaked-env-vars #f*] [*#:local-build? #f*] [*#:substitutable? #t*]
Build a derivation with the given arguments, and return the resulting `<derivation>` object.

When *hash* and *hash-algo* are given, a *fixed-output derivation* is created—i.e., one whose result is known in advance, such as a file download. If, in addition, *recursive?* is true, then that fixed output may be an executable file or a directory and *hash* must be the hash of an archive containing this output.

When *references-graphs* is true, it must be a list of file name/store path pairs. In that case, the reference graph of each store path is exported in the build environment in the corresponding file, in a simple text format.

When *allowed-references* is true, it must be a list of store items or outputs that the derivation's output may refer to. Likewise, *disallowed-references*, if true, must be a list of things the outputs may *not* refer to.

When *leaked-env-vars* is true, it must be a list of strings denoting environment variables that are allowed to "leak" from the daemon's environment to the build environment. This is only applicable to fixed-output derivations—i.e., when *hash* is true. The main use is to allow variables such as `http_proxy` to be passed to derivations that download files.

When *local-build?* is true, declare that the derivation is not a good candidate for offloading and should rather be built locally (see Section 2.4.2 [Daemon Offload Setup], page 8). This is the case for small derivations where the costs of data transfers would outweigh the benefits.

When *substitutable?* is false, declare that substitutes of the derivation's output should not be used (see Section 3.3 [Substitutes], page 25). This is useful, for instance, when building packages that capture details of the host CPU instruction set.

Here's an example with a shell script as its builder, assuming *store* is an open connection to the daemon, and *bash* points to a Bash executable in the store:

```
(use-modules (guix utils)
             (guix store)
             (guix derivations))

(let ((builder   ; add the Bash script to the store
        (add-text-to-store store "my-builder.sh"
                           "echo hello world > $out\n" '())))
  (derivation store "foo"
              bash `("-e" ,builder)
              #:inputs `((,bash) (,builder))
              #:env-vars '(("HOME" . "/homeless"))))
⇒ #<derivation /gnu/store/...-foo.drv => /gnu/store/...-foo>
```

As can be guessed, this primitive is cumbersome to use directly. A better approach is to write build scripts in Scheme, of course! The best course of action for that is to write the build code as a "G-expression", and to pass it to `gexp->derivation`. For more information, see Section 4.6 [G-Expressions], page 56.

Once upon a time, `gexp->derivation` did not exist and constructing derivations with build code written in Scheme was achieved with `build-expression->derivation`, documented below. This procedure is now deprecated in favor of the much nicer `gexp->derivation`.

`build-expression->derivation` *store name exp* [*#:system* (*%current-system*)] [*#:inputs* '()] [*#:outputs* '("out")] [*#:hash #f*] [*#:hash-algo #f*] [*#:recursive? #f*] [*#:env-vars* '()] [*#:modules* '()] [*#:references-graphs #f*] [*#:allowed-references #f*] [*#:disallowed-references #f*] [*#:local-build? #f*] [*#:substitutable? #t*] [*#:guile-for-build #f*] [Scheme Procedure]

Return a derivation that executes Scheme expression *exp* as a builder for derivation *name*. *inputs* must be a list of `(name drv-path sub-drv)` tuples; when *sub-drv*

is omitted, `"out"` is assumed. *modules* is a list of names of Guile modules from the current search path to be copied in the store, compiled, and made available in the load path during the execution of *exp*—e.g., `((guix build utils) (guix build gnu-build-system))`.

exp is evaluated in an environment where `%outputs` is bound to a list of output/path pairs, and where `%build-inputs` is bound to a list of string/output-path pairs made from *inputs*. Optionally, *env-vars* is a list of string pairs specifying the name and value of environment variables visible to the builder. The builder terminates by passing the result of *exp* to `exit`; thus, when *exp* returns `#f`, the build is considered to have failed.

exp is built using *guile-for-build* (a derivation). When *guile-for-build* is omitted or is `#f`, the value of the `%guile-for-build` fluid is used instead.

See the `derivation` procedure for the meaning of *references-graphs*, *allowed-references*, *disallowed-references*, *local-build?*, and *substitutable?*.

Here's an example of a single-output derivation that creates a directory containing one file:

```
(let ((builder '(let ((out (assoc-ref %outputs "out")))
                  (mkdir out)    ; create /gnu/store/...-goo
                  (call-with-output-file (string-append out "/test")
                    (lambda (p)
                      (display '(hello guix) p))))))
  (build-expression->derivation store "goo" builder))

⇒ #<derivation /gnu/store/...-goo.drv => ...>
```

4.5 The Store Monad

The procedures that operate on the store described in the previous sections all take an open connection to the build daemon as their first argument. Although the underlying model is functional, they either have side effects or depend on the current state of the store.

The former is inconvenient: the connection to the build daemon has to be carried around in all those functions, making it impossible to compose functions that do not take that parameter with functions that do. The latter can be problematic: since store operations have side effects and/or depend on external state, they have to be properly sequenced.

This is where the `(guix monads)` module comes in. This module provides a framework for working with *monads*, and a particularly useful monad for our uses, the *store monad*. Monads are a construct that allows two things: associating "context" with values (in our case, the context is the store), and building sequences of computations (here computations include accesses to the store). Values in a monad—values that carry this additional context—are called *monadic values*; procedures that return such values are called *monadic procedures*.

Consider this "normal" procedure:

```
(define (sh-symlink store)
  ;; Return a derivation that symlinks the 'bash' executable.
  (let* ((drv (package-derivation store bash))
         (out (derivation->output-path drv))
```

```
      (sh  (string-append out "/bin/bash")))
  (build-expression->derivation store "sh"
                                `(symlink ,sh %output))))
```

Using `(guix monads)` and `(guix gexp)`, it may be rewritten as a monadic function:

```
(define (sh-symlink)
  ;; Same, but return a monadic value.
  (mlet %store-monad ((drv (package->derivation bash)))
    (gexp->derivation "sh"
                      #~(symlink (string-append #$drv "/bin/bash")
                                 #$output))))
```

There are several things to note in the second version: the `store` parameter is now implicit and is "threaded" in the calls to the `package->derivation` and `gexp->derivation` monadic procedures, and the monadic value returned by `package->derivation` is *bound* using `mlet` instead of plain `let`.

As it turns out, the call to `package->derivation` can even be omitted since it will take place implicitly, as we will see later (see Section 4.6 [G-Expressions], page 56):

```
(define (sh-symlink)
  (gexp->derivation "sh"
                    #~(symlink (string-append #$bash "/bin/bash")
                               #$output)))
```

Calling the monadic `sh-symlink` has no effect. As someone once said, "you exit a monad like you exit a building on fire: by running". So, to exit the monad and get the desired effect, one must use `run-with-store`:

```
(run-with-store (open-connection) (sh-symlink))
⇒ /gnu/store/...-sh-symlink
```

Note that the `(guix monad-repl)` module extends the Guile REPL with new "meta-commands" to make it easier to deal with monadic procedures: `run-in-store`, and `enter-store-monad`. The former is used to "run" a single monadic value through the store:

```
scheme@(guile-user)> ,run-in-store (package->derivation hello)
$1 = #<derivation /gnu/store/...-hello-2.9.drv => ...>
```

The latter enters a recursive REPL, where all the return values are automatically run through the store:

```
scheme@(guile-user)> ,enter-store-monad
store-monad@(guile-user) [1]> (package->derivation hello)
$2 = #<derivation /gnu/store/...-hello-2.9.drv => ...>
store-monad@(guile-user) [1]> (text-file "foo" "Hello!")
$3 = "/gnu/store/...-foo"
store-monad@(guile-user) [1]> ,q
scheme@(guile-user)>
```

Note that non-monadic values cannot be returned in the `store-monad` REPL.

The main syntactic forms to deal with monads in general are provided by the `(guix monads)` module and are described below.

`with-monad` *monad body ...* [Scheme Syntax]
Evaluate any `>>=` or `return` forms in *body* as being in *monad*.

`return` *val* [Scheme Syntax]
Return a monadic value that encapsulates *val*.

`>>=` *mval mproc ...* [Scheme Syntax]
Bind monadic value *mval*, passing its "contents" to monadic procedures *mproc*...[3]. There can be one *mproc* or several of them, as in this example:

```
(run-with-state
    (with-monad %state-monad
      (>>= (return 1)
           (lambda (x) (return (+ 1 x)))
           (lambda (x) (return (* 2 x)))))
  'some-state)

⇒ 4
⇒ some-state
```

`mlet` *monad* ((*var mval*) ...) *body ...* [Scheme Syntax]
`mlet*` *monad* ((*var mval*) ...) *body ...* [Scheme Syntax]
Bind the variables *var* to the monadic values *mval* in *body*, which is a sequence of expressions. As with the bind operator, this can be thought of as "unpacking" the raw, non-monadic value "contained" in *mval* and making *var* refer to that raw, non-monadic value within the scope of the *body*. The form (*var* `->` *val*) binds *var* to the "normal" value *val*, as per `let`. The binding operations occur in sequence from left to right. The last expression of *body* must be a monadic expression, and its result will become the result of the `mlet` or `mlet*` when run in the *monad*.

`mlet*` is to `mlet` what `let*` is to `let` (see Section "Local Bindings" in *GNU Guile Reference Manual*).

`mbegin` *monad mexp ...* [Scheme System]
Bind *mexp* and the following monadic expressions in sequence, returning the result of the last expression. Every expression in the sequence must be a monadic expression.

This is akin to `mlet`, except that the return values of the monadic expressions are ignored. In that sense, it is analogous to `begin`, but applied to monadic expressions.

`mwhen` *condition mexp0 mexp* ...* [Scheme System]
When *condition* is true, evaluate the sequence of monadic expressions *mexp0..mexp** as in an `mbegin`. When *condition* is false, return `*unspecified*` in the current monad. Every expression in the sequence must be a monadic expression.

`munless` *condition mexp0 mexp* ...* [Scheme System]
When *condition* is false, evaluate the sequence of monadic expressions *mexp0..mexp** as in an `mbegin`. When *condition* is true, return `*unspecified*` in the current monad. Every expression in the sequence must be a monadic expression.

[3] This operation is commonly referred to as "bind", but that name denotes an unrelated procedure in Guile. Thus we use this somewhat cryptic symbol inherited from the Haskell language.

The `(guix monads)` module provides the *state monad*, which allows an additional value—the state—to be *threaded* through monadic procedure calls.

`%state-monad` [Scheme Variable]

The state monad. Procedures in the state monad can access and change the state that is threaded.

Consider the example below. The `square` procedure returns a value in the state monad. It returns the square of its argument, but also increments the current state value:

```
(define (square x)
  (mlet %state-monad ((count (current-state)))
    (mbegin %state-monad
      (set-current-state (+ 1 count))
      (return (* x x)))))

(run-with-state (sequence %state-monad (map square (iota 3))) 0)
⇒ (0 1 4)
⇒ 3
```

When "run" through *%state-monad*, we obtain that additional state value, which is the number of `square` calls.

`current-state` [Monadic Procedure]

Return the current state as a monadic value.

`set-current-state` *value* [Monadic Procedure]

Set the current state to *value* and return the previous state as a monadic value.

`state-push` *value* [Monadic Procedure]

Push *value* to the current state, which is assumed to be a list, and return the previous state as a monadic value.

`state-pop` [Monadic Procedure]

Pop a value from the current state and return it as a monadic value. The state is assumed to be a list.

`run-with-state` *mval* [*state*] [Scheme Procedure]

Run monadic value *mval* starting with *state* as the initial state. Return two values: the resulting value, and the resulting state.

The main interface to the store monad, provided by the `(guix store)` module, is as follows.

`%store-monad` [Scheme Variable]

The store monad—an alias for *%state-monad*.

Values in the store monad encapsulate accesses to the store. When its effect is needed, a value of the store monad must be "evaluated" by passing it to the `run-with-store` procedure (see below.)

`run-with-store` *store mval* [*#:guile-for-build*] [*#:system* (*%current-system*)] [Scheme Procedure]
Run *mval*, a monadic value in the store monad, in *store*, an open store connection.

`text-file` *name text* [*references*] [Monadic Procedure]
Return as a monadic value the absolute file name in the store of the file containing *text*, a string. *references* is a list of store items that the resulting text file refers to; it defaults to the empty list.

`interned-file` *file* [*name*] [*#:recursive? #t*] [*#:select?* (*const #t*)] [Monadic Procedure]
Return the name of *file* once interned in the store. Use *name* as its store name, or the basename of *file* if *name* is omitted.

When *recursive?* is true, the contents of *file* are added recursively; if *file* designates a flat file and *recursive?* is true, its contents are added, and its permission bits are kept.

When *recursive?* is true, call `(select? file stat)` for each directory entry, where *file* is the entry's absolute file name and *stat* is the result of `lstat`; exclude entries for which *select?* does not return true.

The example below adds a file to the store, under two different names:

```
(run-with-store (open-connection)
  (mlet %store-monad ((a (interned-file "README"))
                      (b (interned-file "README" "LEGU-MIN")))
    (return (list a b))))

⇒ ("/gnu/store/rwm...-README" "/gnu/store/44i...-LEGU-MIN")
```

The `(guix packages)` module exports the following package-related monadic procedures:

`package-file` *package* [*file*] [*#:system* (*%current-system*)] [*#:target #f*] [*#:output "out"*] [Monadic Procedure]
Return as a monadic value in the absolute file name of *file* within the *output* directory of *package*. When *file* is omitted, return the name of the *output* directory of *package*. When *target* is true, use it as a cross-compilation target triplet.

`package->derivation` *package* [*system*] [Monadic Procedure]
`package->cross-derivation` *package target* [*system*] [Monadic Procedure]
Monadic version of `package-derivation` and `package-cross-derivation` (see Section 4.1 [Defining Packages], page 35).

4.6 G-Expressions

So we have "derivations", which represent a sequence of build actions to be performed to produce an item in the store (see Section 4.4 [Derivations], page 50). These build actions are performed when asking the daemon to actually build the derivations; they are run by the daemon in a container (see Section 2.5 [Invoking guix-daemon], page 11).

It should come as no surprise that we like to write these build actions in Scheme. When we do that, we end up with two *strata* of Scheme code[4]: the "host code"—code that defines packages, talks to the daemon, etc.—and the "build code"—code that actually performs build actions, such as making directories, invoking `make`, etc.

To describe a derivation and its build actions, one typically needs to embed build code inside host code. It boils down to manipulating build code as data, and the homoiconicity of Scheme—code has a direct representation as data—comes in handy for that. But we need more than the normal `quasiquote` mechanism in Scheme to construct build expressions.

The `(guix gexp)` module implements *G-expressions*, a form of S-expressions adapted to build expressions. G-expressions, or *gexps*, consist essentially of three syntactic forms: `gexp`, `ungexp`, and `ungexp-splicing` (or simply: `#~`, `#$`, and `#$@`), which are comparable to `quasiquote`, `unquote`, and `unquote-splicing`, respectively (see Section "Expression Syntax" in *GNU Guile Reference Manual*). However, there are major differences:

- Gexps are meant to be written to a file and run or manipulated by other processes.
- When a high-level object such as a package or derivation is unquoted inside a gexp, the result is as if its output file name had been introduced.
- Gexps carry information about the packages or derivations they refer to, and these dependencies are automatically added as inputs to the build processes that use them.

This mechanism is not limited to package and derivation objects: *compilers* able to "lower" other high-level objects to derivations or files in the store can be defined, such that these objects can also be inserted into gexps. For example, a useful type of high-level objects that can be inserted in a gexp is "file-like objects", which make it easy to add files to the store and to refer to them in derivations and such (see `local-file` and `plain-file` below.)

To illustrate the idea, here is an example of a gexp:

```
(define build-exp
  #~(begin
      (mkdir #$output)
      (chdir #$output)
      (symlink (string-append #$coreutils "/bin/ls")
               "list-files")))
```

This gexp can be passed to `gexp->derivation`; we obtain a derivation that builds a directory containing exactly one symlink to `/gnu/store/...-coreutils-8.22/bin/ls`:

```
(gexp->derivation "the-thing" build-exp)
```

As one would expect, the `"/gnu/store/...-coreutils-8.22"` string is substituted to the reference to the *coreutils* package in the actual build code, and *coreutils* is automatically made an input to the derivation. Likewise, `#$output` (equivalent to `(ungexp output)`) is replaced by a string containing the directory name of the output of the derivation.

In a cross-compilation context, it is useful to distinguish between references to the *native* build of a package—that can run on the host—versus references to cross builds of a package. To that end, the `#+` plays the same role as `#$`, but is a reference to a native package build:

[4] The term *stratum* in this context was coined by Manuel Serrano et al. in the context of their work on Hop. Oleg Kiselyov, who has written insightful essays and code on this topic (`http://okmij.org/ftp/meta-programming/#meta-scheme`), refers to this kind of code generation as *staging*.

```
(gexp->derivation "vi"
   #~(begin
       (mkdir #$output)
       (system* (string-append #+coreutils "/bin/ln")
                "-s"
                (string-append #$emacs "/bin/emacs")
                (string-append #$output "/bin/vi")))
   #:target "mips64el-linux-gnu")
```

In the example above, the native build of *coreutils* is used, so that `ln` can actually run on the host; but then the cross-compiled build of *emacs* is referenced.

Another gexp feature is *imported modules*: sometimes you want to be able to use certain Guile modules from the "host environment" in the gexp, so those modules should be imported in the "build environment". The `with-imported-modules` form allows you to express that:

```
(let ((build (with-imported-modules '((guix build utils))
               #~(begin
                   (use-modules (guix build utils))
                   (mkdir-p (string-append #$output "/bin"))))))
  (gexp->derivation "empty-dir"
                    #~(begin
                        #$build
                        (display "success!\n")
                        #t)))
```

In this example, the `(guix build utils)` module is automatically pulled into the isolated build environment of our gexp, such that `(use-modules (guix build utils))` works as expected.

Usually you want the *closure* of the module to be imported—i.e., the module itself and all the modules it depends on—rather than just the module; failing to do that, attempts to use the module will fail because of missing dependent modules. The `source-module-closure` procedure computes the closure of a module by looking at its source file headers, which comes in handy in this case:

```
(use-modules (guix modules))   ;for 'source-module-closure'

(with-imported-modules (source-module-closure
                         '((guix build utils)
                           (gnu build vm)))
  (gexp->derivation "something-with-vms"
                    #~(begin
                        (use-modules (guix build utils)
                                     (gnu build vm))
                        ...)))
```

The syntactic form to construct gexps is summarized below.

`#~exp` [Scheme Syntax]
`(gexp exp)` [Scheme Syntax]

Return a G-expression containing *exp*. *exp* may contain one or more of the following forms:

`#$obj`
`(ungexp obj)`
: Introduce a reference to *obj*. *obj* may have one of the supported types, for example a package or a derivation, in which case the `ungexp` form is replaced by its output file name—e.g., `"/gnu/store/...-coreutils-8.22`.

  If *obj* is a list, it is traversed and references to supported objects are substituted similarly.

  If *obj* is another gexp, its contents are inserted and its dependencies are added to those of the containing gexp.

  If *obj* is another kind of object, it is inserted as is.

`#$obj:output`
`(ungexp obj output)`
: This is like the form above, but referring explicitly to the *output* of *obj*—this is useful when *obj* produces multiple outputs (see Section 3.4 [Packages with Multiple Outputs], page 27).

`#+obj`
`#+obj:output`
`(ungexp-native obj)`
`(ungexp-native obj output)`
: Same as `ungexp`, but produces a reference to the *native* build of *obj* when used in a cross compilation context.

`#$output[:output]`
`(ungexp output [output])`
: Insert a reference to derivation output *output*, or to the main output when *output* is omitted.

  This only makes sense for gexps passed to `gexp->derivation`.

`#$@lst`
`(ungexp-splicing lst)`
: Like the above, but splices the contents of *lst* inside the containing list.

`#+@lst`
`(ungexp-native-splicing lst)`
: Like the above, but refers to native builds of the objects listed in *lst*.

G-expressions created by `gexp` or `#~` are run-time objects of the `gexp?` type (see below.)

`with-imported-modules modules body...` [Scheme Syntax]

Mark the gexps defined in *body*... as requiring *modules* in their execution environment.

Each item in *modules* can be the name of a module, such as `(guix build utils)`, or it can be a module name, followed by an arrow, followed by a file-like object:

```
`((guix build utils)
  (guix gcrypt)
  ((guix config) => ,(scheme-file "config.scm"
                                  #~(define-module ...))))
```

In the example above, the first two modules are taken from the search path, and the last one is created from the given file-like object.

This form has *lexical* scope: it has an effect on the gexps directly defined in *body*..., but not on those defined, say, in procedures called from *body*....

`gexp?` *obj* [Scheme Procedure]

Return `#t` if *obj* is a G-expression.

G-expressions are meant to be written to disk, either as code building some derivation, or as plain files in the store. The monadic procedures below allow you to do that (see Section 4.5 [The Store Monad], page 52, for more information about monads.)

`gexp->derivation` *name exp* [*#:system* (*%current-system*)] [Monadic Procedure]
[*#:target #f*] [*#:graft? #t*] [*#:hash #f*] [*#:hash-algo #f*] [*#:recursive? #f*] [*#:env-vars '()*] [*#:modules '()*] [*#:module-path* `%load-path`] [*#:references-graphs #f*] [*#:allowed-references #f*] [*#:disallowed-references #f*] [*#:leaked-env-vars #f*] [*#:script-name* (*string-append* `name` `"-builder"`)] [*#:local-build? #f*] [*#:substitutable? #t*] [*#:guile-for-build #f*]

Return a derivation *name* that runs *exp* (a gexp) with *guile-for-build* (a derivation) on *system*; *exp* is stored in a file called *script-name*. When *target* is true, it is used as the cross-compilation target triplet for packages referred to by *exp*.

modules is deprecated in favor of `with-imported-modules`. Its meaning is to make *modules* available in the evaluation context of *exp*; *modules* is a list of names of Guile modules searched in *module-path* to be copied in the store, compiled, and made available in the load path during the execution of *exp*—e.g., `((guix build utils) (guix build gnu-build-system))`.

graft? determines whether packages referred to by *exp* should be grafted when applicable.

When *references-graphs* is true, it must be a list of tuples of one of the following forms:

```
(file-name package)
(file-name package output)
(file-name derivation)
(file-name derivation output)
(file-name store-item)
```

The right-hand-side of each element of *references-graphs* is automatically made an input of the build process of *exp*. In the build environment, each *file-name* contains the reference graph of the corresponding item, in a simple text format.

allowed-references must be either `#f` or a list of output names and packages. In the latter case, the list denotes store items that the result is allowed to refer to. Any reference to another store item will lead to a build error. Similarly for *disallowed-references*, which can list items that must not be referenced by the outputs.

The other arguments are as for `derivation` (see Section 4.4 [Derivations], page 50).

The `local-file`, `plain-file`, `computed-file`, `program-file`, and `scheme-file` procedures below return *file-like objects*. That is, when unquoted in a G-expression, these objects lead to a file in the store. Consider this G-expression:

```
#~(system* #$(file-append glibc "/sbin/nscd") "-f"
           #$(local-file "/tmp/my-nscd.conf"))
```

The effect here is to "intern" `/tmp/my-nscd.conf` by copying it to the store. Once expanded, for instance *via* `gexp->derivation`, the G-expression refers to that copy under `/gnu/store`; thus, modifying or removing the file in `/tmp` does not have any effect on what the G-expression does. `plain-file` can be used similarly; it differs in that the file content is directly passed as a string.

`local-file` *file* [*name*] [*#:recursive? #f*] [*#:select? (const #t)*] [Scheme Procedure]

Return an object representing local file *file* to add to the store; this object can be used in a gexp. If *file* is a relative file name, it is looked up relative to the source file where this form appears. *file* will be added to the store under *name*–by default the base name of *file*.

When *recursive?* is true, the contents of *file* are added recursively; if *file* designates a flat file and *recursive?* is true, its contents are added, and its permission bits are kept.

When *recursive?* is true, call `(`*select? file stat*`)` for each directory entry, where *file* is the entry's absolute file name and *stat* is the result of `lstat`; exclude entries for which *select?* does not return true.

This is the declarative counterpart of the `interned-file` monadic procedure (see Section 4.5 [The Store Monad], page 52).

`plain-file` *name content* [Scheme Procedure]

Return an object representing a text file called *name* with the given *content* (a string) to be added to the store.

This is the declarative counterpart of `text-file`.

`computed-file` *name gexp* [*#:options '(#:local-build? #t)*] [Scheme Procedure]

Return an object representing the store item *name*, a file or directory computed by *gexp*. *options* is a list of additional arguments to pass to `gexp->derivation`.

This is the declarative counterpart of `gexp->derivation`.

`gexp->script` *name exp* [Monadic Procedure]

Return an executable script *name* that runs *exp* using *guile*, with *exp*'s imported modules in its search path.

The example below builds a script that simply invokes the `ls` command:

```
(use-modules (guix gexp) (gnu packages base))
```

```
(gexp->script "list-files"
              #~(execl #$(file-append coreutils "/bin/ls")
                       "ls"))
```

When "running" it through the store (see Section 4.5 [The Store Monad], page 52), we obtain a derivation that produces an executable file `/gnu/store/...-list-files` along these lines:

```
#!/gnu/store/...-guile-2.0.11/bin/guile -ds
!#
(execl "/gnu/store/...-coreutils-8.22"/bin/ls" "ls")
```

`program-file` *name exp* [*#:guile #f*] [Scheme Procedure]

Return an object representing the executable store item *name* that runs *gexp*. *guile* is the Guile package used to execute that script.

This is the declarative counterpart of `gexp->script`.

`gexp->file` *name exp* [*#:set-load-path? #t*] [Monadic Procedure]

Return a derivation that builds a file *name* containing *exp*. When *set-load-path?* is true, emit code in the resulting file to set `%load-path` and `%load-compiled-path` to honor *exp*'s imported modules.

The resulting file holds references to all the dependencies of *exp* or a subset thereof.

`scheme-file` *name exp* [Scheme Procedure]

Return an object representing the Scheme file *name* that contains *exp*.

This is the declarative counterpart of `gexp->file`.

`text-file*` *name text ...* [Monadic Procedure]

Return as a monadic value a derivation that builds a text file containing all of *text*. *text* may list, in addition to strings, objects of any type that can be used in a gexp: packages, derivations, local file objects, etc. The resulting store file holds references to all these.

This variant should be preferred over `text-file` anytime the file to create will reference items from the store. This is typically the case when building a configuration file that embeds store file names, like this:

```
(define (profile.sh)
  ;; Return the name of a shell script in the store that
  ;; initializes the 'PATH' environment variable.
  (text-file* "profile.sh"
              "export PATH=" coreutils "/bin:"
              grep "/bin:" sed "/bin\n"))
```

In this example, the resulting `/gnu/store/...-profile.sh` file will reference *coreutils*, *grep*, and *sed*, thereby preventing them from being garbage-collected during its lifetime.

`mixed-text-file` *name text ...* [Scheme Procedure]

Return an object representing store file *name* containing *text*. *text* is a sequence of strings and file-like objects, as in:

```
(mixed-text-file "profile"
```

```
                    "export PATH=" coreutils "/bin:" grep "/bin")
```

This is the declarative counterpart of `text-file*`.

`file-append` *obj suffix ...* [Scheme Procedure]

Return a file-like object that expands to the concatenation of *obj* and *suffix*, where *obj* is a lowerable object and each *suffix* is a string.

As an example, consider this gexp:

```
(gexp->script "run-uname"
              #~(system* #$(file-append coreutils
                                        "/bin/uname")))
```

The same effect could be achieved with:

```
(gexp->script "run-uname"
              #~(system* (string-append #$coreutils
                                        "/bin/uname")))
```

There is one difference though: in the `file-append` case, the resulting script contains the absolute file name as a string, whereas in the second case, the resulting script contains a `(string-append ...)` expression to construct the file name *at run time*.

Of course, in addition to gexps embedded in "host" code, there are also modules containing build tools. To make it clear that they are meant to be used in the build stratum, these modules are kept in the `(guix build ...)` name space.

Internally, high-level objects are *lowered*, using their compiler, to either derivations or store items. For instance, lowering a package yields a derivation, and lowering a `plain-file` yields a store item. This is achieved using the `lower-object` monadic procedure.

`lower-object` *obj* [*system*] [*#:target #f*] [Monadic Procedure]

Return as a value in *%store-monad* the derivation or store item corresponding to *obj* for *system*, cross-compiling for *target* if *target* is true. *obj* must be an object that has an associated gexp compiler, such as a `<package>`.

5 Utilities

This section describes Guix command-line utilities. Some of them are primarily targeted at developers and users who write new package definitions, while others are more generally useful. They complement the Scheme programming interface of Guix in a convenient way.

5.1 Invoking `guix build`

The `guix build` command builds packages or derivations and their dependencies, and prints the resulting store paths. Note that it does not modify the user's profile—this is the job of the `guix package` command (see Section 3.2 [Invoking guix package], page 18). Thus, it is mainly useful for distribution developers.

The general syntax is:

```
guix build options package-or-derivation...
```

As an example, the following command builds the latest versions of Emacs and of Guile, displays their build logs, and finally displays the resulting directories:

```
guix build emacs guile
```

Similarly, the following command builds all the available packages:

```
guix build --quiet --keep-going \
  `guix package -A | cut -f1,2 --output-delimiter=@`
```

package-or-derivation may be either the name of a package found in the software distribution such as `coreutils` or `coreutils@8.20`, or a derivation such as `/gnu/store/...-coreutils-8.19.drv`. In the former case, a package with the corresponding name (and optionally version) is searched for among the GNU distribution modules (see Section 6.6 [Package Modules], page 231).

Alternatively, the `--expression` option may be used to specify a Scheme expression that evaluates to a package; this is useful when disambiguating among several same-named packages or package variants is needed.

There may be zero or more *options*. The available options are described in the subsections below.

5.1.1 Common Build Options

A number of options that control the build process are common to `guix build` and other commands that can spawn builds, such as `guix package` or `guix archive`. These are the following:

`--load-path=`*directory*
`-L` *directory*
: Add *directory* to the front of the package module search path (see Section 6.6 [Package Modules], page 231).

  This allows users to define their own packages and make them visible to the command-line tools.

`--keep-failed`
`-K`
: Keep the build tree of failed builds. Thus, if a build fails, its build tree is kept under `/tmp`, in a directory whose name is shown at the end of the build log.

This is useful when debugging build issues. See Section 5.1.4 [Debugging Build Failures], page 70, for tips and tricks on how to debug build issues.

`--keep-going`
`-k` Keep going when some of the derivations fail to build; return only once all the builds have either completed or failed.

The default behavior is to stop as soon as one of the specified derivations has failed.

`--dry-run`
`-n` Do not build the derivations.

`--fallback`
When substituting a pre-built binary fails, fall back to building packages locally.

`--substitute-urls=`*urls*
Consider *urls* the whitespace-separated list of substitute source URLs, overriding the default list of URLs of `guix-daemon` (see [`guix-daemon` URLs], page 12).

This means that substitutes may be downloaded from *urls*, provided they are signed by a key authorized by the system administrator (see Section 3.3 [Substitutes], page 25).

When *urls* is the empty string, substitutes are effectively disabled.

`--no-substitutes`
Do not use substitutes for build products. That is, always build things locally instead of allowing downloads of pre-built binaries (see Section 3.3 [Substitutes], page 25).

`--no-grafts`
Do not "graft" packages. In practice, this means that package updates available as grafts are not applied. See Section 6.5 [Security Updates], page 229, for more information on grafts.

`--rounds=`*n*
Build each derivation *n* times in a row, and raise an error if consecutive build results are not bit-for-bit identical.

This is a useful way to detect non-deterministic builds processes. Non-deterministic build processes are a problem because they make it practically impossible for users to *verify* whether third-party binaries are genuine. See Section 5.12 [Invoking guix challenge], page 92, for more.

Note that, currently, the differing build results are not kept around, so you will have to manually investigate in case of an error—e.g., by stashing one of the build results with `guix archive --export` (see Section 3.8 [Invoking guix archive], page 32), then rebuilding, and finally comparing the two results.

`--no-build-hook`
Do not attempt to offload builds *via* the "build hook" of the daemon (see Section 2.4.2 [Daemon Offload Setup], page 8). That is, always build things locally instead of offloading builds to remote machines.

`--max-silent-time=`*seconds*
: When the build or substitution process remains silent for more than *seconds*, terminate it and report a build failure.

`--timeout=`*seconds*
: Likewise, when the build or substitution process lasts for more than *seconds*, terminate it and report a build failure.

  By default there is no timeout. This behavior can be restored with `--timeout=0`.

`--verbosity=`*level*
: Use the given verbosity level. *level* must be an integer between 0 and 5; higher means more verbose output. Setting a level of 4 or more may be helpful when debugging setup issues with the build daemon.

`--cores=`*n*
`-c` *n*
: Allow the use of up to *n* CPU cores for the build. The special value `0` means to use as many CPU cores as available.

`--max-jobs=`*n*
`-M` *n*
: Allow at most *n* build jobs in parallel. See Section 2.5 [Invoking guix-daemon], page 11, for details about this option and the equivalent `guix-daemon` option.

Behind the scenes, `guix build` is essentially an interface to the `package-derivation` procedure of the `(guix packages)` module, and to the `build-derivations` procedure of the `(guix derivations)` module.

In addition to options explicitly passed on the command line, `guix build` and other `guix` commands that support building honor the `GUIX_BUILD_OPTIONS` environment variable.

`GUIX_BUILD_OPTIONS` [Environment Variable]
: Users can define this variable to a list of command line options that will automatically be used by `guix build` and other `guix` commands that can perform builds, as in the example below:

  ```
  $ export GUIX_BUILD_OPTIONS="--no-substitutes -c 2 -L /foo/bar"
  ```

  These options are parsed independently, and the result is appended to the parsed command-line options.

5.1.2 Package Transformation Options

Another set of command-line options supported by `guix build` and also `guix package` are *package transformation options*. These are options that make it possible to define *package variants*—for instance, packages built from different source code. This is a convenient way to create customized packages on the fly without having to type in the definitions of package variants (see Section 4.1 [Defining Packages], page 35).

`--with-source=`*source*
: Use *source* as the source of the corresponding package. *source* must be a file name or a URL, as for `guix download` (see Section 5.3 [Invoking guix download], page 72).

  The "corresponding package" is taken to be the one specified on the command line the name of which matches the base of *source*—e.g., if *source* is

`/src/guile-2.0.10.tar.gz`, the corresponding package is `guile`. Likewise, the version string is inferred from *source*; in the previous example, it is `2.0.10`.

This option allows users to try out versions of packages other than the one provided by the distribution. The example below downloads `ed-1.7.tar.gz` from a GNU mirror and uses that as the source for the `ed` package:

```
guix build ed --with-source=mirror://gnu/ed/ed-1.7.tar.gz
```

As a developer, `--with-source` makes it easy to test release candidates:

```
guix build guile --with-source=../guile-2.0.9.219-e1bb7.tar.xz
```

. . . or to build from a checkout in a pristine environment:

```
$ git clone git://git.sv.gnu.org/guix.git
$ guix build guix --with-source=./guix
```

`--with-input=`*package*`=`*replacement*

Replace dependency on *package* by a dependency on *replacement*. *package* must be a package name, and *replacement* must be a package specification such as `guile` or `guile@1.8`.

For instance, the following command builds Guix, but replaces its dependency on the current stable version of Guile with a dependency on the legacy version of Guile, `guile@2.0`:

```
guix build --with-input=guile=guile@2.0 guix
```

This is a recursive, deep replacement. So in this example, both `guix` and its dependency `guile-json` (which also depends on `guile`) get rebuilt against `guile@2.0`.

This is implemented using the `package-input-rewriting` Scheme procedure (see Section 4.1 [Defining Packages], page 35).

`--with-graft=`*package*`=`*replacement*

This is similar to `--with-input` but with an important difference: instead of rebuilding the whole dependency chain, *replacement* is built and then *grafted* onto the binaries that were initially referring to *package*. See Section 6.5 [Security Updates], page 229, for more information on grafts.

For example, the command below grafts version 3.5.4 of GnuTLS onto Wget and all its dependencies, replacing references to the version of GnuTLS they currently refer to:

```
guix build --with-graft=gnutls=gnutls@3.5.4 wget
```

This has the advantage of being much faster than rebuilding everything. But there is a caveat: it works if and only if *package* and *replacement* are strictly compatible—for example, if they provide a library, the application binary interface (ABI) of those libraries must be compatible. If *replacement* is somehow incompatible with *package*, then the resulting package may be unusable. Use with care!

5.1.3 Additional Build Options

The command-line options presented below are specific to `guix build`.

`--quiet`
`-q` Build quietly, without displaying the build log. Upon completion, the build log is kept in `/var` (or similar) and can always be retrieved using the `--log-file` option.

`--file=`*file*
`-f` *file*

Build the package or derivation that the code within *file* evaluates to.

As an example, *file* might contain a package definition like this (see Section 4.1 [Defining Packages], page 35):

```
(use-modules (guix)
             (guix build-system gnu)
             (guix licenses))

(package
  (name "hello")
  (version "2.10")
  (source (origin
            (method url-fetch)
            (uri (string-append "mirror://gnu/hello/hello-" version
                                ".tar.gz"))
            (sha256
             (base32
              "0ssi1wpaf7plaswqqjwigppsg5fyh99vdlb9kzl7c9lng89ndq1i"))))
  (build-system gnu-build-system)
  (synopsis "Hello, GNU world:  An example GNU package")
  (description "Guess what GNU Hello prints!")
  (home-page "http://www.gnu.org/software/hello/")
  (license gpl3+))
```

`--expression=`*expr*
`-e` *expr* Build the package or derivation *expr* evaluates to.

For example, *expr* may be `(@ (gnu packages guile) guile-1.8)`, which unambiguously designates this specific variant of version 1.8 of Guile.

Alternatively, *expr* may be a G-expression, in which case it is used as a build program passed to `gexp->derivation` (see Section 4.6 [G-Expressions], page 56).

Lastly, *expr* may refer to a zero-argument monadic procedure (see Section 4.5 [The Store Monad], page 52). The procedure must return a derivation as a monadic value, which is then passed through `run-with-store`.

`--source`
`-S` Build the source derivations of the packages, rather than the packages themselves.

For instance, `guix build -S gcc` returns something like `/gnu/store/...-gcc-4.7.2.tar.bz2`, which is the GCC source tarball.

The returned source tarball is the result of applying any patches and code snippets specified in the package `origin` (see Section 4.1 [Defining Packages], page 35).

`--sources`

Fetch and return the source of *package-or-derivation* and all their dependencies, recursively. This is a handy way to obtain a local copy of all the source code needed to build *packages*, allowing you to eventually build them even without network access. It is an extension of the `--source` option and can accept one of the following optional argument values:

`package` This value causes the `--sources` option to behave in the same way as the `--source` option.

`all` Build the source derivations of all packages, including any source that might be listed as `inputs`. This is the default value.

```
$ guix build --sources tzdata
The following derivations will be built:
   /gnu/store/...-tzdata2015b.tar.gz.drv
   /gnu/store/...-tzcode2015b.tar.gz.drv
```

`transitive`

Build the source derivations of all packages, as well of all transitive inputs to the packages. This can be used e.g. to prefetch package source for later offline building.

```
$ guix build --sources=transitive tzdata
The following derivations will be built:
   /gnu/store/...-tzcode2015b.tar.gz.drv
   /gnu/store/...-findutils-4.4.2.tar.xz.drv
   /gnu/store/...-grep-2.21.tar.xz.drv
   /gnu/store/...-coreutils-8.23.tar.xz.drv
   /gnu/store/...-make-4.1.tar.xz.drv
   /gnu/store/...-bash-4.3.tar.xz.drv
...
```

`--system=`*system*

`-s` *system* Attempt to build for *system*—e.g., `i686-linux`—instead of the system type of the build host.

An example use of this is on Linux-based systems, which can emulate different personalities. For instance, passing `--system=i686-linux` on an `x86_64-linux` system allows users to build packages in a complete 32-bit environment.

`--target=`*triplet*

Cross-build for *triplet*, which must be a valid GNU triplet, such as `"mips64el-linux-gnu"` (see Section "Specifying target triplets" in *Autoconf*).

`--check` Rebuild *package-or-derivation*, which are already available in the store, and raise an error if the build results are not bit-for-bit identical.

This mechanism allows you to check whether previously installed substitutes are genuine (see Section 3.3 [Substitutes], page 25), or whether the build result of

a package is deterministic. See Section 5.12 [Invoking guix challenge], page 92, for more background information and tools.

When used in conjunction with `--keep-failed`, the differing output is kept in the store, under `/gnu/store/...-check`. This makes it easy to look for differences between the two results.

`--repair` Attempt to repair the specified store items, if they are corrupt, by re-downloading or rebuilding them.

This operation is not atomic and thus restricted to `root`.

`--derivations`
`-d` Return the derivation paths, not the output paths, of the given packages.

`--root=`*file*
`-r` *file* Make *file* a symlink to the result, and register it as a garbage collector root.

`--log-file`
Return the build log file names or URLs for the given *package-or-derivation*, or raise an error if build logs are missing.

This works regardless of how packages or derivations are specified. For instance, the following invocations are equivalent:

```
guix build --log-file ‘guix build -d guile‘
guix build --log-file ‘guix build guile‘
guix build --log-file guile
guix build --log-file -e ’(@ (gnu packages guile) guile-2.0)’
```

If a log is unavailable locally, and unless `--no-substitutes` is passed, the command looks for a corresponding log on one of the substitute servers (as specified with `--substitute-urls`.)

So for instance, imagine you want to see the build log of GDB on MIPS, but you are actually on an `x86_64` machine:

```
$ guix build --log-file gdb -s mips64el-linux
https://hydra.gnu.org/log/...-gdb-7.10
```

You can freely access a huge library of build logs!

5.1.4 Debugging Build Failures

When defining a new package (see Section 4.1 [Defining Packages], page 35), you will probably find yourself spending some time debugging and tweaking the build until it succeeds. To do that, you need to operate the build commands yourself in an environment as close as possible to the one the build daemon uses.

To that end, the first thing to do is to use the `--keep-failed` or `-K` option of `guix build`, which will keep the failed build tree in `/tmp` or whatever directory you specified as `TMPDIR` (see Section 5.1 [Invoking guix build], page 64).

From there on, you can `cd` to the failed build tree and source the `environment-variables` file, which contains all the environment variable definitions that were in place when the build failed. So let’s say you’re debugging a build failure in package `foo`; a typical session would look like this:

```
$ guix build foo -K
```

```
... build fails
$ cd /tmp/guix-build-foo.drv-0
$ source ./environment-variables
$ cd foo-1.2
```

Now, you can invoke commands as if you were the daemon (almost) and troubleshoot your build process.

Sometimes it happens that, for example, a package's tests pass when you run them manually but they fail when the daemon runs them. This can happen because the daemon runs builds in containers where, unlike in our environment above, network access is missing, `/bin/sh` does not exist, etc. (see Section 2.4.1 [Build Environment Setup], page 7).

In such cases, you may need to run inspect the build process from within a container similar to the one the build daemon creates:

```
$ guix build -K foo
...
$ cd /tmp/guix-build-foo.drv-0
$ guix environment -C foo --ad-hoc strace gdb
[env]# source ./environment-variables
[env]# cd foo-1.2
```

Here, `guix environment -C` creates a container and spawns a new shell in it (see Section 5.10 [Invoking guix environment], page 86). The `--ad-hoc strace gdb` part adds the `strace` and `gdb` commands to the container, which would may find handy while debugging.

To get closer to a container like that used by the build daemon, we can remove `/bin/sh`:

```
[env]# rm /bin/sh
```

(Don't worry, this is harmless: this is all happening in the throw-away container created by `guix environment`.)

The `strace` command is probably not in the search path, but we can run:

```
[env]# $GUIX_ENVIRONMENT/bin/strace -f -o log make check
```

In this way, not only you will have reproduced the environment variables the daemon uses, you will also be running the build process in a container similar to the one the daemon uses.

5.2 Invoking `guix edit`

So many packages, so many source files! The `guix edit` command facilitates the life of users and packagers by pointing their editor at the source file containing the definition of the specified packages. For instance:

```
guix edit gcc@4.9 vim
```

launches the program specified in the `VISUAL` or in the `EDITOR` environment variable to view the recipe of GCC 4.9.3 and that of Vim.

If you are using a Guix Git checkout (see Section 7.1 [Building from Git], page 242), or have created your own packages on `GUIX_PACKAGE_PATH` (see Section 4.1 [Defining Packages], page 35), you will be able to edit the package recipes. Otherwise, you will be able to examine the read-only recipes for packages currently in the store.

5.3 Invoking `guix download`

When writing a package definition, developers typically need to download a source tarball, compute its SHA256 hash, and write that hash in the package definition (see Section 4.1 [Defining Packages], page 35). The `guix download` tool helps with this task: it downloads a file from the given URI, adds it to the store, and prints both its file name in the store and its SHA256 hash.

The fact that the downloaded file is added to the store saves bandwidth: when the developer eventually tries to build the newly defined package with `guix build`, the source tarball will not have to be downloaded again because it is already in the store. It is also a convenient way to temporarily stash files, which may be deleted eventually (see Section 3.5 [Invoking guix gc], page 27).

The `guix download` command supports the same URIs as used in package definitions. In particular, it supports `mirror://` URIs. `https` URIs (HTTP over TLS) are supported *provided* the Guile bindings for GnuTLS are available in the user's environment; when they are not available, an error is raised. See Section "Guile Preparations" in *GnuTLS-Guile*, for more information.

`guix download` verifies HTTPS server certificates by loading the certificates of X.509 authorities from the directory pointed to by the `SSL_CERT_DIR` environment variable (see Section 6.2.9 [X.509 Certificates], page 206), unless `--no-check-certificate` is used.

The following options are available:

`--format=`*fmt*
`-f` *fmt*
: Write the hash in the format specified by *fmt*. For more information on the valid values for *fmt*, see Section 5.4 [Invoking guix hash], page 72.

`--no-check-certificate`
: Do not validate the X.509 certificates of HTTPS servers.

  When using this option, you have *absolutely no guarantee* that you are communicating with the authentic server responsible for the given URL, which makes you vulnerable to "man-in-the-middle" attacks.

`--output=`*file*
`-o` *file*
: Save the downloaded file to *file* instead of adding it to the store.

5.4 Invoking `guix hash`

The `guix hash` command computes the SHA256 hash of a file. It is primarily a convenience tool for anyone contributing to the distribution: it computes the cryptographic hash of a file, which can be used in the definition of a package (see Section 4.1 [Defining Packages], page 35).

The general syntax is:

```
guix hash option file
```

When *file* is `-` (a hyphen), `guix hash` computes the hash of data read from standard input. `guix hash` has the following options:

`--format=`*fmt*
`-f` *fmt*
: Write the hash in the format specified by *fmt*.

Supported formats: `nix-base32`, `base32`, `base16` (`hex` and `hexadecimal` can be used as well).

If the `--format` option is not specified, `guix hash` will output the hash in `nix-base32`. This representation is used in the definitions of packages.

`--recursive`
`-r` Compute the hash on *file* recursively.

In this case, the hash is computed on an archive containing *file*, including its children if it is a directory. Some of the metadata of *file* is part of the archive; for instance, when *file* is a regular file, the hash is different depending on whether *file* is executable or not. Metadata such as time stamps has no impact on the hash (see Section 3.8 [Invoking guix archive], page 32).

`--exclude-vcs`
`-x` When combined with `--recursive`, exclude version control system directories (`.bzr`, `.git`, `.hg`, etc.)

As an example, here is how you would compute the hash of a Git checkout, which is useful when using the `git-fetch` method (see Section 4.1.2 [origin Reference], page 40):

```
$ git clone http://example.org/foo.git
$ cd foo
$ guix hash -rx .
```

5.5 Invoking `guix import`

The `guix import` command is useful for people who would like to add a package to the distribution with as little work as possible—a legitimate demand. The command knows of a few repositories from which it can "import" package metadata. The result is a package definition, or a template thereof, in the format we know (see Section 4.1 [Defining Packages], page 35).

The general syntax is:

```
guix import importer options...
```

importer specifies the source from which to import package metadata, and *options* specifies a package identifier and other options specific to *importer*. Currently, the available "importers" are:

`gnu` Import metadata for the given GNU package. This provides a template for the latest version of that GNU package, including the hash of its source tarball, and its canonical synopsis and description.

Additional information such as the package dependencies and its license needs to be figured out manually.

For example, the following command returns a package definition for GNU Hello:

```
guix import gnu hello
```

Specific command-line options are:

`--key-download=`*policy*
: As for `guix refresh`, specify the policy to handle missing OpenPGP keys when verifying the package signature. See Section 5.6 [Invoking guix refresh], page 77.

`pypi`
: Import metadata from the Python Package Index (`https://pypi.python.org/`)[1]. Information is taken from the JSON-formatted description available at `pypi.python.org` and usually includes all the relevant information, including package dependencies. For maximum efficiency, it is recommended to install the `unzip` utility, so that the importer can unzip Python wheels and gather data from them.

  The command below imports metadata for the `itsdangerous` Python package:

  ```
  guix import pypi itsdangerous
  ```

`gem`
: Import metadata from RubyGems (`https://rubygems.org/`)[2]. Information is taken from the JSON-formatted description available at `rubygems.org` and includes most relevant information, including runtime dependencies. There are some caveats, however. The metadata doesn't distinguish between synopses and descriptions, so the same string is used for both fields. Additionally, the details of non-Ruby dependencies required to build native extensions is unavailable and left as an exercise to the packager.

  The command below imports metadata for the `rails` Ruby package:

  ```
  guix import gem rails
  ```

`cpan`
: Import metadata from MetaCPAN (`https://www.metacpan.org/`)[3]. Information is taken from the JSON-formatted metadata provided through MetaCPAN's API (`https://api.metacpan.org/`) and includes most relevant information, such as module dependencies. License information should be checked closely. If Perl is available in the store, then the `corelist` utility will be used to filter core modules out of the list of dependencies.

  The command command below imports metadata for the `Acme::Boolean` Perl module:

  ```
  guix import cpan Acme::Boolean
  ```

`cran`
: Import metadata from CRAN (`http://cran.r-project.org/`), the central repository for the GNU R statistical and graphical environment (`http://r-project.org`).

  Information is extracted from the `DESCRIPTION` file of the package.

  The command command below imports metadata for the `Cairo` R package:

  ```
  guix import cran Cairo
  ```

  When `--recursive` is added, the importer will traverse the dependency graph of the given upstream package recursively and generate package expressions for all those packages that are not yet in Guix.

[1] This functionality requires Guile-JSON to be installed. See Section 2.2 [Requirements], page 5.

[2] This functionality requires Guile-JSON to be installed. See Section 2.2 [Requirements], page 5.

[3] This functionality requires Guile-JSON to be installed. See Section 2.2 [Requirements], page 5.

When `--archive=bioconductor` is added, metadata is imported from Bioconductor (`http://www.bioconductor.org/`), a repository of R packages for for the analysis and comprehension of high-throughput genomic data in bioinformatics.

Information is extracted from the `DESCRIPTION` file of a package published on the web interface of the Bioconductor SVN repository.

The command below imports metadata for the `GenomicRanges` R package:

```
guix import cran --archive=bioconductor GenomicRanges
```

`nix` Import metadata from a local copy of the source of the Nixpkgs distribution (`http://nixos.org/nixpkgs/`)[4]. Package definitions in Nixpkgs are typically written in a mixture of Nix-language and Bash code. This command only imports the high-level package structure that is written in the Nix language. It normally includes all the basic fields of a package definition.

When importing a GNU package, the synopsis and descriptions are replaced by their canonical upstream variant.

Usually, you will first need to do:

```
export NIX_REMOTE=daemon
```

so that `nix-instantiate` does not try to open the Nix database.

As an example, the command below imports the package definition of LibreOffice (more precisely, it imports the definition of the package bound to the `libreoffice` top-level attribute):

```
guix import nix ~/path/to/nixpkgs libreoffice
```

`hackage` Import metadata from the Haskell community's central package archive Hackage (`https://hackage.haskell.org/`). Information is taken from Cabal files and includes all the relevant information, including package dependencies.

Specific command-line options are:

`--stdin`
`-s` Read a Cabal file from standard input.

`--no-test-dependencies`
`-t` Do not include dependencies required only by the test suites.

`--cabal-environment=`*alist*
`-e` *alist* *alist* is a Scheme alist defining the environment in which the Cabal conditionals are evaluated. The accepted keys are: `os`, `arch`, `impl` and a string representing the name of a flag. The value associated with a flag has to be either the symbol `true` or `false`. The value associated with other keys has to conform to the Cabal file format definition. The default value associated with the keys `os`, `arch` and `impl` is '`linux`', '`x86_64`' and '`ghc`', respectively.

The command below imports metadata for the latest version of the `HTTP` Haskell package without including test dependencies and specifying the value of the flag '`network-uri`' as `false`:

[4] This relies on the `nix-instantiate` command of Nix (`http://nixos.org/nix/`).

```
guix import hackage -t -e "'((\"network-uri\" . false))" HTTP
```

A specific package version may optionally be specified by following the package name by an at-sign and a version number as in the following example:

```
guix import hackage mtl@2.1.3.1
```

`stackage` The `stackage` importer is a wrapper around the `hackage` one. It takes a package name, looks up the package version included in a long-term support (LTS) Stackage (`https://www.stackage.org`) release and uses the `hackage` importer to retrieve its metadata. Note that it is up to you to select an LTS release compatible with the GHC compiler used by Guix.

Specific command-line options are:

`--no-test-dependencies`
`-t` Do not include dependencies required only by the test suites.

`--lts-version=`*version*
`-r` *version*
version is the desired LTS release version. If omitted the latest release is used.

The command below imports metadata for the `HTTP` Haskell package included in the LTS Stackage release version 7.18:

```
guix import stackage --lts-version=7.18 HTTP
```

`elpa` Import metadata from an Emacs Lisp Package Archive (ELPA) package repository (see Section "Packages" in *The GNU Emacs Manual*).

Specific command-line options are:

`--archive=`*repo*
`-a` *repo* *repo* identifies the archive repository from which to retrieve the information. Currently the supported repositories and their identifiers are:

- GNU (`http://elpa.gnu.org/packages`), selected by the `gnu` identifier. This is the default.

  Packages from `elpa.gnu.org` are signed with one of the keys contained in the GnuPG keyring at `share/emacs/25.1/etc/package-keyring.gpg` (or similar) in the `emacs` package (see Section "Package Installation" in *The GNU Emacs Manual*).
- MELPA-Stable (`http://stable.melpa.org/packages`), selected by the `melpa-stable` identifier.
- MELPA (`http://melpa.org/packages`), selected by the `melpa` identifier.

`crate` Import metadata from the crates.io Rust package repository crates.io (`https://crates.io`).

The structure of the `guix import` code is modular. It would be useful to have more importers for other package formats, and your help is welcome here (see Chapter 7 [Contributing], page 242).

5.6 Invoking `guix refresh`

The primary audience of the `guix refresh` command is developers of the GNU software distribution. By default, it reports any packages provided by the distribution that are outdated compared to the latest upstream version, like this:

```
$ guix refresh
gnu/packages/gettext.scm:29:13: gettext would be upgraded from 0.18.1.1 to 0.18.2.1
gnu/packages/glib.scm:77:12: glib would be upgraded from 2.34.3 to 2.37.0
```

Alternately, one can specify packages to consider, in which case a warning is emitted for packages that lack an updater:

```
$ guix refresh coreutils guile guile-ssh
gnu/packages/ssh.scm:205:2: warning: no updater for guile-ssh
gnu/packages/guile.scm:136:12: guile would be upgraded from 2.0.12 to 2.0.13
```

`guix refresh` browses the upstream repository of each package and determines the highest version number of the releases therein. The command knows how to update specific types of packages: GNU packages, ELPA packages, etc.—see the documentation for `--type` below. There are many packages, though, for which it lacks a method to determine whether a new upstream release is available. However, the mechanism is extensible, so feel free to get in touch with us to add a new method!

When passed `--update`, it modifies distribution source files to update the version numbers and source tarball hashes of those package recipes (see Section 4.1 [Defining Packages], page 35). This is achieved by downloading each package's latest source tarball and its associated OpenPGP signature, authenticating the downloaded tarball against its signature using `gpg`, and finally computing its hash. When the public key used to sign the tarball is missing from the user's keyring, an attempt is made to automatically retrieve it from a public key server; when this is successful, the key is added to the user's keyring; otherwise, `guix refresh` reports an error.

The following options are supported:

`--expression=`*expr*
`-e` *expr* Consider the package *expr* evaluates to.

This is useful to precisely refer to a package, as in this example:

```
guix refresh -l -e '(@@ (gnu packages commencement) glibc-final)'
```

This command lists the dependents of the "final" libc (essentially all the packages.)

`--update`
`-u` Update distribution source files (package recipes) in place. This is usually run from a checkout of the Guix source tree (see Section 7.2 [Running Guix Before It Is Installed], page 243):

```
$ ./pre-inst-env guix refresh -s non-core -u
```

See Section 4.1 [Defining Packages], page 35, for more information on package definitions.

`--select=[`*subset*`]`
`-s` *subset* Select all the packages in *subset*, one of `core` or `non-core`.

The `core` subset refers to all the packages at the core of the distribution—i.e., packages that are used to build "everything else". This includes GCC, libc, Binutils, Bash, etc. Usually, changing one of these packages in the distribution entails a rebuild of all the others. Thus, such updates are an inconvenience to users in terms of build time or bandwidth used to achieve the upgrade.

The `non-core` subset refers to the remaining packages. It is typically useful in cases where an update of the core packages would be inconvenient.

`--type=`*updater*
`-t` *updater*

Select only packages handled by *updater* (may be a comma-separated list of updaters). Currently, *updater* may be one of:

`gnu` the updater for GNU packages;

`gnome` the updater for GNOME packages;

`kde` the updater for KDE packages;

`xorg` the updater for X.org packages;

`kernel.org` the updater for packages hosted on kernel.org;

`elpa` the updater for ELPA (`http://elpa.gnu.org/`) packages;

`cran` the updater for CRAN (`http://cran.r-project.org/`) packages;

`bioconductor` the updater for Bioconductor (`http://www.bioconductor.org/`) R packages;

`cpan` the updater for CPAN (`http://www.cpan.org/`) packages;

`pypi` the updater for PyPI (`https://pypi.python.org`) packages.

`gem` the updater for RubyGems (`https://rubygems.org`) packages.

`github` the updater for GitHub (`https://github.com`) packages.

`hackage` the updater for Hackage (`https://hackage.haskell.org`) packages.

`stackage` the updater for Stackage (`https://www.stackage.org`) packages.

`crate` the updater for Crates (`https://crates.io`) packages.

For instance, the following command only checks for updates of Emacs packages hosted at `elpa.gnu.org` and for updates of CRAN packages:

```
$ guix refresh --type=elpa,cran
gnu/packages/statistics.scm:819:13: r-testthat would be upgraded from 0.10.0
gnu/packages/emacs.scm:856:13: emacs-auctex would be upgraded from 11.88.6 t
```

In addition, `guix refresh` can be passed one or more package names, as in this example:

```
$ ./pre-inst-env guix refresh -u emacs idutils gcc@4.8
```

The command above specifically updates the `emacs` and `idutils` packages. The `--select` option would have no effect in this case.

When considering whether to upgrade a package, it is sometimes convenient to know which packages would be affected by the upgrade and should be checked for compatibility. For this the following option may be used when passing `guix refresh` one or more package names:

`--list-updaters`
`-L`
: List available updaters and exit (see `--type` above.)

  For each updater, display the fraction of packages it covers; at the end, display the fraction of packages covered by all these updaters.

`--list-dependent`
`-l`
: List top-level dependent packages that would need to be rebuilt as a result of upgrading one or more packages.

  See Section 5.9 [Invoking guix graph], page 82, for information on how to visualize the list of dependents of a package.

Be aware that the `--list-dependent` option only *approximates* the rebuilds that would be required as a result of an upgrade. More rebuilds might be required under some circumstances.

```
$ guix refresh --list-dependent flex
Building the following 120 packages would ensure 213 dependent packages are rebuilt:
hop@2.4.0 geiser@0.4 notmuch@0.18 mu@0.9.9.5 cflow@1.4 idutils@4.6 ...
```

The command above lists a set of packages that could be built to check for compatibility with an upgraded `flex` package.

The following options can be used to customize GnuPG operation:

`--gpg=`*command*
: Use *command* as the GnuPG 2.x command. *command* is searched for in `$PATH`.

`--key-download=`*policy*
: Handle missing OpenPGP keys according to *policy*, which may be one of:

  `always`
  : Always download missing OpenPGP keys from the key server, and add them to the user's GnuPG keyring.

  `never`
  : Never try to download missing OpenPGP keys. Instead just bail out.

  `interactive`
  : When a package signed with an unknown OpenPGP key is encountered, ask the user whether to download it or not. This is the default behavior.

`--key-server=`*host*
: Use *host* as the OpenPGP key server when importing a public key.

The `github` updater uses the GitHub API (`https://developer.github.com/v3/`) to query for new releases. When used repeatedly e.g. when refreshing all packages, GitHub will eventually refuse to answer any further API requests. By default 60 API requests per hour are allowed, and a full refresh on all GitHub packages in Guix requires more than this. Authentication with GitHub through the use of an API token alleviates these limits. To use an API token, set the environment variable `GUIX_GITHUB_TOKEN` to a token procured from `https://github.com/settings/tokens` or otherwise.

5.7 Invoking `guix lint`

The `guix lint` command is meant to help package developers avoid common errors and use a consistent style. It runs a number of checks on a given set of packages in order to find common mistakes in their definitions. Available *checkers* include (see `--list-checkers` for a complete list):

`synopsis`
`description`
: Validate certain typographical and stylistic rules about package descriptions and synopses.

`inputs-should-be-native`
: Identify inputs that should most likely be native inputs.

`source`
`home-page`
`mirror-url`
`source-file-name`
: Probe `home-page` and `source` URLs and report those that are invalid. Suggest a `mirror://` URL when applicable. Check that the source file name is meaningful, e.g. is not just a version number or "git-checkout", without a declared `file-name` (see Section 4.1.2 [origin Reference], page 40).

`cve`
: Report known vulnerabilities found in the Common Vulnerabilities and Exposures (CVE) databases of the current and past year published by the US NIST (`https://nvd.nist.gov/download.cfm#CVE_FEED`).

  To view information about a particular vulnerability, visit pages such as:

  - '`https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-YYYY-ABCD`'
  - '`https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-YYYY-ABCD`'

  where `CVE-YYYY-ABCD` is the CVE identifier—e.g., `CVE-2015-7554`.

  Package developers can specify in package recipes the Common Platform Enumeration (CPE) (`https://nvd.nist.gov/cpe.cfm`) name and version of the package when they differ from the name that Guix uses, as in this example:

  ```
  (package
    (name "grub")
    ;; ...
    ;; CPE calls this package "grub2".
    (properties '((cpe-name . "grub2"))))
  ```

`formatting`
: Warn about obvious source code formatting issues: trailing white space, use of tabulations, etc.

The general syntax is:

```
guix lint options package...
```

If no package is given on the command line, then all packages are checked. The *options* may be zero or more of the following:

`--list-checkers`
`-l` List and describe all the available checkers that will be run on packages and exit.

`--checkers`
`-c` Only enable the checkers specified in a comma-separated list using the names returned by `--list-checkers`.

5.8 Invoking `guix size`

The `guix size` command helps package developers profile the disk usage of packages. It is easy to overlook the impact of an additional dependency added to a package, or the impact of using a single output for a package that could easily be split (see Section 3.4 [Packages with Multiple Outputs], page 27). Such are the typical issues that `guix size` can highlight.

The command can be passed a package specification such as `gcc@4.8` or `guile:debug`, or a file name in the store. Consider this example:

```
$ guix size coreutils
store item                                 total    self
/gnu/store/...-coreutils-8.23           70.0    13.9  19.8%
/gnu/store/...-gmp-6.0.0a               55.3     2.5   3.6%
/gnu/store/...-acl-2.2.52               53.7     0.5   0.7%
/gnu/store/...-attr-2.4.46              53.2     0.3   0.5%
/gnu/store/...-gcc-4.8.4-lib            52.9    15.7  22.4%
/gnu/store/...-glibc-2.21               37.2    37.2  53.1%
```

The store items listed here constitute the *transitive closure* of Coreutils—i.e., Coreutils and all its dependencies, recursively—as would be returned by:

```
$ guix gc -R /gnu/store/...-coreutils-8.23
```

Here the output shows three columns next to store items. The first column, labeled "total", shows the size in mebibytes (MiB) of the closure of the store item—that is, its own size plus the size of all its dependencies. The next column, labeled "self", shows the size of the item itself. The last column shows the ratio of the size of the item itself to the space occupied by all the items listed here.

In this example, we see that the closure of Coreutils weighs in at 70 MiB, half of which is taken by libc. (That libc represents a large fraction of the closure is not a problem *per se* because it is always available on the system anyway.)

When the package passed to `guix size` is available in the store, `guix size` queries the daemon to determine its dependencies, and measures its size in the store, similar to `du -ms --apparent-size` (see Section "du invocation" in *GNU Coreutils*).

When the given package is *not* in the store, `guix size` reports information based on the available substitutes (see Section 3.3 [Substitutes], page 25). This makes it possible it to profile disk usage of store items that are not even on disk, only available remotely.

You can also specify several package names:

```
$ guix size coreutils grep sed bash
store item                                 total    self
/gnu/store/...-coreutils-8.24           77.8    13.8  13.4%
```

```
/gnu/store/...-grep-2.22                73.1      0.8    0.8%
/gnu/store/...-bash-4.3.42              72.3      4.7    4.6%
/gnu/store/...-readline-6.3             67.6      1.2    1.2%
...
total: 102.3 MiB
```

In this example we see that the combination of the four packages takes 102.3 MiB in total, which is much less than the sum of each closure since they have a lot of dependencies in common.

The available options are:

`--substitute-urls=`*urls*
: Use substitute information from *urls*. See [client-substitute-urls], page 65.

`--map-file=`*file*
: Write a graphical map of disk usage in PNG format to *file*.

  For the example above, the map looks like this:

  This option requires that Guile-Charting (`http://wingolog.org/software/guile-charting/`) be installed and visible in Guile's module search path. When that is not the case, `guix size` fails as it tries to load it.

`--system=`*system*
`-s` *system* Consider packages for *system*—e.g., `x86_64-linux`.

5.9 Invoking `guix graph`

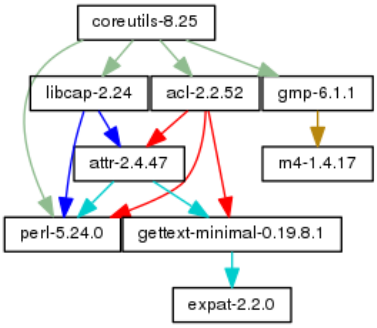
Packages and their dependencies form a *graph*, specifically a directed acyclic graph (DAG). It can quickly become difficult to have a mental model of the package DAG, so the `guix graph` command provides a visual representation of the DAG. By default, `guix graph` emits a DAG representation in the input format of Graphviz (`http://www.graphviz.org/`), so its output can be passed directly to the `dot` command of Graphviz. It can also emit an HTML page with embedded JavaScript code to display a "chord diagram" in a Web browser, using the d3.js (`https://d3js.org/`) library, or emit Cypher queries to construct a graph in a graph database supporting the openCypher (`http://www.opencypher.org/`) query language. The general syntax is:

```
guix graph options package...
```

For example, the following command generates a PDF file representing the package DAG for the GNU Core Utilities, showing its build-time dependencies:

```
guix graph coreutils | dot -Tpdf > dag.pdf
```

The output looks like this:



Nice little graph, no?

But there is more than one graph! The one above is concise: it is the graph of package objects, omitting implicit inputs such as GCC, libc, grep, etc. It is often useful to have such a concise graph, but sometimes one may want to see more details. `guix graph` supports several types of graphs, allowing you to choose the level of detail:

`package` This is the default type used in the example above. It shows the DAG of package objects, excluding implicit dependencies. It is concise, but filters out many details.

`reverse-package`
This shows the *reverse* DAG of packages. For example:

```
guix graph --type=reverse-package ocaml
```

... yields the graph of packages that depend on OCaml.

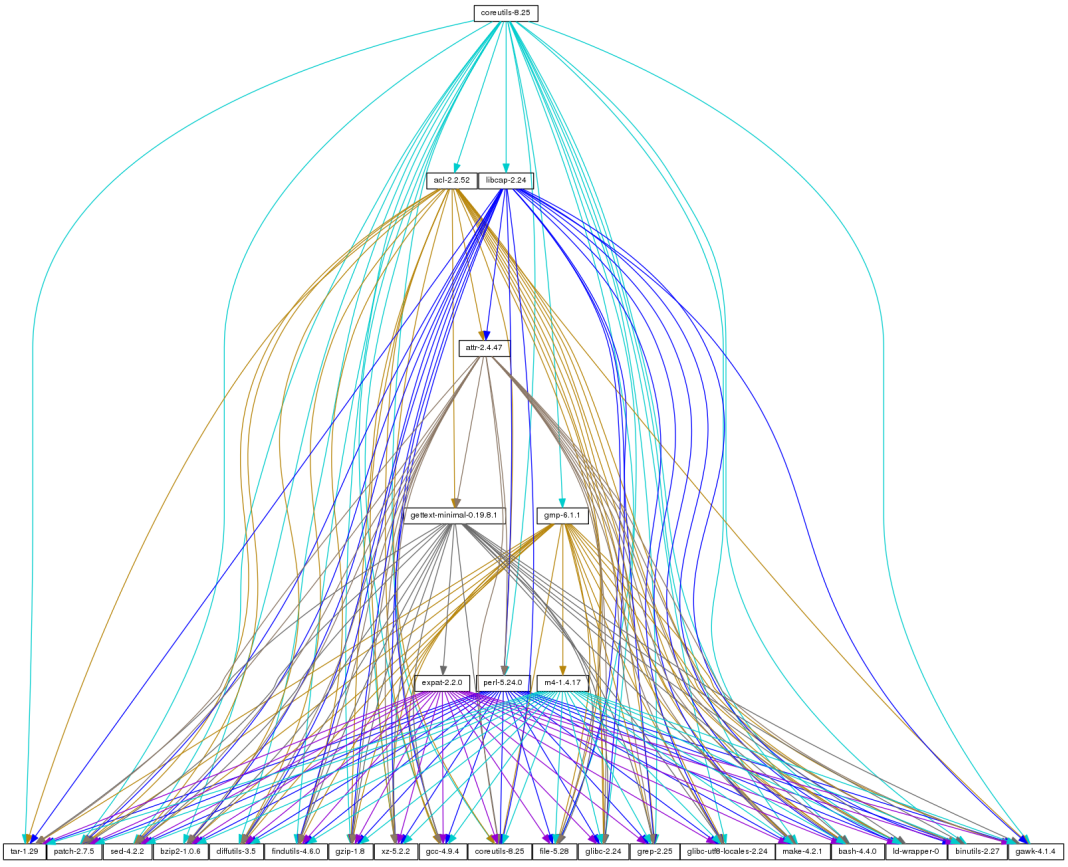
Note that for core packages this can yield huge graphs. If all you want is to know the number of packages that depend on a given package, use `guix refresh --list-dependent` (see Section 5.6 [Invoking guix refresh], page 77).

`bag-emerged`
This is the package DAG, *including* implicit inputs.

For instance, the following command:

```
guix graph --type=bag-emerged coreutils | dot -Tpdf > dag.pdf
```

... yields this bigger graph:



At the bottom of the graph, we see all the implicit inputs of *gnu-build-system* (see Section 4.2 [Build Systems], page 41).

Now, note that the dependencies of these implicit inputs—that is, the *bootstrap dependencies* (see Section 6.8 [Bootstrapping], page 237)—are not shown here, for conciseness.

`bag` Similar to `bag-emerged`, but this time including all the bootstrap dependencies.

`bag-with-origins`
Similar to `bag`, but also showing origins and their dependencies.

`derivations`
This is the most detailed representation: It shows the DAG of derivations (see Section 4.4 [Derivations], page 50) and plain store items. Compared to the above representation, many additional nodes are visible, including build scripts, patches, Guile modules, etc.

For this type of graph, it is also possible to pass a `.drv` file name instead of a package name, as in:

```
guix graph -t derivation ‘guix system build -d my-config.scm‘
```

All the types above correspond to *build-time dependencies*. The following graph type represents the *run-time dependencies*:

`references`
: This is the graph of *references* of a package output, as returned by `guix gc --references` (see Section 3.5 [Invoking guix gc], page 27).

  If the given package output is not available in the store, `guix graph` attempts to obtain dependency information from substitutes.

  Here you can also pass a store file name instead of a package name. For example, the command below produces the reference graph of your profile (which can be big!):

  ```
  guix graph -t references ‘readlink -f ~/.guix-profile‘
  ```

`referrers`
: This is the graph of the *referrers* of a store item, as returned by `guix gc --referrers` (see Section 3.5 [Invoking guix gc], page 27).

  This relies exclusively on local information from your store. For instance, let us suppose that the current Inkscape is available in 10 profiles on your machine; `guix graph -t referrers inkscape` will show a graph rooted at Inkscape and with those 10 profiles linked to it.

  It can help determine what is preventing a store item from being garbage collected.

The available options are the following:

`--type=`*type*
`-t` *type*
: Produce a graph output of *type*, where *type* must be one of the values listed above.

`--list-types`
: List the supported graph types.

`--backend=`*backend*
`-b` *backend*
: Produce a graph using the selected *backend*.

`--list-backends`
: List the supported graph backends.

  Currently, the available backends are Graphviz and d3.js.

`--expression=`*expr*
`-e` *expr*
: Consider the package *expr* evaluates to.

  This is useful to precisely refer to a package, as in this example:

  ```
  guix graph -e '(@@ (gnu packages commencement) gnu-make-final)'
  ```

5.10 Invoking `guix environment`

The purpose of `guix environment` is to assist hackers in creating reproducible development environments without polluting their package profile. The `guix environment` tool takes one or more packages, builds all of their inputs, and creates a shell environment to use them.

The general syntax is:

```
guix environment options package...
```

The following example spawns a new shell set up for the development of GNU Guile:

```
guix environment guile
```

If the needed dependencies are not built yet, `guix environment` automatically builds them. The environment of the new shell is an augmented version of the environment that `guix environment` was run in. It contains the necessary search paths for building the given package added to the existing environment variables. To create a "pure" environment, in which the original environment variables have been unset, use the `--pure` option[5].

`guix environment` defines the `GUIX_ENVIRONMENT` variable in the shell it spawns; its value is the file name of the profile of this environment. This allows users to, say, define a specific prompt for development environments in their `.bashrc` (see Section "Bash Startup Files" in *The GNU Bash Reference Manual*):

```
if [ -n "$GUIX_ENVIRONMENT" ]
then
    export PS1="\u@\h \w [dev]\$ "
fi
```

... or to browse the profile:

```
$ ls "$GUIX_ENVIRONMENT/bin"
```

Additionally, more than one package may be specified, in which case the union of the inputs for the given packages are used. For example, the command below spawns a shell where all of the dependencies of both Guile and Emacs are available:

```
guix environment guile emacs
```

Sometimes an interactive shell session is not desired. An arbitrary command may be invoked by placing the `--` token to separate the command from the rest of the arguments:

```
guix environment guile -- make -j4
```

In other situations, it is more convenient to specify the list of packages needed in the environment. For example, the following command runs `python` from an environment containing Python 2.7 and NumPy:

```
guix environment --ad-hoc python2-numpy python-2.7 -- python
```

Furthermore, one might want the dependencies of a package and also some additional packages that are not build-time or runtime dependencies, but are useful when developing nonetheless. Because of this, the `--ad-hoc` flag is positional. Packages appearing before

[5] Users sometimes wrongfully augment environment variables such as `PATH` in their `~/.bashrc` file. As a consequence, when `guix environment` launches it, Bash may read `~/.bashrc`, thereby introducing "impurities" in these environment variables. It is an error to define such environment variables in `.bashrc`; instead, they should be defined in `.bash_profile`, which is sourced only by log-in shells. See Section "Bash Startup Files" in *The GNU Bash Reference Manual*, for details on Bash start-up files.

`--ad-hoc` are interpreted as packages whose dependencies will be added to the environment. Packages appearing after are interpreted as packages that will be added to the environment directly. For example, the following command creates a Guix development environment that additionally includes Git and strace:

```
guix environment guix --ad-hoc git strace
```

Sometimes it is desirable to isolate the environment as much as possible, for maximal purity and reproducibility. In particular, when using Guix on a host distro that is not GuixSD, it is desirable to prevent access to `/usr/bin` and other system-wide resources from the development environment. For example, the following command spawns a Guile REPL in a "container" where only the store and the current working directory are mounted:

```
guix environment --ad-hoc --container guile -- guile
```

Note: The `--container` option requires Linux-libre 3.19 or newer.

The available options are summarized below.

`--root=`*file*
`-r` *file* Make *file* a symlink to the profile for this environment, and register it as a garbage collector root.

This is useful if you want to protect your environment from garbage collection, to make it "persistent".

When this option is omitted, the environment is protected from garbage collection only for the duration of the `guix environment` session. This means that next time you recreate the same environment, you could have to rebuild or re-download packages.

`--expression=`*expr*
`-e` *expr* Create an environment for the package or list of packages that *expr* evaluates to.

For example, running:

```
guix environment -e '(@ (gnu packages maths) petsc-openmpi)'
```

starts a shell with the environment for this specific variant of the PETSc package.

Running:

```
guix environment --ad-hoc -e '(@ (gnu) %base-packages)'
```

starts a shell with all the GuixSD base packages available.

The above commands only use the default output of the given packages. To select other outputs, two element tuples can be specified:

```
guix environment --ad-hoc -e '(list ( (gnu packages bash) bash) "include")'
```

`--load=`*file*
`-l` *file* Create an environment for the package or list of packages that the code within *file* evaluates to.

As an example, *file* might contain a definition like this (see Section 4.1 [Defining Packages], page 35):

```
(use-modules (guix)
             (gnu packages gdb)
```

```
                        (gnu packages autotools)
                        (gnu packages texinfo))

              ;; Augment the package definition of GDB with the build tools
              ;; needed when developing GDB (and which are not needed when
              ;; simply installing it.)
              (package (inherit gdb)
                (native-inputs `(("autoconf" ,autoconf-2.64)
                                 ("automake" ,automake)
                                 ("texinfo" ,texinfo)
                                 ,@(package-native-inputs gdb))))
```

`--ad-hoc` Include all specified packages in the resulting environment, as if an *ad hoc* package were defined with them as inputs. This option is useful for quickly creating an environment without having to write a package expression to contain the desired inputs.

For instance, the command:

```
guix environment --ad-hoc guile guile-sdl -- guile
```

runs `guile` in an environment where Guile and Guile-SDL are available.

Note that this example implicitly asks for the default output of `guile` and `guile-sdl`, but it is possible to ask for a specific output—e.g., `glib:bin` asks for the `bin` output of `glib` (see Section 3.4 [Packages with Multiple Outputs], page 27).

This option may be composed with the default behavior of `guix environment`. Packages appearing before `--ad-hoc` are interpreted as packages whose dependencies will be added to the environment, the default behavior. Packages appearing after are interpreted as packages that will be added to the environment directly.

`--pure` Unset existing environment variables when building the new environment. This has the effect of creating an environment in which search paths only contain package inputs.

`--search-paths`
Display the environment variable definitions that make up the environment.

`--system=`*system*
`-s` *system* Attempt to build for *system*—e.g., `i686-linux`.

`--container`
`-C` Run *command* within an isolated container. The current working directory outside the container is mapped inside the container. Additionally, a dummy home directory is created that matches the current user's home directory, and `/etc/passwd` is configured accordingly. The spawned process runs as the current user outside the container, but has root privileges in the context of the container.

`--network`
`-N` For containers, share the network namespace with the host system. Containers created without this flag only have access to the loopback device.

`--expose=`*source*`[=`*target*`]`

For containers, expose the file system *source* from the host system as the read-only file system *target* within the container. If *target* is not specified, *source* is used as the target mount point in the container.

The example below spawns a Guile REPL in a container in which the user's home directory is accessible read-only via the `/exchange` directory:

```
guix environment --container --expose=$HOME=/exchange --ad-hoc guile -- guil
```

`--share=`*source*`[=`*target*`]`

For containers, share the file system *source* from the host system as the writable file system *target* within the container. If *target* is not specified, *source* is used as the target mount point in the container.

The example below spawns a Guile REPL in a container in which the user's home directory is accessible for both reading and writing via the `/exchange` directory:

```
guix environment --container --share=$HOME=/exchange --ad-hoc guile -- guile
```

`guix environment` also supports all of the common build options that `guix build` supports (see Section 5.1.1 [Common Build Options], page 64).

5.11 Invoking `guix publish`

The purpose of `guix publish` is to enable users to easily share their store with others, who can then use it as a substitute server (see Section 3.3 [Substitutes], page 25).

When `guix publish` runs, it spawns an HTTP server which allows anyone with network access to obtain substitutes from it. This means that any machine running Guix can also act as if it were a build farm, since the HTTP interface is compatible with Hydra, the software behind the `hydra.gnu.org` build farm.

For security, each substitute is signed, allowing recipients to check their authenticity and integrity (see Section 3.3 [Substitutes], page 25). Because `guix publish` uses the signing key of the system, which is only readable by the system administrator, it must be started as root; the `--user` option makes it drop root privileges early on.

The signing key pair must be generated before `guix publish` is launched, using `guix archive --generate-key` (see Section 3.8 [Invoking guix archive], page 32).

The general syntax is:

```
guix publish options...
```

Running `guix publish` without any additional arguments will spawn an HTTP server on port 8080:

```
guix publish
```

Once a publishing server has been authorized (see Section 3.8 [Invoking guix archive], page 32), the daemon may download substitutes from it:

```
guix-daemon --substitute-urls=http://example.org:8080
```

By default, `guix publish` compresses archives on the fly as it serves them. This "on-the-fly" mode is convenient in that it requires no setup and is immediately available. However, when serving lots of clients, we recommend using the `--cache` option, which enables caching of the archives before they are sent to clients—see below for details.

As a bonus, `guix publish` also serves as a content-addressed mirror for source files referenced in `origin` records (see Section 4.1.2 [origin Reference], page 40). For instance, assuming `guix publish` is running on `example.org`, the following URL returns the raw `hello-2.10.tar.gz` file with the given SHA256 hash (represented in `nix-base32` format, see Section 5.4 [Invoking guix hash], page 72):

```
http://example.org/file/hello-2.10.tar.gz/sha256/0ssi1...ndq1i
```

Obviously, these URLs only work for files that are in the store; in other cases, they return 404 ("Not Found").

The following options are available:

`--port=`*port*
`-p` *port* Listen for HTTP requests on *port*.

`--listen=`*host*
: Listen on the network interface for *host*. The default is to accept connections from any interface.

`--user=`*user*
`-u` *user* Change privileges to *user* as soon as possible—i.e., once the server socket is open and the signing key has been read.

`--compression[=`*level*`]`
`-C [`*level*`]`
: Compress data using the given *level*. When *level* is zero, disable compression. The range 1 to 9 corresponds to different gzip compression levels: 1 is the fastest, and 9 is the best (CPU-intensive). The default is 3.

  Unless `--cache` is used, compression occurs on the fly and the compressed streams are not cached. Thus, to reduce load on the machine that runs `guix publish`, it may be a good idea to choose a low compression level, to run `guix publish` behind a caching proxy, or to use `--cache`. Using `--cache` has the advantage that it allows `guix publish` to add `Content-Length` HTTP header to its responses.

`--cache=`*directory*
`-c` *directory*
: Cache archives and meta-data (`.narinfo` URLs) to *directory* and only serve archives that are in cache.

  When this option is omitted, archives and meta-data are created on-the-fly. This can reduce the available bandwidth, especially when compression is enabled, since this may become CPU-bound. Another drawback of the default mode is that the length of archives is not known in advance, so `guix publish` does not add a `Content-Length` HTTP header to its responses, which in turn prevents clients from knowing the amount of data being downloaded.

  Conversely, when `--cache` is used, the first request for a store item (*via* a `.narinfo` URL) returns 404 and triggers a background process to *bake* the archive—computing its `.narinfo` and compressing the archive, if needed. Once the archive is cached in *directory*, subsequent requests succeed and are served directly from the cache, which guarantees that clients get the best possible bandwidth.

The "baking" process is performed by worker threads. By default, one thread per CPU core is created, but this can be customized. See `--workers` below.

When `--ttl` is used, cached entries are automatically deleted when they have expired.

`--workers=`*N*
: When `--cache` is used, request the allocation of *N* worker threads to "bake" archives.

`--ttl=`*ttl*
: Produce `Cache-Control` HTTP headers that advertise a time-to-live (TTL) of *ttl*. *ttl* must denote a duration: `5d` means 5 days, `1m` means 1 month, and so on.

  This allows the user's Guix to keep substitute information in cache for *ttl*. However, note that `guix publish` does not itself guarantee that the store items it provides will indeed remain available for as long as *ttl*.

  Additionally, when `--cache` is used, cached entries that have not been accessed for *ttl* may be deleted.

`--nar-path=`*path*
: Use *path* as the prefix for the URLs of "nar" files (see Section 3.8 [Invoking guix archive], page 32).

  By default, nars are served at a URL such as `/nar/gzip/...-coreutils-8.25`. This option allows you to change the `/nar` part to *path*.

`--public-key=`*file*
`--private-key=`*file*
: Use the specific *file*s as the public/private key pair used to sign the store items being published.

  The files must correspond to the same key pair (the private key is used for signing and the public key is merely advertised in the signature metadata). They must contain keys in the canonical s-expression format as produced by `guix archive --generate-key` (see Section 3.8 [Invoking guix archive], page 32). By default, `/etc/guix/signing-key.pub` and `/etc/guix/signing-key.sec` are used.

`--repl[=`*port*`]`
`-r [`*port*`]`
: Spawn a Guile REPL server (see Section "REPL Servers" in *GNU Guile Reference Manual*) on *port* (37146 by default). This is used primarily for debugging a running `guix publish` server.

Enabling `guix publish` on a GuixSD system is a one-liner: just instantiate a `guix-publish-service-type` service in the `services` field of the `operating-system` declaration (see [guix-publish-service-type], page 128).

If you are instead running Guix on a "foreign distro", follow these instructions:"

- If your host distro uses the systemd init system:

```
# ln -s ~root/.guix-profile/lib/systemd/system/guix-publish.service \
        /etc/systemd/system/
# systemctl start guix-publish && systemctl enable guix-publish
```

- If your host distro uses the Upstart init system:

```
# ln -s ~root/.guix-profile/lib/upstart/system/guix-publish.conf /etc/init/
# start guix-publish
```

- Otherwise, proceed similarly with your distro's init system.

5.12 Invoking `guix challenge`

Do the binaries provided by this server really correspond to the source code it claims to build? Is a package build process deterministic? These are the questions the `guix challenge` command attempts to answer.

The former is obviously an important question: Before using a substitute server (see Section 3.3 [Substitutes], page 25), one had better *verify* that it provides the right binaries, and thus *challenge* it. The latter is what enables the former: If package builds are deterministic, then independent builds of the package should yield the exact same result, bit for bit; if a server provides a binary different from the one obtained locally, it may be either corrupt or malicious.

We know that the hash that shows up in `/gnu/store` file names is the hash of all the inputs of the process that built the file or directory—compilers, libraries, build scripts, etc. (see Chapter 1 [Introduction], page 2). Assuming deterministic build processes, one store file name should map to exactly one build output. `guix challenge` checks whether there is, indeed, a single mapping by comparing the build outputs of several independent builds of any given store item.

The command output looks like this:

```
$ guix challenge --substitute-urls="https://hydra.gnu.org https://guix.example.org"
updating list of substitutes from 'https://hydra.gnu.org'... 100.0%
updating list of substitutes from 'https://guix.example.org'... 100.0%
/gnu/store/...-openssl-1.0.2d contents differ:
  local hash: 0725l22r5jnzazaacncwsvp9kgf42266ayyp814v7djxs7nk963q
  https://hydra.gnu.org/nar/...-openssl-1.0.2d: 0725l22r5jnzazaacncwsvp9kgf42266ayyp814v7djxs7nk963q
  https://guix.example.org/nar/...-openssl-1.0.2d: 1zy4fmaaqcnjrzzajkdn3f5gmjk754b43qkq47llbyak9z0qjyim
/gnu/store/...-git-2.5.0 contents differ:
  local hash: 00p3bmryhjxrhpn2gxs2fy0a15lnip05l97205pgbk5ra395hyha
  https://hydra.gnu.org/nar/...-git-2.5.0: 069nb85bv4d4a6slrwjdy8v1cn4cwspm3kdbmyb81d6zckj3nq9f
  https://guix.example.org/nar/...-git-2.5.0: 0mdqa9w1p6cmli6976v4wi0sw9r4p5prkj7lzfd1877wk11c9c73
/gnu/store/...-pius-2.1.1 contents differ:
  local hash: 0k4v3m9z1zp8xzzizb7d8kjj72f9172xv078sq4wl73vnq9ig3ax
  https://hydra.gnu.org/nar/...-pius-2.1.1: 0k4v3m9z1zp8xzzizb7d8kjj72f9172xv078sq4wl73vnq9ig3ax
  https://guix.example.org/nar/...-pius-2.1.1: 1cy25x1a4fzq5rk0pmvc8xhwyffnqz95h2bpvqsz2mpvlbccy0gs
```

In this example, `guix challenge` first scans the store to determine the set of locally-built derivations—as opposed to store items that were downloaded from a substitute server—and then queries all the substitute servers. It then reports those store items for which the servers obtained a result different from the local build.

As an example, `guix.example.org` always gets a different answer. Conversely, `hydra.gnu.org` agrees with local builds, except in the case of Git. This might indicate that the build process of Git is non-deterministic, meaning that its output varies as a function of various things that Guix does not fully control, in spite of building packages in isolated environments (see Section 3.1 [Features], page 17). Most common sources of non-determinism include the addition of timestamps in build results, the inclusion

of random numbers, and directory listings sorted by inode number. See `https://reproducible-builds.org/docs/`, for more information.

To find out what is wrong with this Git binary, we can do something along these lines (see Section 3.8 [Invoking guix archive], page 32):

```
$ wget -q -O - https://hydra.gnu.org/nar/...-git-2.5.0 \
   | guix archive -x /tmp/git
$ diff -ur --no-dereference /gnu/store/...-git.2.5.0 /tmp/git
```

This command shows the difference between the files resulting from the local build, and the files resulting from the build on `hydra.gnu.org` (see Section "Overview" in *Comparing and Merging Files*). The `diff` command works great for text files. When binary files differ, a better option is Diffoscope (`https://diffoscope.org/`), a tool that helps visualize differences for all kinds of files.

Once you have done that work, you can tell whether the differences are due to a non-deterministic build process or to a malicious server. We try hard to remove sources of non-determinism in packages to make it easier to verify substitutes, but of course, this is a process that involves not just Guix, but a large part of the free software community. In the meantime, `guix challenge` is one tool to help address the problem.

If you are writing packages for Guix, you are encouraged to check whether `hydra.gnu.org` and other substitute servers obtain the same build result as you did with:

```
$ guix challenge package
```

where *package* is a package specification such as `guile@2.0` or `glibc:debug`.

The general syntax is:

```
guix challenge options [packages...]
```

When a difference is found between the hash of a locally-built item and that of a server-provided substitute, or among substitutes provided by different servers, the command displays it as in the example above and its exit code is 2 (other non-zero exit codes denote other kinds of errors.)

The one option that matters is:

`--substitute-urls=urls`
: Consider *urls* the whitespace-separated list of substitute source URLs to compare to.

`--verbose`
`-v`
: Show details about matches (identical contents) in addition to information about mismatches.

5.13 Invoking `guix copy`

The `guix copy` command copies items from the store of one machine to that of another machine over a secure shell (SSH) connection[6]. For example, the following command copies the `coreutils` package, the user's profile, and all their dependencies over to *host*, logged in as *user*:

```
guix copy --to=user@host \
```

[6] This command is available only when Guile-SSH was found. See Section 2.2 [Requirements], page 5, for details.

```
coreutils `readlink -f ~/.guix-profile`
```

If some of the items to be copied are already present on *host*, they are not actually sent.

The command below retrieves `libreoffice` and `gimp` from *host*, assuming they are available there:

```
guix copy --from=host libreoffice gimp
```

The SSH connection is established using the Guile-SSH client, which is compatible with OpenSSH: it honors `~/.ssh/known_hosts` and `~/.ssh/config`, and uses the SSH agent for authentication.

The key used to sign items that are sent must be accepted by the remote machine. Likewise, the key used by the remote machine to sign items you are retrieving must be in `/etc/guix/acl` so it is accepted by your own daemon. See Section 3.8 [Invoking guix archive], page 32, for more information about store item authentication.

The general syntax is:

```
guix copy [--to=spec|--from=spec] items...
```

You must always specify one of the following options:

`--to=`*spec*
`--from=`*spec*
: Specify the host to send to or receive from. *spec* must be an SSH spec such as `example.org`, `charlie@example.org`, or `charlie@example.org:2222`.

The *items* can be either package names, such as `gimp`, or store items, such as `/gnu/store/...-idutils-4.6`.

When specifying the name of a package to send, it is first built if needed, unless `--dry-run` was specified. Common build options are supported (see Section 5.1.1 [Common Build Options], page 64).

5.14 Invoking `guix container`

> **Note:** As of version 0.13.0, this tool is experimental. The interface is subject to radical change in the future.

The purpose of `guix container` is to manipulate processes running within an isolated environment, commonly known as a "container", typically created by the `guix environment` (see Section 5.10 [Invoking guix environment], page 86) and `guix system container` (see Section 6.2.13 [Invoking guix system], page 214) commands.

The general syntax is:

```
guix container action options...
```

action specifies the operation to perform with a container, and *options* specifies the context-specific arguments for the action.

The following actions are available:

`exec`
: Execute a command within the context of a running container.

  The syntax is:

  ```
  guix container exec pid program arguments...
  ```

pid specifies the process ID of the running container. *program* specifies an executable file name within the root file system of the container. *arguments* are the additional options that will be passed to *program*.

The following command launches an interactive login shell inside a GuixSD container, started by `guix system container`, and whose process ID is 9001:

```
guix container exec 9001 /run/current-system/profile/bin/bash --login
```

Note that the *pid* cannot be the parent process of a container. It must be PID 1 of the container or one of its child processes.

6 GNU Distribution

Guix comes with a distribution of the GNU system consisting entirely of free software[1]. The distribution can be installed on its own (see Section 6.1 [System Installation], page 96), but it is also possible to install Guix as a package manager on top of an installed GNU/Linux system (see Chapter 2 [Installation], page 3). To distinguish between the two, we refer to the standalone distribution as the Guix System Distribution, or GuixSD.

The distribution provides core GNU packages such as GNU libc, GCC, and Binutils, as well as many GNU and non-GNU applications. The complete list of available packages can be browsed on-line (`http://www.gnu.org/software/guix/packages`) or by running `guix package` (see Section 3.2 [Invoking guix package], page 18):

```
guix package --list-available
```

Our goal is to provide a practical 100% free software distribution of Linux-based and other variants of GNU, with a focus on the promotion and tight integration of GNU components, and an emphasis on programs and tools that help users exert that freedom.

Packages are currently available on the following platforms:

`x86_64-linux`
: Intel/AMD `x86_64` architecture, Linux-Libre kernel;

`i686-linux`
: Intel 32-bit architecture (IA32), Linux-Libre kernel;

`armhf-linux`
: ARMv7-A architecture with hard float, Thumb-2 and NEON, using the EABI hard-float application binary interface (ABI), and Linux-Libre kernel.

`aarch64-linux`
: little-endian 64-bit ARMv8-A processors, Linux-Libre kernel. This is currently in an experimental stage, with limited support. See Chapter 7 [Contributing], page 242, for how to help!

`mips64el-linux`
: little-endian 64-bit MIPS processors, specifically the Loongson series, n32 ABI, and Linux-Libre kernel.

GuixSD itself is currently only available on `i686` and `x86_64`.

For information on porting to other architectures or kernels, see Section 6.9 [Porting], page 241.

Building this distribution is a cooperative effort, and you are invited to join! See Chapter 7 [Contributing], page 242, for information about how you can help.

6.1 System Installation

This section explains how to install the Guix System Distribution (GuixSD) on a machine. The Guix package manager can also be installed on top of a running GNU/Linux system, see Chapter 2 [Installation], page 3.

[1] The term "free" here refers to the freedom provided to users of that software (`http://www.gnu.org/philosophy/free-sw.html`).

6.1.1 Limitations

As of version 0.13.0, the Guix System Distribution (GuixSD) is not production-ready. It may contain bugs and lack important features. Thus, if you are looking for a stable production system that respects your freedom as a computer user, a good solution at this point is to consider one of the more established GNU/Linux distributions (`http://www.gnu.org/distros/free-distros.html`). We hope you can soon switch to the GuixSD without fear, of course. In the meantime, you can also keep using your distribution and try out the package manager on top of it (see Chapter 2 [Installation], page 3).

Before you proceed with the installation, be aware of the following noteworthy limitations applicable to version 0.13.0:

- The installation process does not include a graphical user interface and requires familiarity with GNU/Linux (see the following subsections to get a feel of what that means.)
- Support for the Logical Volume Manager (LVM) is missing.
- More and more system services are provided (see Section 6.2.7 [Services], page 119), but some may be missing.
- More than 5,300 packages are available, but you may occasionally find that a useful package is missing.
- GNOME, Xfce, LXDE, and Enlightenment are available (see Section 6.2.7.7 [Desktop Services], page 154), as well as a number of X11 window managers. However, some graphical applications may be missing, as well as KDE.

You have been warned! But more than a disclaimer, this is an invitation to report issues (and success stories!), and to join us in improving it. See Chapter 7 [Contributing], page 242, for more info.

6.1.2 Hardware Considerations

GNU GuixSD focuses on respecting the user's computing freedom. It builds around the kernel Linux-libre, which means that only hardware for which free software drivers and firmware exist is supported. Nowadays, a wide range of off-the-shelf hardware is supported on GNU/Linux-libre—from keyboards to graphics cards to scanners and Ethernet controllers. Unfortunately, there are still areas where hardware vendors deny users control over their own computing, and such hardware is not supported on GuixSD.

One of the main areas where free drivers or firmware are lacking is WiFi devices. WiFi devices known to work include those using Atheros chips (AR9271 and AR7010), which corresponds to the `ath9k` Linux-libre driver, and those using Broadcom/AirForce chips (BCM43xx with Wireless-Core Revision 5), which corresponds to the `b43-open` Linux-libre driver. Free firmware exists for both and is available out-of-the-box on GuixSD, as part of *%base-firmware* (see Section 6.2.2 [operating-system Reference], page 109).

The Free Software Foundation (`https://www.fsf.org/`) runs *Respects Your Freedom* (`https://www.fsf.org/ryf`) (RYF), a certification program for hardware products that respect your freedom and your privacy and ensure that you have control over your device. We encourage you to check the list of RYF-certified devices.

Another useful resource is the H-Node (`https://www.h-node.org/`) web site. It contains a catalog of hardware devices with information about their support in GNU/Linux.

6.1.3 USB Stick Installation

An installation image for USB sticks can be downloaded from ‘`ftp://alpha.gnu.org/gnu/guix/guixsd-usb-i`
where *system* is one of:

`x86_64-linux`
: for a GNU/Linux system on Intel/AMD-compatible 64-bit CPUs;

`i686-linux`
: for a 32-bit GNU/Linux system on Intel-compatible CPUs.

Make sure to download the associated `.sig` file and to verify the authenticity of the image against it, along these lines:

```
$ wget ftp://alpha.gnu.org/gnu/guix/guixsd-usb-install-0.13.0.system.xz.sig
$ gpg --verify guixsd-usb-install-0.13.0.system.xz.sig
```

If that command fails because you do not have the required public key, then run this command to import it:

```
$ gpg --keyserver pgp.mit.edu --recv-keys 3CE464558A84FDC69DB40CFB090B11993D9AEBB5
```

and rerun the `gpg --verify` command.

This image contains a single partition with the tools necessary for an installation. It is meant to be copied *as is* to a large-enough USB stick.

To copy the image to a USB stick, follow these steps:

1. Decompress the image using the `xz` command:

   ```
   xz -d guixsd-usb-install-0.13.0.system.xz
   ```

2. Insert a USB stick of 1 GiB or more into your machine, and determine its device name. Assuming that the USB stick is known as `/dev/sdX`, copy the image with:

   ```
   dd if=guixsd-usb-install-0.13.0.x86_64 of=/dev/sdX
   sync
   ```

   Access to `/dev/sdX` usually requires root privileges.

Once this is done, you should be able to reboot the system and boot from the USB stick. The latter usually requires you to get in the BIOS' boot menu, where you can choose to boot from the USB stick.

See Section 6.1.6 [Installing GuixSD in a VM], page 102, if, instead, you would like to install GuixSD in a virtual machine (VM).

6.1.4 Preparing for Installation

Once you have successfully booted the image on the USB stick, you should end up with a root prompt. Several console TTYs are configured and can be used to run commands as root. TTY2 shows this documentation, browsable using the Info reader commands (see *Stand-alone GNU Info*). The installation system runs the GPM mouse daemon, which allows you to select text with the left mouse button and to paste it with the middle button.

> **Note:** Installation requires access to the Internet so that any missing dependencies of your system configuration can be downloaded. See the “Networking” section below.

The installation system includes many common tools needed for this task. But it is also a full-blown GuixSD system, which means that you can install additional packages, should you need it, using `guix package` (see Section 3.2 [Invoking guix package], page 18).

6.1.4.1 Keyboard Layout

The installation image uses the US qwerty keyboard layout. If you want to change it, you can use the `loadkeys` command. For example, the following command selects the Dvorak keyboard layout:

```
loadkeys dvorak
```

See the files under `/run/current-system/profile/share/keymaps` for a list of available keyboard layouts. Run `man loadkeys` for more information.

6.1.4.2 Networking

Run the following command see what your network interfaces are called:

```
ifconfig -a
```

. . . or, using the GNU/Linux-specific `ip` command:

```
ip a
```

Wired interfaces have a name starting with '`e`'; for example, the interface corresponding to the first on-board Ethernet controller is called '`eno1`'. Wireless interfaces have a name starting with '`w`', like '`w1p2s0`'.

Wired connection

To configure a wired network run the following command, substituting *interface* with the name of the wired interface you want to use.

```
ifconfig interface up
```

Wireless connection

To configure wireless networking, you can create a configuration file for the `wpa_supplicant` configuration tool (its location is not important) using one of the available text editors such as `zile`:

```
zile wpa_supplicant.conf
```

As an example, the following stanza can go to this file and will work for many wireless networks, provided you give the actual SSID and passphrase for the network you are connecting to:

```
network={
  ssid="my-ssid"
  key_mgmt=WPA-PSK
  psk="the network's secret passphrase"
}
```

Start the wireless service and run it in the background with the following command (substitute *interface* with the name of the network interface you want to use):

```
wpa_supplicant -c wpa_supplicant.conf -i interface -B
```

Run `man wpa_supplicant` for more information.

At this point, you need to acquire an IP address. On a network where IP addresses are automatically assigned *via* DHCP, you can run:

```
dhclient -v interface
```

Try to ping a server to see if networking is up and running:

```
ping -c 3 gnu.org
```

Setting up network access is almost always a requirement because the image does not contain all the software and tools that may be needed.

If you want to, you can continue the installation remotely by starting an SSH server:

```
herd start ssh-daemon
```

Make sure to either set a password with `passwd`, or configure OpenSSH public key authentication before logging in.

6.1.4.3 Disk Partitioning

Unless this has already been done, the next step is to partition, and then format the target partition(s).

The installation image includes several partitioning tools, including Parted (see Section "Overview" in *GNU Parted User Manual*), `fdisk`, and `cfdisk`. Run it and set up your disk with the partition layout you want:

```
cfdisk
```

If your disk uses the GUID Partition Table (GPT) format and you plan to install BIOS-based GRUB (which is the default), make sure a BIOS Boot Partition is available (see Section "BIOS installation" in *GNU GRUB manual*).

Once you are done partitioning the target hard disk drive, you have to create a file system on the relevant partition(s)[2].

Preferably, assign partitions a label so that you can easily and reliably refer to them in `file-system` declarations (see Section 6.2.3 [File Systems], page 112). This is typically done using the `-L` option of `mkfs.ext4` and related commands. So, assuming the target root partition lives at `/dev/sda1`, a file system with the label `my-root` can be created with:

```
mkfs.ext4 -L my-root /dev/sda1
```

If you are instead planning to encrypt the root partition, you can use the Cryptsetup/LUKS utilities to do that (see `man cryptsetup` for more information.) Assuming you want to store the root partition on `/dev/sda1`, the command sequence would be along these lines:

```
cryptsetup luksFormat /dev/sda1
cryptsetup open --type luks /dev/sda1 my-partition
mkfs.ext4 -L my-root /dev/mapper/my-partition
```

Once that is done, mount the target root partition under `/mnt` with a command like (again, assuming `my-root` is the label of the root partition):

```
mount LABEL=my-root /mnt
```

Finally, if you plan to use one or more swap partitions (see Section "Memory Concepts" in *The GNU C Library Reference Manual*), make sure to initialize them with `mkswap`. Assuming you have one swap partition on `/dev/sda2`, you would run:

```
mkswap /dev/sda2
```

[2] Currently GuixSD only supports ext4 and btrfs file systems. In particular, code that reads partition UUIDs and labels only works for these file system types.

```
swapon /dev/sda2
```

Alternatively, you may use a swap file. For example, assuming that in the new system you want to use the file `/swapfile` as a swap file, you would run[3]:

```
# This is 10 GiB of swap space.  Adjust "count" to change the size.
dd if=/dev/zero of=/mnt/swapfile bs=1MiB count=10240
# For security, make the file readable and writable only by root.
chmod 600 /mnt/swapfile
mkswap /mnt/swapfile
swapon /mnt/swapfile
```

Note that if you have encrypted the root partition and created a swap file in its file system as described above, then the encryption also protects the swap file, just like any other file in that file system.

6.1.5 Proceeding with the Installation

With the target partitions ready and the target root mounted on `/mnt`, we're ready to go. First, run:

```
herd start cow-store /mnt
```

This makes `/gnu/store` copy-on-write, such that packages added to it during the installation phase are written to the target disk on `/mnt` rather than kept in memory. This is necessary because the first phase of the `guix system init` command (see below) entails downloads or builds to `/gnu/store` which, initially, is an in-memory file system.

Next, you have to edit a file and provide the declaration of the operating system to be installed. To that end, the installation system comes with three text editors: GNU nano (see *GNU nano Manual*), GNU Zile (an Emacs clone), and nvi (a clone of the original BSD `vi` editor). We strongly recommend storing that file on the target root file system, say, as `/mnt/etc/config.scm`. Failing to do that, you will have lost your configuration file once you have rebooted into the newly-installed system.

See Section 6.2.1 [Using the Configuration System], page 103, for an overview of the configuration file. The example configurations discussed in that section are available under `/etc/configuration` in the installation image. Thus, to get started with a system configuration providing a graphical display server (a "desktop" system), you can run something along these lines:

```
# mkdir /mnt/etc
# cp /etc/configuration/desktop.scm /mnt/etc/config.scm
# zile /mnt/etc/config.scm
```

You should pay attention to what your configuration file contains, and in particular:

- Make sure the `grub-configuration` form refers to the device you want to install GRUB on.
- Be sure that your partition labels match the value of their respective `device` fields in your `file-system` configuration, assuming your `file-system` configuration sets the value of `title` to `'label`.

[3] This example will work for many types of file systems (e.g., ext4). However, for copy-on-write file systems (e.g., btrfs), the required steps may be different. For details, see the manual pages for `mkswap` and `swapon`.

- If there are encrypted or RAID partitions, make sure to add a `mapped-devices` field to describe them (see Section 6.2.4 [Mapped Devices], page 114).

Once you are done preparing the configuration file, the new system must be initialized (remember that the target root file system is mounted under `/mnt`):

```
guix system init /mnt/etc/config.scm /mnt
```

This copies all the necessary files and installs GRUB on `/dev/sdX`, unless you pass the `--no-bootloader` option. For more information, see Section 6.2.13 [Invoking guix system], page 214. This command may trigger downloads or builds of missing packages, which can take some time.

Once that command has completed—and hopefully succeeded!—you can run `reboot` and boot into the new system. The `root` password in the new system is initially empty; other users' passwords need to be initialized by running the `passwd` command as `root`, unless your configuration specifies otherwise (see [user-account-password], page 117).

From then on, you can update GuixSD whenever you want by running `guix pull` as `root` (see Section 3.6 [Invoking guix pull], page 29), and then running `guix system reconfigure` to build a new system generation with the latest packages and services (see Section 6.2.13 [Invoking guix system], page 214). We recommend doing that regularly so that your system includes the latest security updates (see Section 6.5 [Security Updates], page 229).

Join us on `#guix` on the Freenode IRC network or on `guix-devel@gnu.org` to share your experience—good or not so good.

6.1.6 Installing GuixSD in a Virtual Machine

If you'd like to install GuixSD in a virtual machine (VM) or on a virtual private server (VPS) rather than on your beloved machine, this section is for you.

To boot a QEMU (`http://qemu.org/`) VM for installing GuixSD in a disk image, follow these steps:

1. First, retrieve and decompress the GuixSD installation image as described previously (see Section 6.1.3 [USB Stick Installation], page 98).
2. Create a disk image that will hold the installed system. To make a qcow2-formatted disk image, use the `qemu-img` command:

   ```
   qemu-img create -f qcow2 guixsd.img 5G
   ```

   This will create a 5GB file.
3. Boot the USB installation image in an VM:

   ```
   qemu-system-x86_64 -m 1024 -smp 1 \
     -net user -net nic,model=virtio -boot menu=on \
     -drive file=guixsd.img \
     -drive file=guixsd-usb-install-0.13.0.system
   ```

   In the VM console, quickly press the *F12* key to enter the boot menu. Then press the *2* key and the *RET* key to validate your selection.
4. You're now root in the VM, proceed with the installation process. See Section 6.1.4 [Preparing for Installation], page 98, and follow the instructions.

Once installation is complete, you can boot the system that's on your `guixsd.img` image. See Section 6.2.14 [Running GuixSD in a VM], page 218, for how to do that.

6.1.7 Building the Installation Image

The installation image described above was built using the `guix system` command, specifically:

```
guix system disk-image --image-size=1G gnu/system/install.scm
```

Have a look at `gnu/system/install.scm` in the source tree, and see also Section 6.2.13 [Invoking guix system], page 214, for more information about the installation image.

6.2 System Configuration

The Guix System Distribution supports a consistent whole-system configuration mechanism. By that we mean that all aspects of the global system configuration—such as the available system services, timezone and locale settings, user accounts—are declared in a single place. Such a *system configuration* can be *instantiated*—i.e., effected.

One of the advantages of putting all the system configuration under the control of Guix is that it supports transactional system upgrades, and makes it possible to roll back to a previous system instantiation, should something go wrong with the new one (see Section 3.1 [Features], page 17). Another advantage is that it makes it easy to replicate the exact same configuration across different machines, or at different points in time, without having to resort to additional administration tools layered on top of the own tools of the system.

This section describes this mechanism. First we focus on the system administrator's viewpoint—explaining how the system is configured and instantiated. Then we show how this mechanism can be extended, for instance to support new system services.

6.2.1 Using the Configuration System

The operating system is configured by providing an `operating-system` declaration in a file that can then be passed to the `guix system` command (see Section 6.2.13 [Invoking guix system], page 214). A simple setup, with the default system services, the default Linux-Libre kernel, initial RAM disk, and boot loader looks like this:

```
;; This is an operating system configuration template
;; for a "bare bones" setup, with no X11 display server.

(use-modules (gnu))
(use-service-modules networking ssh)
(use-package-modules admin)

(operating-system
  (host-name "komputilo")
  (timezone "Europe/Berlin")
  (locale "en_US.utf8")

  ;; Assuming /dev/sdX is the target hard disk, and "my-root" is
  ;; the label of the target root file system.
  (bootloader (grub-configuration (device "/dev/sdX")))
  (file-systems (cons (file-system
                        (device "my-root")
                        (title 'label)
```

```
                        (mount-point "/")
                        (type "ext4"))
                      %base-file-systems))

  ;; This is where user accounts are specified.  The "root"
  ;; account is implicit, and is initially created with the
  ;; empty password.
  (users (cons (user-account
                (name "alice")
                (comment "Bob's sister")
                (group "users")

                ;; Adding the account to the "wheel" group
                ;; makes it a sudoer.  Adding it to "audio"
                ;; and "video" allows the user to play sound
                ;; and access the webcam.
                (supplementary-groups '("wheel"
                                        "audio" "video"))
                (home-directory "/home/alice"))
               %base-user-accounts))

  ;; Globally-installed packages.
  (packages (cons tcpdump %base-packages))

  ;; Add services to the baseline: a DHCP client and
  ;; an SSH server.
  (services (cons* (dhcp-client-service)
                   (service openssh-service-type
                            (openssh-configuration
                              (port-number 2222)))
                   %base-services)))
```

This example should be self-describing. Some of the fields defined above, such as `host-name` and `bootloader`, are mandatory. Others, such as `packages` and `services`, can be omitted, in which case they get a default value.

Below we discuss the effect of some of the most important fields (see Section 6.2.2 [operating-system Reference], page 109, for details about all the available fields), and how to *instantiate* the operating system using `guix system`.

Globally-Visible Packages

The `packages` field lists packages that will be globally visible on the system, for all user accounts—i.e., in every user's `PATH` environment variable—in addition to the per-user profiles (see Section 3.2 [Invoking guix package], page 18). The *%base-packages* variable provides all the tools one would expect for basic user and administrator tasks—including the GNU Core Utilities, the GNU Networking Utilities, the GNU Zile lightweight text editor, `find`, `grep`, etc. The example above adds tcpdump to those, taken from the `(gnu packages`

`admin`) module (see Section 6.6 [Package Modules], page 231). The `(list package output)` syntax can be used to add a specific output of a package:

```
(use-modules (gnu packages))
(use-modules (gnu packages dns))

(operating-system
  ;; ...
  (packages (cons (list bind "utils")
                  %base-packages)))
```

Referring to packages by variable name, like *tcpdump* above, has the advantage of being unambiguous; it also allows typos and such to be diagnosed right away as "unbound variables". The downside is that one needs to know which module defines which package, and to augment the `use-package-modules` line accordingly. To avoid that, one can use the `specification->package` procedure of the `(gnu packages)` module, which returns the best package for a given name or name and version:

```
(use-modules (gnu packages))

(operating-system
  ;; ...
  (packages (append (map specification->package
                         '("tcpdump" "htop" "gnupg@2.0"))
                    %base-packages)))
```

System Services

The `services` field lists *system services* to be made available when the system starts (see Section 6.2.7 [Services], page 119). The `operating-system` declaration above specifies that, in addition to the basic services, we want the `lshd` secure shell daemon listening on port 2222 (see Section 6.2.7.4 [Networking Services], page 132). Under the hood, `lsh-service` arranges so that `lshd` is started with the right command-line options, possibly with supporting configuration files generated as needed (see Section 6.2.15 [Defining Services], page 219).

Occasionally, instead of using the base services as is, you will want to customize them. To do this, use `modify-services` (see Section 6.2.15.3 [Service Reference], page 222) to modify the list.

For example, suppose you want to modify `guix-daemon` and Mingetty (the console log-in) in the *%base-services* list (see Section 6.2.7.1 [Base Services], page 120). To do that, you can write the following in your operating system declaration:

```
(define %my-services
  ;; My very own list of services.
  (modify-services %base-services
    (guix-service-type config =>
                       (guix-configuration
                        (inherit config)
                        (use-substitutes? #f)
                        (extra-options '("--gc-keep-derivations"))))
    (mingetty-service-type config =>
```

```
                          (mingetty-configuration
                           (inherit config)))))

  (operating-system
    ;; ...
    (services %my-services))
```

This changes the configuration—i.e., the service parameters—of the `guix-service-type` instance, and that of all the `mingetty-service-type` instances in the *%base-services* list. Observe how this is accomplished: first, we arrange for the original configuration to be bound to the identifier `config` in the *body*, and then we write the *body* so that it evaluates to the desired configuration. In particular, notice how we use `inherit` to create a new configuration which has the same values as the old configuration, but with a few modifications.

The configuration for a typical "desktop" usage, with an encrypted root partition, the X11 display server, GNOME and Xfce (users can choose which of these desktop environments to use at the log-in screen by pressing *F1*), network management, power management, and more, would look like this:

```
;; This is an operating system configuration template
;; for a "desktop" setup with GNOME and Xfce where the
;; root partition is encrypted with LUKS.

(use-modules (gnu) (gnu system nss))
(use-service-modules desktop)
(use-package-modules certs gnome)

(operating-system
  (host-name "antelope")
  (timezone "Europe/Paris")
  (locale "en_US.utf8")

  ;; Assuming /dev/sdX is the target hard disk, and "my-root"
  ;; is the label of the target root file system.
  (bootloader (grub-configuration (device "/dev/sdX")))

  ;; Specify a mapped device for the encrypted root partition.
  ;; The UUID is that returned by 'cryptsetup luksUUID'.
  (mapped-devices
   (list (mapped-device
          (source (uuid "12345678-1234-1234-1234-123456789abc"))
          (target "the-root-device")
          (type luks-device-mapping))))

  (file-systems (cons (file-system
                        (device "my-root")
                        (title 'label)
                        (mount-point "/")
```

```
                      (type "ext4")
                      (dependencies mapped-devices))
                    %base-file-systems))

  (users (cons (user-account
                (name "bob")
                (comment "Alice's brother")
                (group "users")
                (supplementary-groups '("wheel" "netdev"
                                        "audio" "video"))
                (home-directory "/home/bob"))
               %base-user-accounts))

  ;; This is where we specify system-wide packages.
  (packages (cons* nss-certs         ;for HTTPS access
                   gvfs              ;for user mounts
                   %base-packages))

  ;; Add GNOME and/or Xfce---we can choose at the log-in
  ;; screen with F1.  Use the "desktop" services, which
  ;; include the X11 log-in service, networking with Wicd,
  ;; and more.
  (services (cons* (gnome-desktop-service)
                   (xfce-desktop-service)
                   %desktop-services))

  ;; Allow resolution of '.local' host names with mDNS.
  (name-service-switch %mdns-host-lookup-nss))
```

A graphical environment with a choice of lightweight window managers instead of full-blown desktop environments would look like this:

```
;; This is an operating system configuration template
;; for a "desktop" setup without full-blown desktop
;; environments.

(use-modules (gnu) (gnu system nss))
(use-service-modules desktop)
(use-package-modules wm ratpoison certs suckless)

(operating-system
  (host-name "antelope")
  (timezone "Europe/Paris")
  (locale "en_US.utf8")

  ;; Assuming /dev/sdX is the target hard disk, and "my-root"
  ;; is the label of the target root file system.
  (bootloader (grub-configuration (device "/dev/sdX")))
```

```
(file-systems (cons (file-system
                      (device "my-root")
                      (title 'label)
                      (mount-point "/")
                      (type "ext4"))
                    %base-file-systems))

(users (cons (user-account
              (name "alice")
              (comment "Bob's sister")
              (group "users")
              (supplementary-groups '("wheel" "netdev"
                                      "audio" "video"))
              (home-directory "/home/alice"))
             %base-user-accounts))

;; Add a bunch of window managers; we can choose one at
;; the log-in screen with F1.
(packages (cons* ratpoison i3-wm i3status dmenu ;window managers
                 nss-certs                      ;for HTTPS access
                 %base-packages))

;; Use the "desktop" services, which include the X11
;; log-in service, networking with Wicd, and more.
(services %desktop-services)

;; Allow resolution of '.local' host names with mDNS.
(name-service-switch %mdns-host-lookup-nss))
```

See Section 6.2.7.7 [Desktop Services], page 154, for the exact list of services provided by *%desktop-services*. See Section 6.2.9 [X.509 Certificates], page 206, for background information about the `nss-certs` package that is used here.

Again, *%desktop-services* is just a list of service objects. If you want to remove services from there, you can do so using the procedures for list filtering (see Section "SRFI-1 Filtering and Partitioning" in *GNU Guile Reference Manual*). For instance, the following expression returns a list that contains all the services in *%desktop-services* minus the Avahi service:

```
(remove (lambda (service)
          (eq? (service-kind service) avahi-service-type))
        %desktop-services)
```

Instantiating the System

Assuming the `operating-system` declaration is stored in the `my-system-config.scm` file, the `guix system reconfigure my-system-config.scm` command instantiates that configuration, and makes it the default GRUB boot entry (see Section 6.2.13 [Invoking guix system], page 214).

The normal way to change the system configuration is by updating this file and rerunning `guix system reconfigure`. One should never have to touch files in `/etc` or to run commands that modify the system state such as `useradd` or `grub-install`. In fact, you must avoid that since that would not only void your warranty but also prevent you from rolling back to previous versions of your system, should you ever need to.

Speaking of roll-back, each time you run `guix system reconfigure`, a new *generation* of the system is created—without modifying or deleting previous generations. Old system generations get an entry in the GRUB boot menu, allowing you to boot them in case something went wrong with the latest generation. Reassuring, no? The `guix system list-generations` command lists the system generations available on disk. It is also possible to roll back the system via the commands `guix system roll-back` and `guix system switch-generation`.

Although the command `guix system reconfigure` will not modify previous generations, must take care when the current generation is not the latest (e.g., after invoking `guix system roll-back`), since the operation might overwrite a later generation (see Section 6.2.13 [Invoking guix system], page 214).

The Programming Interface

At the Scheme level, the bulk of an `operating-system` declaration is instantiated with the following monadic procedure (see Section 4.5 [The Store Monad], page 52):

`operating-system-derivation` *os* [Monadic Procedure]

Return a derivation that builds *os*, an `operating-system` object (see Section 4.4 [Derivations], page 50).

The output of the derivation is a single directory that refers to all the packages, configuration files, and other supporting files needed to instantiate *os*.

This procedure is provided by the `(gnu system)` module. Along with `(gnu services)` (see Section 6.2.7 [Services], page 119), this module contains the guts of GuixSD. Make sure to visit it!

6.2.2 `operating-system` Reference

This section summarizes all the options available in `operating-system` declarations (see Section 6.2.1 [Using the Configuration System], page 103).

`operating-system` [Data Type]

This is the data type representing an operating system configuration. By that, we mean all the global system configuration, not per-user configuration (see Section 6.2.1 [Using the Configuration System], page 103).

`kernel` (default: *linux-libre*)

The package object of the operating system kernel to use[4].

`kernel-arguments` (default: `'()`)

List of strings or gexps representing additional arguments to pass on the command-line of the kernel—e.g., `("console=ttyS0")`.

[4] Currently only the Linux-libre kernel is supported. In the future, it will be possible to use the GNU Hurd.

`bootloader`
: The system bootloader configuration object. See Section 6.2.12 [GRUB Configuration], page 211.

`initrd` (default: `base-initrd`)
: A two-argument monadic procedure that returns an initial RAM disk for the Linux kernel. See Section 6.2.11 [Initial RAM Disk], page 209.

`firmware` (default: *%base-firmware*)
: List of firmware packages loadable by the operating system kernel.

  The default includes firmware needed for Atheros- and Broadcom-based WiFi devices (Linux-libre modules `ath9k` and `b43-open`, respectively). See Section 6.1.2 [Hardware Considerations], page 97, for more info on supported hardware.

`host-name`
: The host name.

`hosts-file`
: A file-like object (see Section 4.6 [G-Expressions], page 56) for use as `/etc/hosts` (see Section "Host Names" in *The GNU C Library Reference Manual*). The default is a file with entries for `localhost` and *host-name*.

`mapped-devices` (default: `'()`)
: A list of mapped devices. See Section 6.2.4 [Mapped Devices], page 114.

`file-systems`
: A list of file systems. See Section 6.2.3 [File Systems], page 112.

`swap-devices` (default: `'()`)
: A list of strings identifying devices or files to be used for "swap space" (see Section "Memory Concepts" in *The GNU C Library Reference Manual*). For example, `'("/dev/sda3")` or `'("/swapfile")`. It is possible to specify a swap file in a file system on a mapped device, provided that the necessary device mapping and file system are also specified. See Section 6.2.4 [Mapped Devices], page 114, and Section 6.2.3 [File Systems], page 112.

`users` (default: `%base-user-accounts`)
`groups` (default: *%base-groups*)
: List of user accounts and groups. See Section 6.2.5 [User Accounts], page 116.

`skeletons` (default: `(default-skeletons)`)
: A list target file name/file-like object tuples (see Section 4.6 [G-Expressions], page 56). These are the skeleton files that will be added to the home directory of newly-created user accounts.

  For instance, a valid value may look like this:

```
‘((".bashrc" ,(plain-file "bashrc" "echo Hello\n"))
  (".guile" ,(plain-file "guile"
                         "(use-modules (ice-9 readline))
                          (activate-readline)")))
```

`issue` (default: *%default-issue*)
: A string denoting the contents of the `/etc/issue` file, which is displayed when users log in on a text console.

`packages` (default: *%base-packages*)
: The set of packages installed in the global profile, which is accessible at `/run/current-system/profile`.

  The default set includes core utilities and it is good practice to install non-core utilities in user profiles (see Section 3.2 [Invoking guix package], page 18).

`timezone` A timezone identifying string—e.g., `"Europe/Paris"`.

  You can run the `tzselect` command to find out which timezone string corresponds to your region. Choosing an invalid timezone name causes `guix system` to fail.

`locale` (default: `"en_US.utf8"`)
: The name of the default locale (see Section "Locale Names" in *The GNU C Library Reference Manual*). See Section 6.2.6 [Locales], page 118, for more information.

`locale-definitions` (default: *%default-locale-definitions*)
: The list of locale definitions to be compiled and that may be used at run time. See Section 6.2.6 [Locales], page 118.

`locale-libcs` (default: `(list` *`glibc`*`)`)
: The list of GNU libc packages whose locale data and tools are used to build the locale definitions. See Section 6.2.6 [Locales], page 118, for compatibility considerations that justify this option.

`name-service-switch` (default: *%default-nss*)
: Configuration of the libc name service switch (NSS)—a `<name-service-switch>` object. See Section 6.2.10 [Name Service Switch], page 207, for details.

`services` (default: *%base-services*)
: A list of service objects denoting system services. See Section 6.2.7 [Services], page 119.

`pam-services` (default: `(base-pam-services)`)
: Linux *pluggable authentication module* (PAM) services.

`setuid-programs` (default: *%setuid-programs*)
: List of string-valued G-expressions denoting setuid programs. See Section 6.2.8 [Setuid Programs], page 206.

`sudoers-file` (default: *%sudoers-specification*)
: The contents of the `/etc/sudoers` file as a file-like object (see Section 4.6 [G-Expressions], page 56).

  This file specifies which users can use the `sudo` command, what they are allowed to do, and what privileges they may gain. The default is that only `root` and members of the `wheel` group may use `sudo`.

6.2.3 File Systems

The list of file systems to be mounted is specified in the `file-systems` field of the operating system declaration (see Section 6.2.1 [Using the Configuration System], page 103). Each file system is declared using the `file-system` form, like this:

```
(file-system
  (mount-point "/home")
  (device "/dev/sda3")
  (type "ext4"))
```

As usual, some of the fields are mandatory—those shown in the example above—while others can be omitted. These are described below.

`file-system` [Data Type]

Objects of this type represent file systems to be mounted. They contain the following members:

`type` This is a string specifying the type of the file system—e.g., `"ext4"`.

`mount-point`
This designates the place where the file system is to be mounted.

`device` This names the "source" of the file system. By default it is the name of a node under `/dev`, but its meaning depends on the `title` field described below.

`title` (default: `'device`)
This is a symbol that specifies how the `device` field is to be interpreted.

When it is the symbol `device`, then the `device` field is interpreted as a file name; when it is `label`, then `device` is interpreted as a partition label name; when it is `uuid`, `device` is interpreted as a partition unique identifier (UUID).

UUIDs may be converted from their string representation (as shown by the `tune2fs -l` command) using the `uuid` form[5], like this:

```
(file-system
  (mount-point "/home")
  (type "ext4")
  (title 'uuid)
  (device (uuid "4dab5feb-d176-45de-b287-9b0a6e4c01cb")))
```

The `label` and `uuid` options offer a way to refer to disk partitions without having to hard-code their actual device name[6].

However, when the source of a file system is a mapped device (see Section 6.2.4 [Mapped Devices], page 114), its `device` field *must* refer to the mapped device name—e.g., `/dev/mapper/root-partition`—and

[5] The `uuid` form expects 16-byte UUIDs as defined in RFC 4122 (`https://tools.ietf.org/html/rfc4122`). This is the form of UUID used by the ext2 family of file systems and others, but it is different from "UUIDs" found in FAT file systems, for instance.

[6] Note that, while it is tempting to use `/dev/disk/by-uuid` and similar device names to achieve the same result, this is not recommended: These special device nodes are created by the udev daemon and may be unavailable at the time the device is mounted.

consequently `title` must be set to `'device`. This is required so that the system knows that mounting the file system depends on having the corresponding device mapping established.

`flags` (default: `'()`)
: This is a list of symbols denoting mount flags. Recognized flags include `read-only`, `bind-mount`, `no-dev` (disallow access to special files), `no-suid` (ignore setuid and setgid bits), and `no-exec` (disallow program execution.)

`options` (default: `#f`)
: This is either `#f`, or a string denoting mount options.

`mount?` (default: `#t`)
: This value indicates whether to automatically mount the file system when the system is brought up. When set to `#f`, the file system gets an entry in `/etc/fstab` (read by the `mount` command) but is not automatically mounted.

`needed-for-boot?` (default: `#f`)
: This Boolean value indicates whether the file system is needed when booting. If that is true, then the file system is mounted when the initial RAM disk (initrd) is loaded. This is always the case, for instance, for the root file system.

`check?` (default: `#t`)
: This Boolean indicates whether the file system needs to be checked for errors before being mounted.

`create-mount-point?` (default: `#f`)
: When true, the mount point is created if it does not exist yet.

`dependencies` (default: `'()`)
: This is a list of `<file-system>` or `<mapped-device>` objects representing file systems that must be mounted or mapped devices that must be opened before (and unmounted or closed after) this one.

  As an example, consider a hierarchy of mounts: `/sys/fs/cgroup` is a dependency of `/sys/fs/cgroup/cpu` and `/sys/fs/cgroup/memory`.

  Another example is a file system that depends on a mapped device, for example for an encrypted partition (see Section 6.2.4 [Mapped Devices], page 114).

The `(gnu system file-systems)` exports the following useful variables.

`%base-file-systems` [Scheme Variable]
: These are essential file systems that are required on normal systems, such as *%pseudo-terminal-file-system* and *%immutable-store* (see below.) Operating system declarations should always contain at least these.

`%pseudo-terminal-file-system` [Scheme Variable]
: This is the file system to be mounted as `/dev/pts`. It supports *pseudo-terminals* created *via* `openpty` and similar functions (see Section “Pseudo-Terminals” in *The*

GNU C Library Reference Manual). Pseudo-terminals are used by terminal emulators such as `xterm`.

`%shared-memory-file-system` [Scheme Variable]
This file system is mounted as `/dev/shm` and is used to support memory sharing across processes (see Section "Memory-mapped I/O" in *The GNU C Library Reference Manual*).

`%immutable-store` [Scheme Variable]
This file system performs a read-only "bind mount" of `/gnu/store`, making it read-only for all the users including `root`. This prevents against accidental modification by software running as `root` or by system administrators.

The daemon itself is still able to write to the store: it remounts it read-write in its own "name space."

`%binary-format-file-system` [Scheme Variable]
The `binfmt_misc` file system, which allows handling of arbitrary executable file types to be delegated to user space. This requires the `binfmt.ko` kernel module to be loaded.

`%fuse-control-file-system` [Scheme Variable]
The `fusectl` file system, which allows unprivileged users to mount and unmount user-space FUSE file systems. This requires the `fuse.ko` kernel module to be loaded.

6.2.4 Mapped Devices

The Linux kernel has a notion of *device mapping*: a block device, such as a hard disk partition, can be *mapped* into another device, usually in `/dev/mapper/`, with additional processing over the data that flows through it[7]. A typical example is encryption device mapping: all writes to the mapped device are encrypted, and all reads are deciphered, transparently. Guix extends this notion by considering any device or set of devices that are *transformed* in some way to create a new device; for instance, RAID devices are obtained by *assembling* several other devices, such as hard disks or partitions, into a new one that behaves as one partition. Other examples, not yet implemented, are LVM logical volumes.

Mapped devices are declared using the `mapped-device` form, defined as follows; for examples, see below.

`mapped-device` [Data Type]
Objects of this type represent device mappings that will be made when the system boots up.

`source` This is either a string specifying the name of the block device to be mapped, such as `"/dev/sda3"`, or a list of such strings when several devices need to be assembled for creating a new one.

[7] Note that the GNU Hurd makes no difference between the concept of a "mapped device" and that of a file system: both boil down to *translating* input/output operations made on a file to operations on its backing store. Thus, the Hurd implements mapped devices, like file systems, using the generic *translator* mechanism (see Section "Translators" in *The GNU Hurd Reference Manual*).

`target` This string specifies the name of the resulting mapped device. For kernel mappers such as encrypted devices of type `luks-device-mapping`, specifying `"my-partition"` leads to the creation of the `"/dev/mapper/my-partition"` device. For RAID devices of type `raid-device-mapping`, the full device name such as `"/dev/md0"` needs to be given.

`type` This must be a `mapped-device-kind` object, which specifies how *source* is mapped to *target*.

`luks-device-mapping` [Scheme Variable]
This defines LUKS block device encryption using the `cryptsetup` command from the package with the same name. It relies on the `dm-crypt` Linux kernel module.

`raid-device-mapping` [Scheme Variable]
This defines a RAID device, which is assembled using the `mdadm` command from the package with the same name. It requires a Linux kernel module for the appropriate RAID level to be loaded, such as `raid456` for RAID-4, RAID-5 or RAID-6, or `raid10` for RAID-10.

The following example specifies a mapping from `/dev/sda3` to `/dev/mapper/home` using LUKS—the Linux Unified Key Setup (`https://gitlab.com/cryptsetup/cryptsetup`), a standard mechanism for disk encryption. The `/dev/mapper/home` device can then be used as the `device` of a `file-system` declaration (see Section 6.2.3 [File Systems], page 112).

```
(mapped-device
  (source "/dev/sda3")
  (target "home")
  (type luks-device-mapping))
```

Alternatively, to become independent of device numbering, one may obtain the LUKS UUID (*unique identifier*) of the source device by a command like:

```
cryptsetup luksUUID /dev/sda3
```

and use it as follows:

```
(mapped-device
  (source (uuid "cb67fc72-0d54-4c88-9d4b-b225f30b0f44"))
  (target "home")
  (type luks-device-mapping))
```

It is also desirable to encrypt swap space, since swap space may contain sensitive data. One way to accomplish that is to use a swap file in a file system on a device mapped via LUKS encryption. In this way, the swap file is encrypted because the entire device is encrypted. See Section 6.1.4 [Disk Partitioning], page 98, for an example.

A RAID device formed of the partitions `/dev/sda1` and `/dev/sdb1` may be declared as follows:

```
(mapped-device
  (source (list "/dev/sda1" "/dev/sdb1"))
  (target "/dev/md0")
  (type raid-device-mapping))
```

The `/dev/md0` device can then be used as the `device` of a `file-system` declaration (see Section 6.2.3 [File Systems], page 112). Note that the RAID level need not be given; it is chosen during the initial creation and formatting of the RAID device and is determined automatically later.

6.2.5 User Accounts

User accounts and groups are entirely managed through the `operating-system` declaration. They are specified with the `user-account` and `user-group` forms:

```
(user-account
  (name "alice")
  (group "users")
  (supplementary-groups '("wheel"   ;allow use of sudo, etc.
                          "audio"   ;sound card
                          "video"   ;video devices such as webcams
                          "cdrom")) ;the good ol' CD-ROM
  (comment "Bob's sister")
  (home-directory "/home/alice"))
```

When booting or upon completion of `guix system reconfigure`, the system ensures that only the user accounts and groups specified in the `operating-system` declaration exist, and with the specified properties. Thus, account or group creations or modifications made by directly invoking commands such as `useradd` are lost upon reconfiguration or reboot. This ensures that the system remains exactly as declared.

`user-account` [Data Type]

Objects of this type represent user accounts. The following members may be specified:

`name` The name of the user account.

`group` This is the name (a string) or identifier (a number) of the user group this account belongs to.

`supplementary-groups` (default: `'()`)
Optionally, this can be defined as a list of group names that this account belongs to.

`uid` (default: `#f`)
This is the user ID for this account (a number), or `#f`. In the latter case, a number is automatically chosen by the system when the account is created.

`comment` (default: `""`)
A comment about the account, such as the account owner's full name.

`home-directory`
This is the name of the home directory for the account.

`create-home-directory?` (default: `#t`)
Indicates whether the home directory of this account should be created if it does not exist yet.

`shell` (default: Bash)
: This is a G-expression denoting the file name of a program to be used as the shell (see Section 4.6 [G-Expressions], page 56).

`system?` (default: `#f`)
: This Boolean value indicates whether the account is a "system" account. System accounts are sometimes treated specially; for instance, graphical login managers do not list them.

`password` (default: `#f`)
: You would normally leave this field to `#f`, initialize user passwords as `root` with the `passwd` command, and then let users change it with `passwd`. Passwords set with `passwd` are of course preserved across reboot and reconfiguration.

  If you *do* want to have a preset password for an account, then this field must contain the encrypted password, as a string. See Section "crypt" in *The GNU C Library Reference Manual*, for more information on password encryption, and Section "Encryption" in *GNU Guile Reference Manual*, for information on Guile's `crypt` procedure.

User group declarations are even simpler:

```
(user-group (name "students"))
```

`user-group` [Data Type]

This type is for, well, user groups. There are just a few fields:

`name` The name of the group.

`id` (default: `#f`)
: The group identifier (a number). If `#f`, a new number is automatically allocated when the group is created.

`system?` (default: `#f`)
: This Boolean value indicates whether the group is a "system" group. System groups have low numerical IDs.

`password` (default: `#f`)
: What, user groups can have a password? Well, apparently yes. Unless `#f`, this field specifies the password of the group.

For convenience, a variable lists all the basic user groups one may expect:

`%base-groups` [Scheme Variable]

This is the list of basic user groups that users and/or packages expect to be present on the system. This includes groups such as "root", "wheel", and "users", as well as groups used to control access to specific devices such as "audio", "disk", and "cdrom".

`%base-user-accounts` [Scheme Variable]

This is the list of basic system accounts that programs may expect to find on a GNU/Linux system, such as the "nobody" account.

Note that the "root" account is not included here. It is a special-case and is automatically added whether or not it is specified.

6.2.6 Locales

A *locale* defines cultural conventions for a particular language and region of the world (see Section "Locales" in *The GNU C Library Reference Manual*). Each locale has a name that typically has the form *`language_territory.codeset`*—e.g., `fr_LU.utf8` designates the locale for the French language, with cultural conventions from Luxembourg, and using the UTF-8 encoding.

Usually, you will want to specify the default locale for the machine using the `locale` field of the `operating-system` declaration (see Section 6.2.2 [operating-system Reference], page 109).

The selected locale is automatically added to the *locale definitions* known to the system if needed, with its codeset inferred from its name—e.g., `bo_CN.utf8` will be assumed to use the `UTF-8` codeset. Additional locale definitions can be specified in the `locale-definitions` slot of `operating-system`—this is useful, for instance, if the codeset could not be inferred from the locale name. The default set of locale definitions includes some widely used locales, but not all the available locales, in order to save space.

For instance, to add the North Frisian locale for Germany, the value of that field may be:

```
(cons (locale-definition
        (name "fy_DE.utf8") (source "fy_DE"))
      %default-locale-definitions)
```

Likewise, to save space, one might want `locale-definitions` to list only the locales that are actually used, as in:

```
(list (locale-definition
        (name "ja_JP.eucjp") (source "ja_JP")
        (charset "EUC-JP")))
```

The compiled locale definitions are available at `/run/current-system/locale/X.Y`, where `X.Y` is the libc version, which is the default location where the GNU libc provided by Guix looks for locale data. This can be overridden using the `LOCPATH` environment variable (see [locales-and-locpath], page 14).

The `locale-definition` form is provided by the `(gnu system locale)` module. Details are given below.

`locale-definition` [Data Type]

This is the data type of a locale definition.

`name` The name of the locale. See Section "Locale Names" in *The GNU C Library Reference Manual*, for more information on locale names.

`source` The name of the source for that locale. This is typically the *`language_territory`* part of the locale name.

`charset` (default: `"UTF-8"`)
The "character set" or "code set" for that locale, as defined by IANA (`http://www.iana.org/assignments/character-sets`).

`%default-locale-definitions` [Scheme Variable]

A list of commonly used UTF-8 locales, used as the default value of the `locale-definitions` field of `operating-system` declarations.

These locale definitions use the *normalized codeset* for the part that follows the dot in the name (see Section "Using gettextized software" in *The GNU C Library Reference Manual*). So for instance it has `uk_UA.utf8` but *not*, say, `uk_UA.UTF-8`.

6.2.6.1 Locale Data Compatibility Considerations

`operating-system` declarations provide a `locale-libcs` field to specify the GNU libc packages that are used to compile locale declarations (see Section 6.2.2 [operating-system Reference], page 109). "Why would I care?", you may ask. Well, it turns out that the binary format of locale data is occasionally incompatible from one libc version to another.

For instance, a program linked against libc version 2.21 is unable to read locale data produced with libc 2.22; worse, that program *aborts* instead of simply ignoring the incompatible locale data[8]. Similarly, a program linked against libc 2.22 can read most, but not all, of the locale data from libc 2.21 (specifically, `LC_COLLATE` data is incompatible); thus calls to `setlocale` may fail, but programs will not abort.

The "problem" in GuixSD is that users have a lot of freedom: They can choose whether and when to upgrade software in their profiles, and might be using a libc version different from the one the system administrator used to build the system-wide locale data.

Fortunately, unprivileged users can also install their own locale data and define *GUIX_LOCPATH* accordingly (see [locales-and-locpath], page 14).

Still, it is best if the system-wide locale data at `/run/current-system/locale` is built for all the libc versions actually in use on the system, so that all the programs can access it—this is especially crucial on a multi-user system. To do that, the administrator can specify several libc packages in the `locale-libcs` field of `operating-system`:

```
(use-package-modules base)

(operating-system
  ;; ...
  (locale-libcs (list glibc-2.21 (canonical-package glibc))))
```

This example would lead to a system containing locale definitions for both libc 2.21 and the current version of libc in `/run/current-system/locale`.

6.2.7 Services

An important part of preparing an `operating-system` declaration is listing *system services* and their configuration (see Section 6.2.1 [Using the Configuration System], page 103). System services are typically daemons launched when the system boots, or other actions needed at that time—e.g., configuring network access.

GuixSD has a broad definition of "service" (see Section 6.2.15.1 [Service Composition], page 219), but many services are managed by the GNU Shepherd (see Section 6.2.15.4 [Shepherd Services], page 226). On a running system, the `herd` command allows you to list the available services, show their status, start and stop them, or do other specific operations (see Section "Jump Start" in *The GNU Shepherd Manual*). For example:

```
# herd status
```

[8] Versions 2.23 and later of GNU libc will simply skip the incompatible locale data, which is already an improvement.

The above command, run as `root`, lists the currently defined services. The `herd doc` command shows a synopsis of the given service:

```
# herd doc nscd
Run libc's name service cache daemon (nscd).
```

The `start`, `stop`, and `restart` sub-commands have the effect you would expect. For instance, the commands below stop the nscd service and restart the Xorg display server:

```
# herd stop nscd
Service nscd has been stopped.
# herd restart xorg-server
Service xorg-server has been stopped.
Service xorg-server has been started.
```

The following sections document the available services, starting with the core services, that may be used in an `operating-system` declaration.

6.2.7.1 Base Services

The `(gnu services base)` module provides definitions for the basic services that one expects from the system. The services exported by this module are listed below.

`%base-services` [Scheme Variable]

This variable contains a list of basic services (see Section 6.2.15.2 [Service Types and Services], page 221, for more information on service objects) one would expect from the system: a login service (mingetty) on each tty, syslogd, the libc name service cache daemon (nscd), the udev device manager, and more.

This is the default value of the `services` field of `operating-system` declarations. Usually, when customizing a system, you will want to append services to *%base-services*, like this:

```
(cons* (avahi-service) (lsh-service) %base-services)
```

`special-files-service-type` [Scheme Variable]

This is the service that sets up "special files" such as `/bin/sh`; an instance of it is part of `%base-services`.

The value associated with `special-files-service-type` services must be a list of tuples where the first element is the "special file" and the second element is its target. By default it is:

```
`(("/bin/sh" ,(file-append bash "/bin/sh")))
```

If you want to add, say, `/usr/bin/env` to your system, you can change it to:

```
`(("/bin/sh" ,(file-append bash "/bin/sh"))
  ("/usr/bin/env" ,(file-append coreutils "/bin/env")))
```

Since this is part of `%base-services`, you can use `modify-services` to customize the set of special files (see Section 6.2.15.3 [Service Reference], page 222). But the simple way to add a special file is *via* the `extra-special-file` procedure (see below.)

`extra-special-file` *file target* [Scheme Procedure]

Use *target* as the "special file" *file*.

For example, adding the following lines to the `services` field of your operating system declaration leads to a `/usr/bin/env` symlink:

```
(extra-special-file "/usr/bin/env"
                    (file-append coreutils "/bin/env"))
```

`host-name-service` *name* [Scheme Procedure]
Return a service that sets the host name to *name*.

`login-service` *config* [Scheme Procedure]
Return a service to run login according to *config*, a `<login-configuration>` object, which specifies the message of the day, among other things.

`login-configuration` [Data Type]
This is the data type representing the configuration of login.

`motd` A file-like object containing the "message of the day".

`allow-empty-passwords?` (default: `#t`)
Allow empty passwords by default so that first-time users can log in when the 'root' account has just been created.

`mingetty-service` *config* [Scheme Procedure]
Return a service to run mingetty according to *config*, a `<mingetty-configuration>` object, which specifies the tty to run, among other things.

`mingetty-configuration` [Data Type]
This is the data type representing the configuration of Mingetty, which provides the default implementation of virtual console log-in.

`tty` The name of the console this Mingetty runs on—e.g., `"tty1"`.

`auto-login` (default: `#f`)
When true, this field must be a string denoting the user name under which the system automatically logs in. When it is `#f`, a user name and password must be entered to log in.

`login-program` (default: `#f`)
This must be either `#f`, in which case the default log-in program is used (`login` from the Shadow tool suite), or a gexp denoting the name of the log-in program.

`login-pause?` (default: `#f`)
When set to `#t` in conjunction with *auto-login*, the user will have to press a key before the log-in shell is launched.

`mingetty` (default: *mingetty*)
The Mingetty package to use.

`agetty-service` *config* [Scheme Procedure]
Return a service to run agetty according to *config*, an `<agetty-configuration>` object, which specifies the tty to run, among other things.

`agetty-configuration` [Data Type]

This is the data type representing the configuration of agetty, which implements virtual and serial console log-in. See the `agetty(8)` man page for more information.

`tty` The name of the console this agetty runs on, as a string—e.g., `"ttyS0"`. This argument is mandatory.

`baud-rate` (default: `#f`)
A string containing a comma-separated list of one or more baud rates, in descending order.

`term` (default: `#f`)
A string containing the value used for the `TERM` environment variable.

`eight-bits?` (default: `#f`)
When `#t`, the tty is assumed to be 8-bit clean, and parity detection is disabled.

`auto-login` (default: `#f`)
When passed a login name, as a string, the specified user will be logged in automatically without prompting for their login name or password.

`no-reset?` (default: `#f`)
When `#t`, don't reset terminal cflags (control modes).

`host` (default: `#f`)
This accepts a string containing the "login_host", which will be written into the `/var/run/utmpx` file.

`remote?` (default: `#f`)
When set to `#t` in conjunction with *host*, this will add an `-r` fakehost option to the command line of the login program specified in *login-program*.

`flow-control?` (default: `#f`)
When set to `#t`, enable hardware (RTS/CTS) flow control.

`no-issue?` (default: `#f`)
When set to `#t`, the contents of the `/etc/issue` file will not be displayed before presenting the login prompt.

`init-string` (default: `#f`)
This accepts a string that will be sent to the tty or modem before sending anything else. It can be used to initialize a modem.

`no-clear?` (default: `#f`)
When set to `#t`, agetty will not clear the screen before showing the login prompt.

`login-program` (default: (file-append shadow "/bin/login"))
This must be either a gexp denoting the name of a log-in program, or unset, in which case the default value is the `login` from the Shadow tool suite.

`local-line` (default: `#f`)
: Control the CLOCAL line flag. This accepts one of three symbols as arguments, `'auto`, `'always`, or `'never`. If `#f`, the default value chosen by agetty is `'auto`.

`extract-baud?` (default: `#f`)
: When set to `#t`, instruct agetty to try to extract the baud rate from the status messages produced by certain types of modems.

`skip-login?` (default: `#f`)
: When set to `#t`, do not prompt the user for a login name. This can be used with *login-program* field to use non-standard login systems.

`no-newline?` (default: `#f`)
: When set to `#t`, do not print a newline before printing the `/etc/issue` file.

`login-options` (default: `#f`)
: This option accepts a string containing options that are passed to the login program. When used with the *login-program*, be aware that a malicious user could try to enter a login name containing embedded options that could be parsed by the login program.

`login-pause` (default: `#f`)
: When set to `#t`, wait for any key before showing the login prompt. This can be used in conjunction with *auto-login* to save memory by lazily spawning shells.

`chroot` (default: `#f`)
: Change root to the specified directory. This option accepts a directory path as a string.

`hangup?` (default: `#f`)
: Use the Linux system call `vhangup` to do a virtual hangup of the specified terminal.

`keep-baud?` (default: `#f`)
: When set to `#t`, try to keep the existing baud rate. The baud rates from *baud-rate* are used when agetty receives a `BREAK` character.

`timeout` (default: `#f`)
: When set to an integer value, terminate if no user name could be read within *timeout* seconds.

`detect-case?` (default: `#f`)
: When set to `#t`, turn on support for detecting an uppercase-only terminal. This setting will detect a login name containing only uppercase letters as indicating an uppercase-only terminal and turn on some upper-to-lower case conversions. Note that this will not support Unicode characters.

`wait-cr?` (default: `#f`)
: When set to `#t`, wait for the user or modem to send a carriage-return or linefeed character before displaying `/etc/issue` or login prompt. This is typically used with the *init-string* option.

`no-hints?` (default: `#f`)
: When set to `#t`, do not print hints about Num, Caps, and Scroll locks.

`no-hostname?` (default: `#f`)
: By default, the hostname is printed. When this option is set to `#t`, no hostname will be shown at all.

`long-hostname?` (default: `#f`)
: By default, the hostname is only printed until the first dot. When this option is set to `#t`, the fully qualified hostname by `gethostname` or `getaddrinfo` is shown.

`erase-characters` (default: `#f`)
: This option accepts a string of additional characters that should be interpreted as backspace when the user types their login name.

`kill-characters` (default: `#f`)
: This option accepts a string that should be interpreted to mean "ignore all previous characters" (also called a "kill" character) when the types their login name.

`chdir` (default: `#f`)
: This option accepts, as a string, a directory path that will be changed to before login.

`delay` (default: `#f`)
: This options accepts, as an integer, the number of seconds to sleep before opening the tty and displaying the login prompt.

`nice` (default: `#f`)
: This option accepts, as an integer, the nice value with which to run the `login` program.

`extra-options` (default: `'()`)
: This option provides an "escape hatch" for the user to provide arbitrary command-line arguments to `agetty` as a list of strings.

`kmscon-service-type` *config* [Scheme Procedure]
Return a service to run kmscon (`https://www.freedesktop.org/wiki/Software/kmscon`) according to *config*, a `<kmscon-configuration>` object, which specifies the tty to run, among other things.

`kmscon-configuration` [Data Type]
This is the data type representing the configuration of Kmscon, which implements virtual console log-in.

`virtual-terminal`
: The name of the console this Kmscon runs on—e.g., `"tty1"`.

`login-program` (default: `#~(string-append #$shadow "/bin/login")`)
: A gexp denoting the name of the log-in program. The default log-in program is `login` from the Shadow tool suite.

`login-arguments` (default: `'("-p")`)
: A list of arguments to pass to `login`.

`hardware-acceleration?` (default: #f)
: Whether to use hardware acceleration.

`kmscon` (default: *kmscon*)
: The Kmscon package to use.

`nscd-service` [*config*] [*#:glibc glibc*] [*#:name-services '()*] [Scheme Procedure]
Return a service that runs the libc name service cache daemon (nscd) with the given *config*—an `<nscd-configuration>` object. See Section 6.2.10 [Name Service Switch], page 207, for an example.

`%nscd-default-configuration` [Scheme Variable]
This is the default `<nscd-configuration>` value (see below) used by `nscd-service`. It uses the caches defined by *%nscd-default-caches*; see below.

`nscd-configuration` [Data Type]
This is the data type representing the name service cache daemon (nscd) configuration.

`name-services` (default: `'()`)
: List of packages denoting *name services* that must be visible to the nscd—e.g., `(list nss-mdns)`.

`glibc` (default: *glibc*)
: Package object denoting the GNU C Library providing the `nscd` command.

`log-file` (default: `"/var/log/nscd.log"`)
: Name of the nscd log file. This is where debugging output goes when `debug-level` is strictly positive.

`debug-level` (default: `0`)
: Integer denoting the debugging levels. Higher numbers mean that more debugging output is logged.

`caches` (default: *%nscd-default-caches*)
: List of `<nscd-cache>` objects denoting things to be cached; see below.

`nscd-cache` [Data Type]
Data type representing a cache database of nscd and its parameters.

`database` This is a symbol representing the name of the database to be cached. Valid values are `passwd`, `group`, `hosts`, and `services`, which designate the corresponding NSS database (see Section "NSS Basics" in *The GNU C Library Reference Manual*).

`positive-time-to-live`
`negative-time-to-live` (default: `20`)
: A number representing the number of seconds during which a positive or negative lookup result remains in cache.

`check-files?` (default: `#t`)
: Whether to check for updates of the files corresponding to *database*.

  For instance, when *database* is `hosts`, setting this flag instructs nscd to check for updates in `/etc/hosts` and to take them into account.

`persistent?` (default: `#t`)
: Whether the cache should be stored persistently on disk.

`shared?` (default: `#t`)
: Whether the cache should be shared among users.

`max-database-size` (default: 32 MiB)
: Maximum size in bytes of the database cache.

`%nscd-default-caches` [Scheme Variable]
List of `<nscd-cache>` objects used by default by `nscd-configuration` (see above).

It enables persistent and aggressive caching of service and host name lookups. The latter provides better host name lookup performance, resilience in the face of unreliable name servers, and also better privacy—often the result of host name lookups is in local cache, so external name servers do not even need to be queried.

`syslog-configuration` [Data Type]
This data type represents the configuration of the syslog daemon.

`syslogd` (default: `#~(string-append #$inetutils "/libexec/syslogd")`)
: The syslog daemon to use.

`config-file` (default: `%default-syslog.conf`)
: The syslog configuration file to use.

`syslog-service` *config* [Scheme Procedure]
Return a service that runs a syslog daemon according to *config*.

See Section "syslogd invocation" in *GNU Inetutils*, for more information on the configuration file syntax.

`guix-configuration` [Data Type]
This data type represents the configuration of the Guix build daemon. See Section 2.5 [Invoking guix-daemon], page 11, for more information.

`guix` (default: *guix*)
: The Guix package to use.

`build-group` (default: `"guixbuild"`)
: Name of the group for build user accounts.

`build-accounts` (default: `10`)
: Number of build user accounts to create.

`authorize-key?` (default: `#t`)
: Whether to authorize the substitute keys listed in `authorized-keys`—by default that of `hydra.gnu.org` (see Section 3.3 [Substitutes], page 25).

`authorized-keys` (default: *%default-authorized-guix-keys*)
: The list of authorized key files for archive imports, as a list of string-valued gexps (see Section 3.8 [Invoking guix archive], page 32). By default, it contains that of `hydra.gnu.org` (see Section 3.3 [Substitutes], page 25).

`use-substitutes?` (default: `#t`)
: Whether to use substitutes.

`substitute-urls` (default: *%default-substitute-urls*)
: The list of URLs where to look for substitutes by default.

`extra-options` (default: `'()`)
: List of extra command-line options for `guix-daemon`.

`log-file` (default: `"/var/log/guix-daemon.log"`)
: File where `guix-daemon`'s standard output and standard error are written.

`lsof` (default: *lsof*)
: The lsof package to use.

`http-proxy` (default: `#f`)
: The HTTP proxy used for downloading fixed-output derivations and substitutes.

`tmpdir` (default: `#f`)
: A directory path where the `guix-daemon` will perform builds.

`guix-service` *config* [Scheme Procedure]
: Return a service that runs the Guix build daemon according to *config*.

`udev-service` [*#:udev udev*] [Scheme Procedure]
: Run *udev*, which populates the `/dev` directory dynamically.

`urandom-seed-service` *#f* [Scheme Procedure]
: Save some entropy in *%random-seed-file* to seed `/dev/urandom` when rebooting.

`%random-seed-file` [Scheme Variable]
: This is the name of the file where some random bytes are saved by *urandom-seed-service* to seed `/dev/urandom` when rebooting. It defaults to `/var/lib/random-seed`.

`console-keymap-service` *files ...* [Scheme Procedure]
: Return a service to load console keymaps from *files* using `loadkeys` command. Most likely, you want to load some default keymap, which can be done like this:

```
(console-keymap-service "dvorak")
```

Or, for example, for a Swedish keyboard, you may need to combine the following keymaps:

```
(console-keymap-service "se-lat6" "se-fi-lat6")
```

Also you can specify a full file name (or file names) of your keymap(s). See `man loadkeys` for details.

`gpm-service` [*#:gpm gpm*] [*#:options*] [Scheme Procedure]
Run *gpm*, the general-purpose mouse daemon, with the given command-line *options*. GPM allows users to use the mouse in the console, notably to select, copy, and paste text. The default value of *options* uses the `ps2` protocol, which works for both USB and PS/2 mice.

This service is not part of *%base-services*.

`guix-publish-service-type` [Scheme Variable]
This is the service type for `guix publish` (see Section 5.11 [Invoking guix publish], page 89). Its value must be a `guix-configuration` object, as described below.

This assumes that `/etc/guix` already contains a signing key pair as created by `guix archive --generate-key` (see Section 3.8 [Invoking guix archive], page 32). If that is not the case, the service will fail to start.

`guix-publish-configuration` [Data Type]
Data type representing the configuration of the `guix publish` service.

`guix` (default: `guix`)
The Guix package to use.

`port` (default: `80`)
The TCP port to listen for connections.

`host` (default: `"localhost"`)
The host (and thus, network interface) to listen to. Use `"0.0.0.0"` to listen on all the network interfaces.

`compression-level` (default: `3`)
The gzip compression level at which substitutes are compressed. Use `0` to disable compression altogether, and `9` to get the best compression ratio at the expense of increased CPU usage.

`nar-path` (default: `"nar"`)
The URL path at which "nars" can be fetched. See Section 5.11 [Invoking guix publish], page 89, for details.

`cache` (default: `#f`)
When it is `#f`, disable caching and instead generate archives on demand. Otherwise, this should be the name of a directory—e.g., `"/var/cache/guix/publish"`—where `guix publish` caches archives and meta-data ready to be sent. See Section 5.11 [Invoking guix publish], page 89, for more information on the tradeoffs involved.

`workers` (default: `#f`)
When it is an integer, this is the number of worker threads used for caching; when `#f`, the number of processors is used. See Section 5.11 [Invoking guix publish], page 89, for more information.

`ttl` (default: `#f`)
When it is an integer, this denotes the *time-to-live* of the published archives. See Section 5.11 [Invoking guix publish], page 89, for more information.

`rngd-service` [*#:rng-tools* `rng-tools`] [*#:device* "*/dev/hwrng*"] [Scheme Procedure]

Return a service that runs the `rngd` program from *rng-tools* to add *device* to the kernel's entropy pool. The service will fail if *device* does not exist.

`pam-limits-service` [*#:limits* `limits`] [Scheme Procedure]

Return a service that installs a configuration file for the `pam_limits` module (`http://linux-pam.org/Linux-PAM-html/sag-pam_limits.html`). The procedure optionally takes a list of `pam-limits-entry` values, which can be used to specify `ulimit` limits and nice priority limits to user sessions.

The following limits definition sets two hard and soft limits for all login sessions of users in the `realtime` group:

```
(pam-limits-service
 (list
  (pam-limits-entry "@realtime" 'both 'rtprio 99)
  (pam-limits-entry "@realtime" 'both 'memlock 'unlimited)))
```

The first entry increases the maximum realtime priority for non-privileged processes; the second entry lifts any restriction of the maximum address space that can be locked in memory. These settings are commonly used for real-time audio systems.

6.2.7.2 Scheduled Job Execution

The `(gnu services mcron)` module provides an interface to GNU mcron, a daemon to run jobs at scheduled times (see *GNU mcron*). GNU mcron is similar to the traditional Unix `cron` daemon; the main difference is that it is implemented in Guile Scheme, which provides a lot of flexibility when specifying the scheduling of jobs and their actions.

The example below defines an operating system that runs the `updatedb` (see Section "Invoking updatedb" in *Finding Files*) and the `guix gc` commands (see Section 3.5 [Invoking guix gc], page 27) daily, as well as the `mkid` command on behalf of an unprivileged user (see Section "mkid invocation" in *ID Database Utilities*). It uses gexps to introduce job definitions that are passed to mcron (see Section 4.6 [G-Expressions], page 56).

```
(use-modules (guix) (gnu) (gnu services mcron))
(use-package-modules base idutils)

(define updatedb-job
  ;; Run 'updatedb' at 3AM every day.  Here we write the
  ;; job's action as a Scheme procedure.
  #~(job '(next-hour '(3))
         (lambda ()
           (execl (string-append #$findutils "/bin/updatedb")
                  "updatedb"
                  "--prunepaths=/tmp /var/tmp /gnu/store"))))

(define garbage-collector-job
  ;; Collect garbage 5 minutes after midnight every day.
  ;; The job's action is a shell command.
  #~(job "5 0 * * *"            ;Vixie cron syntax
```

```
          "guix gc -F 1G"))

(define idutils-job
  ;; Update the index database as user "charlie" at 12:15PM
  ;; and 19:15PM.  This runs from the user's home directory.
  #~(job '(next-minute-from (next-hour '(12 19)) '(15))
         (string-append #$idutils "/bin/mkid src")
         #:user "charlie"))

(operating-system
  ;; ...
  (services (cons (mcron-service (list garbage-collector-job
                                       updatedb-job
                                       idutils-job))
                  %base-services)))
```

See Section "Guile Syntax" in *GNU mcron*, for more information on mcron job specifications. Below is the reference of the mcron service.

`mcron-service` *jobs* [*#:mcron mcron2*] [Scheme Procedure]

Return an mcron service running *mcron* that schedules *jobs*, a list of gexps denoting mcron job specifications.

This is a shorthand for:

```
(service mcron-service-type
         (mcron-configuration (mcron mcron) (jobs jobs)))
```

`mcron-service-type` [Scheme Variable]

This is the type of the `mcron` service, whose value is an `mcron-configuration` object.

This service type can be the target of a service extension that provides it additional job specifications (see Section 6.2.15.1 [Service Composition], page 219). In other words, it is possible to define services that provide additional mcron jobs to run.

`mcron-configuration` [Data Type]

Data type representing the configuration of mcron.

`mcron` (default: *mcron2*)
: The mcron package to use.

`jobs`
: This is a list of gexps (see Section 4.6 [G-Expressions], page 56), where each gexp corresponds to an mcron job specification (see Section "Syntax" in *GNU mcron*).

6.2.7.3 Log Rotation

Log files such as those found in `/var/log` tend to grow endlessly, so it's a good idea to *rotate* them once in a while—i.e., archive their contents in separate files, possibly compressed. The `(gnu services admin)` module provides an interface to GNU Rot[t]log, a log rotation tool (see *GNU Rot[t]log Manual*).

The example below defines an operating system that provides log rotation with the default settings.

```
(use-modules (guix) (gnu))
```

```
(use-service-modules admin mcron)
(use-package-modules base idutils)

(operating-system
  ;; ...
  (services (cons* (mcron-service)
                   (service rottlog-service-type)
                   %base-services)))
```

`rottlog-service-type` [Scheme Variable]

This is the type of the Rottlog service, whose value is a `rottlog-configuration` object.

This service type can define mcron jobs (see Section 6.2.7.2 [Scheduled Job Execution], page 129) to run the rottlog service.

`rottlog-configuration` [Data Type]

Data type representing the configuration of rottlog.

`rottlog` (default: `rottlog`)
: The Rottlog package to use.

`rc-file` (default: `(file-append rottlog "/etc/rc")`)
: The Rottlog configuration file to use (see Section "Mandatory RC Variables" in *GNU Rot[t]log Manual*).

`periodic-rotations` (default: `‘(("weekly" %default-rotations))`)
: A list of Rottlog period-name/period-config tuples.

  For example, taking an example from the Rottlog manual (see Section "Period Related File Examples" in *GNU Rot[t]log Manual*), a valid tuple might be:

```
("daily" ,(plain-file "daily"
                      "\
     /var/log/apache/* {
        storedir apache-archives
        rotate 6
        notifempty
        nocompress
     }"))
```

`jobs` This is a list of gexps where each gexp corresponds to an mcron job specification (see Section 6.2.7.2 [Scheduled Job Execution], page 129).

`%default-rotations` [Scheme Variable]

Specifies weekly rotation of *%rotated-files* and `"/var/log/shepherd.log"`.

`%rotated-files` [Scheme Variable]

The list of syslog-controlled files to be rotated. By default it is: `’("/var/log/messages" "/var/log/secure")`.

6.2.7.4 Networking Services

The `(gnu services networking)` module provides services to configure the network interface.

`dhcp-client-service` [*#:dhcp* `isc-dhcp`] [Scheme Procedure]
Return a service that runs *dhcp*, a Dynamic Host Configuration Protocol (DHCP) client, on all the non-loopback network interfaces.

`static-networking-service-type` [Scheme Variable]
This is the type for statically-configured network interfaces.

`static-networking-service` *interface ip* [*#:netmask #f*] [Scheme Procedure]
[*#:gateway #f*] [*#:name-servers '()*]
Return a service that starts *interface* with address *ip*. If *netmask* is true, use it as the network mask. If *gateway* is true, it must be a string specifying the default network gateway.

This procedure can be called several times, one for each network interface of interest. Behind the scenes what it does is extend `static-networking-service-type` with additional network interfaces to handle.

`wicd-service` [*#:wicd* `wicd`] [Scheme Procedure]
Return a service that runs Wicd (`https://launchpad.net/wicd`), a network management daemon that aims to simplify wired and wireless networking.

This service adds the *wicd* package to the global profile, providing several commands to interact with the daemon and configure networking: `wicd-client`, a graphical user interface, and the `wicd-cli` and `wicd-curses` user interfaces.

`network-manager-service-type` [Scheme Variable]
This is the service type for the NetworkManager (`https://wiki.gnome.org/Projects/NetworkManager`) service. The value for this service type is a `network-manager-configuration` record.

`network-manager-configuration` [Data Type]
Data type representing the configuration of NetworkManager.

`network-manager` (default: `network-manager`)
The NetworkManager package to use.

`dns` (default: `"default"`)
Processing mode for DNS, which affects how NetworkManager uses the `resolv.conf` configuration file.

‘`default`’ NetworkManager will update `resolv.conf` to reflect the nameservers provided by currently active connections.

‘`dnsmasq`’ NetworkManager will run `dnsmasq` as a local caching nameserver, using a "split DNS" configuration if you are connected to a VPN, and then update `resolv.conf` to point to the local nameserver.

‘`none`’ NetworkManager will not modify `resolv.conf`.

`connman-service-type` [Scheme Variable]

This is the service type to run Connman (`https://01.org/connman`), a network connection manager.

Its value must be an `connman-configuration` record as in this example:

```
(service connman-service-type
         (connman-configuration
           (disable-vpn? #t)))
```

See below for details about `connman-configuration`.

`connman-configuration` [Data Type]

Data Type representing the configuration of connman.

`connman` (default: *connman*)
The connman package to use.

`disable-vpn?` (default: `#f`)
When true, enable connman's vpn plugin.

`wpa-supplicant-service-type` [Scheme Variable]

This is the service type to run WPA supplicant (`https://w1.fi/wpa_supplicant/`), an authentication daemon required to authenticate against encrypted WiFi or ethernet networks. It is configured to listen for requests on D-Bus.

The value of this service is the `wpa-supplicant` package to use. Thus, it can be instantiated like this:

```
(use-modules (gnu services networking))

(service wpa-supplicant-service-type)
```

`ntp-service` [*#:ntp* `ntp`] [*#:servers* `%ntp-servers`] [*#:allow-large-adjustment? #f*] [Scheme Procedure]

Return a service that runs the daemon from *ntp*, the Network Time Protocol package (`http://www.ntp.org`). The daemon will keep the system clock synchronized with that of *servers*. *allow-large-adjustment?* determines whether `ntpd` is allowed to make an initial adjustment of more than 1,000 seconds.

`%ntp-servers` [Scheme Variable]

List of host names used as the default NTP servers.

`inetd-service-type` [Scheme variable]

This service runs the `inetd` (see Section "inetd invocation" in *GNU Inetutils*) daemon. `inetd` listens for connections on internet sockets, and lazily starts the specified server program when a connection is made on one of these sockets.

The value of this service is an `inetd-configuration` object. The following example configures the `inetd` daemon to provide the built-in `echo` service, as well as an smtp service which forwards smtp traffic over ssh to a server `smtp-server` behind a gateway `hostname`:

```
(service
 inetd-service-type
```

```
(inetd-configuration
 (entries (list
           (inetd-entry
            (name "echo")
            (socket-type 'stream)
            (protocol "tcp")
            (wait? #f)
            (user "root"))
           (inetd-entry
            (node "127.0.0.1")
            (name "smtp")
            (socket-type 'stream)
            (protocol "tcp")
            (wait? #f)
            (user "root")
            (program (file-append openssh "/bin/ssh"))
            (arguments
             '("ssh" "-qT" "-i" "/path/to/ssh_key"
               "-W" "smtp-server:25" "user@hostname")))))
```

See below for more details about `inetd-configuration`.

`inetd-configuration` [Data Type]

Data type representing the configuration of `inetd`.

`program` (default: `(file-append inetutils "/libexec/inetd")`)
The `inetd` executable to use.

`entries` (default: `'()`)
A list of `inetd` service entries. Each entry should be created by the `inetd-entry` constructor.

`inetd-entry` [Data Type]

Data type representing an entry in the `inetd` configuration. Each entry corresponds to a socket where `inetd` will listen for requests.

`node` (default: `#f`)
Optional string, a comma-separated list of local addresses `inetd` should use when listening for this service. See Section "Configuration file" in *GNU Inetutils* for a complete description of all options.

`name` A string, the name must correspond to an entry in `/etc/services`.

`socket-type`
One of `'stream`, `'dgram`, `'raw`, `'rdm` or `'seqpacket`.

`protocol` A string, must correspond to an entry in `/etc/protocols`.

`wait?` (default: `#t`)
Whether `inetd` should wait for the server to exit before listening to new service requests.

`user` A string containing the user (and, optionally, group) name of the user as whom the server should run. The group name can be specified in a suffix, separated by a colon or period, i.e. `"user"`, `"user:group"` or `"user.group"`.

`program` (default: `"internal"`)
The server program which will serve the requests, or `"internal"` if `inetd` should use a built-in service.

`arguments` (default: `'()`)
A list strings or file-like objects, which are the server program's arguments, starting with the zeroth argument, i.e. the name of the program itself. For `inetd`'s internal services, this entry must be `'()` or `'("internal")`.

See Section "Configuration file" in *GNU Inetutils* for a more detailed discussion of each configuration field.

`tor-service` [*config-file*] [*#:tor tor*] [Scheme Procedure]
Return a service to run the Tor (`https://torproject.org`) anonymous networking daemon.

The daemon runs as the `tor` unprivileged user. It is passed *config-file*, a file-like object, with an additional `User tor` line and lines for hidden services added via `tor-hidden-service`. Run `man tor` for information about the configuration file.

`tor-hidden-service` *name mapping* [Scheme Procedure]
Define a new Tor *hidden service* called *name* and implementing *mapping*. *mapping* is a list of port/host tuples, such as:

```
'((22 "127.0.0.1:22")
  (80 "127.0.0.1:8080"))
```

In this example, port 22 of the hidden service is mapped to local port 22, and port 80 is mapped to local port 8080.

This creates a `/var/lib/tor/hidden-services/`*name* directory, where the `hostname` file contains the `.onion` host name for the hidden service.

See the Tor project's documentation (`https://www.torproject.org/docs/tor-hidden-service.html.en`) for more information.

`bitlbee-service` [*#:bitlbee bitlbee*] [*#:interface* "*127.0.0.1*"] [*#:port 6667*] [*#:extra-settings* `""`] [Scheme Procedure]
Return a service that runs BitlBee (`http://bitlbee.org`), a daemon that acts as a gateway between IRC and chat networks.

The daemon will listen to the interface corresponding to the IP address specified in *interface*, on *port*. `127.0.0.1` means that only local clients can connect, whereas `0.0.0.0` means that connections can come from any networking interface.

In addition, *extra-settings* specifies a string to append to the configuration file.

Furthermore, `(gnu services ssh)` provides the following services.

`lsh-service` [*#:host-key* `"/etc/lsh/host-key"`] [*#:daemonic? #t*] [*#:interfaces '()*] [*#:port-number 22*] [*#:allow-empty-passwords? #f*] [*#:root-login? #f*] [*#:syslog-output? #t*] [*#:x11-forwarding? #t*] [*#:tcp/ip-forwarding? #t*] [*#:password-authentication? #t*] [*#:public-key-authentication? #t*] [*#:initialize? #t*] [Scheme Procedure]

Run the `lshd` program from *lsh* to listen on port *port-number*. *host-key* must designate a file containing the host key, and readable only by root.

When *daemonic?* is true, `lshd` will detach from the controlling terminal and log its output to syslogd, unless one sets *syslog-output?* to false. Obviously, it also makes lsh-service depend on existence of syslogd service. When *pid-file?* is true, `lshd` writes its PID to the file called *pid-file*.

When *initialize?* is true, automatically create the seed and host key upon service activation if they do not exist yet. This may take long and require interaction.

When *initialize?* is false, it is up to the user to initialize the randomness generator (see Section "lsh-make-seed" in *LSH Manual*), and to create a key pair with the private key stored in file *host-key* (see Section "lshd basics" in *LSH Manual*).

When *interfaces* is empty, lshd listens for connections on all the network interfaces; otherwise, *interfaces* must be a list of host names or addresses.

allow-empty-passwords? specifies whether to accept log-ins with empty passwords, and *root-login?* specifies whether to accept log-ins as root.

The other options should be self-descriptive.

`openssh-service-type` [Scheme Variable]

This is the type for the OpenSSH (`http://www.openssh.org`) secure shell daemon, `sshd`. Its value must be an `openssh-configuration` record as in this example:

```
(service openssh-service-type
         (openssh-configuration
           (x11-forwarding? #t)
           (permit-root-login 'without-password)))
```

See below for details about `openssh-configuration`.

`openssh-configuration` [Data Type]

This is the configuration record for OpenSSH's `sshd`.

`pid-file` (default: `"/var/run/sshd.pid"`)
: Name of the file where `sshd` writes its PID.

`port-number` (default: `22`)
: TCP port on which `sshd` listens for incoming connections.

`permit-root-login` (default: `#f`)
: This field determines whether and when to allow logins as root. If `#f`, root logins are disallowed; if `#t`, they are allowed. If it's the symbol `'without-password`, then root logins are permitted but not with password-based authentication.

`allow-empty-passwords?` (default: `#f`)
: When true, users with empty passwords may log in. When false, they may not.

`password-authentication?` (default: `#t`)
: When true, users may log in with their password. When false, they have other authentication methods.

`public-key-authentication?` (default: `#t`)
: When true, users may log in using public key authentication. When false, users have to use other authentication method.

  Authorized public keys are stored in `~/.ssh/authorized_keys`. This is used only by protocol version 2.

`x11-forwarding?` (default: `#f`)
: When true, forwarding of X11 graphical client connections is enabled—in other words, `ssh` options `-X` and `-Y` will work.

`challenge-response-authentication?` (default: `#f`)
: Specifies whether challenge response authentication is allowed (e.g. via PAM).

`use-pam?` (default: `#t`)
: Enables the Pluggable Authentication Module interface. If set to `#t`, this will enable PAM authentication using `challenge-response-authentication?` and `password-authentication?`, in addition to PAM account and session module processing for all authentication types.

  Because PAM challenge response authentication usually serves an equivalent role to password authentication, you should disable either `challenge-response-authentication?` or `password-authentication?`.

`print-last-log?` (default: `#t`)
: Specifies whether `sshd` should print the date and time of the last user login when a user logs in interactively.

`subsystems` (default: `'(("sftp" "internal-sftp"))`)
: Configures external subsystems (e.g. file transfer daemon).

  This is a list of two-element lists, each of which containing the subsystem name and a command (with optional arguments) to execute upon subsystem request.

  The command `internal-sftp` implements an in-process SFTP server. Alternately, one can specify the `sftp-server` command:

```
(service openssh-service-type
         (openssh-configuration
          (subsystems
           '(("sftp" ,(file-append openssh "/libexec/sftp-server"))))))
```

`dropbear-service` [*config*] [Scheme Procedure]
: Run the Dropbear SSH daemon (`https://matt.ucc.asn.au/dropbear/dropbear.html`) with the given *config*, a `<dropbear-configuration>` object.

  For example, to specify a Dropbear service listening on port 1234, add this call to the operating system's `services` field:

```
(dropbear-service (dropbear-configuration
```

```
                    (port-number 1234)))
```

`dropbear-configuration` [Data Type]

This data type represents the configuration of a Dropbear SSH daemon.

`dropbear` (default: *dropbear*)
: The Dropbear package to use.

`port-number` (default: 22)
: The TCP port where the daemon waits for incoming connections.

`syslog-output?` (default: `#t`)
: Whether to enable syslog output.

`pid-file` (default: `"/var/run/dropbear.pid"`)
: File name of the daemon's PID file.

`root-login?` (default: `#f`)
: Whether to allow `root` logins.

`allow-empty-passwords?` (default: `#f`)
: Whether to allow empty passwords.

`password-authentication?` (default: `#t`)
: Whether to enable password-based authentication.

`%facebook-host-aliases` [Scheme Variable]

This variable contains a string for use in `/etc/hosts` (see Section "Host Names" in *The GNU C Library Reference Manual*). Each line contains a entry that maps a known server name of the Facebook on-line service—e.g., `www.facebook.com`—to the local host—`127.0.0.1` or its IPv6 equivalent, `::1`.

This variable is typically used in the `hosts-file` field of an `operating-system` declaration (see Section 6.2.2 [operating-system Reference], page 109):

```
(use-modules (gnu) (guix))

(operating-system
  (host-name "mymachine")
  ;; ...
  (hosts-file
    ;; Create a /etc/hosts file with aliases for "localhost"
    ;; and "mymachine", as well as for Facebook servers.
    (plain-file "hosts"
                (string-append (local-host-aliases host-name)
                               %facebook-host-aliases))))
```

This mechanism can prevent programs running locally, such as Web browsers, from accessing Facebook.

The `(gnu services avahi)` provides the following definition.

`avahi-service` [*#:avahi* `avahi`] [*#:host-name #f*] [Scheme Procedure]
[*#:publish? #t*] [*#:ipv4? #t*] [*#:ipv6? #t*] [*#:wide-area? #f*]
[*#:domains-to-browse '()*] [*#:debug? #f*]

Return a service that runs `avahi-daemon`, a system-wide mDNS/DNS-SD responder that allows for service discovery and "zero-configuration" host name lookups (see `http://avahi.org/`), and extends the name service cache daemon (nscd) so that it can resolve `.local` host names using nss-mdns (`http://0pointer.de/lennart/projects/nss-mdns/`). Additionally, add the *avahi* package to the system profile so that commands such as `avahi-browse` are directly usable.

If *host-name* is different from `#f`, use that as the host name to publish for this machine; otherwise, use the machine's actual host name.

When *publish?* is true, publishing of host names and services is allowed; in particular, avahi-daemon will publish the machine's host name and IP address via mDNS on the local network.

When *wide-area?* is true, DNS-SD over unicast DNS is enabled.

Boolean values *ipv4?* and *ipv6?* determine whether to use IPv4/IPv6 sockets.

`openvswitch-service-type` [Scheme Variable]
This is the type of the Open vSwitch (`http://www.openvswitch.org`) service, whose value should be an `openvswitch-configuration` object.

`openvswitch-configuration` [Data Type]
Data type representing the configuration of Open vSwitch, a multilayer virtual switch which is designed to enable massive network automation through programmatic extension.

`package` (default: *openvswitch*)
Package object of the Open vSwitch.

6.2.7.5 X Window

Support for the X Window graphical display system—specifically Xorg—is provided by the `(gnu services xorg)` module. Note that there is no `xorg-service` procedure. Instead, the X server is started by the *login manager*, currently SLiM.

`sddm-configuration` [Data Type]
This is the data type representing the sddm service configuration.

`display-server` (default: `"x11"`)
Select display server to use for the greeter. Valid values are `"x11"` or `"wayland"`.

`numlock` (default: `"on"`)
Valid values are `"on"`, `"off"` or `"none"`.

`halt-command` (default `#~(string-apppend #$shepherd "/sbin/halt")`)
Command to run when halting.

`reboot-command` (default `#~(string-append #$shepherd "/sbin/reboot")`)
Command to run when rebooting.

`theme` (default `"maldives"`)
: Theme to use. Default themes provided by SDDM are `"elarun"` or `"maldives"`.

`themes-directory` (default `"/run/current-system/profile/share/sddm/themes"`)
: Directory to look for themes.

`faces-directory` (default `"/run/current-system/profile/share/sddm/faces"`)
: Directory to look for faces.

`default-path` (default `"/run/current-system/profile/bin"`)
: Default PATH to use.

`minimum-uid` (default 1000)
: Minimum UID to display in SDDM.

`maximum-uid` (default 2000)
: Maximum UID to display in SDDM

`remember-last-user?` (default #t)
: Remember last user.

`remember-last-session?` (default #t)
: Remember last session.

`hide-users` (default `""`)
: Usernames to hide from SDDM greeter.

`hide-shells` (default `#~(string-append #$shadow "/sbin/nologin")`)
: Users with shells listed will be hidden from the SDDM greeter.

`session-command` (default `#~(string-append #$sddm "/share/sddm/scripts/wayland-session")`)
: Script to run before starting a wayland session.

`sessions-directory` (default `"/run/current-system/profile/share/wayland-sessions"`)
: Directory to look for desktop files starting wayland sessions.

`xorg-server-path` (default `xorg-start-command`)
: Path to xorg-server.

`xauth-path` (default `#~(string-append #$xauth "/bin/xauth")`)
: Path to xauth.

`xephyr-path` (default `#~(string-append #$xorg-server "/bin/Xephyr")`)
: Path to Xephyr.

`xdisplay-start` (default `#~(string-append #$sddm "/share/sddm/scripts/Xsetup")`)
: Script to run after starting xorg-server.

`xdisplay-stop` (default `#~(string-append #$sddm "/share/sddm/scripts/Xstop")`)
: Script to run before stopping xorg-server.

`xsession-command` (default: `xinitr`)
: Script to run before starting a X session.

`xsessions-directory` (default: `"/run/current-system/profile/share/xsessions"`)
: Directory to look for desktop files starting X sessions.

`minimum-vt` (default: 7)
: Minimum VT to use.

`xserver-arguments` (default `"-nolisten tcp"`)
: Arguments to pass to xorg-server.

`auto-login-user` (default `""`)
: User to use for auto-login.

`auto-login-session` (default `""`)
: Desktop file to use for auto-login.

`relogin?` (default #f)
: Relogin after logout.

`sddm-service` *config* [Scheme Procedure]

Return a service that spawns the SDDM graphical login manager for config of type `<sddm-configuration>`.

```
(sddm-service (sddm-configuration
               (auto-login-user "Alice")
               (auto-login-session "xfce.desktop")))
```

`slim-service` [*#:allow-empty-passwords? #f*] [*#:auto-login? #f*] [*#:default-user* `""`] [*#:startx*] [*#:theme* `%default-slim-theme`] [*#:theme-name* `%default-slim-theme-name`] [Scheme Procedure]

Return a service that spawns the SLiM graphical login manager, which in turn starts the X display server with *startx*, a command as returned by `xorg-start-command`.

SLiM automatically looks for session types described by the `.desktop` files in `/run/current-system/profile/share/xsessions` and allows users to choose a session from the log-in screen using `F1`. Packages such as *xfce*, *sawfish*, and *ratpoison* provide `.desktop` files; adding them to the system-wide set of packages automatically makes them available at the log-in screen.

In addition, `~/.xsession` files are honored. When available, `~/.xsession` must be an executable that starts a window manager and/or other X clients.

When *allow-empty-passwords?* is true, allow logins with an empty password. When *auto-login?* is true, log in automatically as *default-user*.

If *theme* is `#f`, use the default log-in theme; otherwise *theme* must be a gexp denoting the name of a directory containing the theme to use. In that case, *theme-name* specifies the name of the theme.

`%default-theme` [Scheme Variable]
`%default-theme-name` [Scheme Variable]

The G-Expression denoting the default SLiM theme and its name.

`xorg-start-command` [*#:guile*] [*#:configuration-file #f*] [*#:xorg-server* `xorg-server`] [Scheme Procedure]

Return a derivation that builds a *guile* script to start the X server from *xorg-server*. *configuration-file* is the server configuration file or a derivation that builds it; when omitted, the result of `xorg-configuration-file` is used.

Usually the X server is started by a login manager.

`xorg-configuration-file` [*#:drivers '()*] [*#:resolutions '()*] [*#:extra-config '()*] [Scheme Procedure]

Return a configuration file for the Xorg server containing search paths for all the common drivers.

drivers must be either the empty list, in which case Xorg chooses a graphics driver automatically, or a list of driver names that will be tried in this order—e.g., `(\"modesetting\" \"vesa\")`.

Likewise, when *resolutions* is the empty list, Xorg chooses an appropriate screen resolution; otherwise, it must be a list of resolutions—e.g., `((1024 768) (640 480))`.

Last, *extra-config* is a list of strings or objects appended to the `text-file*` argument list. It is used to pass extra text to be added verbatim to the configuration file.

`screen-locker-service` *package* [*name*] [Scheme Procedure]

Add *package*, a package for a screen-locker or screen-saver whose command is *program*, to the set of setuid programs and add a PAM entry for it. For example:

```
(screen-locker-service xlockmore "xlock")
```

makes the good ol' XlockMore usable.

6.2.7.6 Printing Services

The `(gnu services cups)` module provides a Guix service definition for the CUPS printing service. To add printer support to a GuixSD system, add a `cups-service` to the operating system definition:

`cups-service-type` [Scheme Variable]

The service type for the CUPS print server. Its value should be a valid CUPS configuration (see below). To use the default settings, simply write:

```
(service cups-service-type)
```

The CUPS configuration controls the basic things about your CUPS installation: what interfaces it listens on, what to do if a print job fails, how much logging to do, and so on. To actually add a printer, you have to visit the `http://localhost:631` URL, or use a tool such as GNOME's printer configuration services. By default, configuring a CUPS service will generate a self-signed certificate if needed, for secure connections to the print server.

Suppose you want to enable the Web interface of CUPS and also add support for HP printers *via* the `hplip` package. You can do that directly, like this (you need to use the `(gnu packages cups)` module):

```
(service cups-service-type
         (cups-configuration
           (web-interface? #t)
```

```
(extensions
  (list cups-filters hplip))))
```

The available configuration parameters follow. Each parameter definition is preceded by its type; for example, '`string-list foo`' indicates that the `foo` parameter should be specified as a list of strings. There is also a way to specify the configuration as a string, if you have an old `cupsd.conf` file that you want to port over from some other system; see the end for more details.

Available `cups-configuration` fields are:

`package cups` [`cups-configuration` parameter]
The CUPS package.

`package-list extensions` [`cups-configuration` parameter]
Drivers and other extensions to the CUPS package.

`files-configuration files-configuration` [`cups-configuration` parameter]
Configuration of where to write logs, what directories to use for print spools, and related privileged configuration parameters.

Available `files-configuration` fields are:

`log-location access-log` [`files-configuration` parameter]
Defines the access log filename. Specifying a blank filename disables access log generation. The value `stderr` causes log entries to be sent to the standard error file when the scheduler is running in the foreground, or to the system log daemon when run in the background. The value `syslog` causes log entries to be sent to the system log daemon. The server name may be included in filenames using the string `%s`, as in `/var/log/cups/%s-access_log`.

Defaults to '`"/var/log/cups/access_log"`'.

`file-name cache-dir` [`files-configuration` parameter]
Where CUPS should cache data.

Defaults to '`"/var/cache/cups"`'.

`string config-file-perm` [`files-configuration` parameter]
Specifies the permissions for all configuration files that the scheduler writes.

Note that the permissions for the printers.conf file are currently masked to only allow access from the scheduler user (typically root). This is done because printer device URIs sometimes contain sensitive authentication information that should not be generally known on the system. There is no way to disable this security feature.

Defaults to '`"0640"`'.

`log-location error-log` [`files-configuration` parameter]
Defines the error log filename. Specifying a blank filename disables access log generation. The value `stderr` causes log entries to be sent to the standard error file when the scheduler is running in the foreground, or to the system log daemon when run in the background. The value `syslog` causes log entries to be

sent to the system log daemon. The server name may be included in filenames using the string `%s`, as in `/var/log/cups/%s-error_log`.

Defaults to '`"/var/log/cups/error_log"`'.

`string fatal-errors` [`files-configuration` parameter]
Specifies which errors are fatal, causing the scheduler to exit. The kind strings are:

`none` No errors are fatal.

`all` All of the errors below are fatal.

`browse` Browsing initialization errors are fatal, for example failed connections to the DNS-SD daemon.

`config` Configuration file syntax errors are fatal.

`listen` Listen or Port errors are fatal, except for IPv6 failures on the loopback or `any` addresses.

`log` Log file creation or write errors are fatal.

`permissions`
Bad startup file permissions are fatal, for example shared TLS certificate and key files with world-read permissions.

Defaults to '`"all -browse"`'.

`boolean file-device?` [`files-configuration` parameter]
Specifies whether the file pseudo-device can be used for new printer queues. The URI `file:///dev/null` is always allowed.

Defaults to '`#f`'.

`string group` [`files-configuration` parameter]
Specifies the group name or ID that will be used when executing external programs.

Defaults to '`"lp"`'.

`string log-file-perm` [`files-configuration` parameter]
Specifies the permissions for all log files that the scheduler writes.

Defaults to '`"0644"`'.

`log-location page-log` [`files-configuration` parameter]
Defines the page log filename. Specifying a blank filename disables access log generation. The value `stderr` causes log entries to be sent to the standard error file when the scheduler is running in the foreground, or to the system log daemon when run in the background. The value `syslog` causes log entries to be sent to the system log daemon. The server name may be included in filenames using the string `%s`, as in `/var/log/cups/%s-page_log`.

Defaults to '`"/var/log/cups/page_log"`'.

`string remote-root` [`files-configuration` parameter]
Specifies the username that is associated with unauthenticated accesses by clients claiming to be the root user. The default is `remroot`.

Defaults to '`"remroot"`'.

`file-name request-root` [`files-configuration` parameter]
Specifies the directory that contains print jobs and other HTTP request data.

Defaults to '`"/var/spool/cups"`'.

`sandboxing sandboxing` [`files-configuration` parameter]
Specifies the level of security sandboxing that is applied to print filters, backends, and other child processes of the scheduler; either `relaxed` or `strict`. This directive is currently only used/supported on macOS.

Defaults to '`strict`'.

`file-name server-keychain` [`files-configuration` parameter]
Specifies the location of TLS certificates and private keys. CUPS will look for public and private keys in this directory: a `.crt` files for PEM-encoded certificates and corresponding `.key` files for PEM-encoded private keys.

Defaults to '`"/etc/cups/ssl"`'.

`file-name server-root` [`files-configuration` parameter]
Specifies the directory containing the server configuration files.

Defaults to '`"/etc/cups"`'.

`boolean sync-on-close?` [`files-configuration` parameter]
Specifies whether the scheduler calls fsync(2) after writing configuration or state files.

Defaults to '`#f`'.

`space-separated-string-list system-group` [`files-configuration` parameter]
Specifies the group(s) to use for `@SYSTEM` group authentication.

`file-name temp-dir` [`files-configuration` parameter]
Specifies the directory where temporary files are stored.

Defaults to '`"/var/spool/cups/tmp"`'.

`string user` [`files-configuration` parameter]
Specifies the user name or ID that is used when running external programs.

Defaults to '`"lp"`'.

`access-log-level access-log-level` [`cups-configuration` parameter]
Specifies the logging level for the AccessLog file. The `config` level logs when printers and classes are added, deleted, or modified and when configuration files are accessed or updated. The `actions` level logs when print jobs are submitted, held, released, modified, or canceled, and any of the conditions for `config`. The `all` level logs all requests.

Defaults to '`actions`'.

`boolean auto-purge-jobs?` [`cups-configuration` parameter]
Specifies whether to purge job history data automatically when it is no longer required for quotas.

Defaults to '`#f`'.

`browse-local-protocols` [`cups-configuration` parameter]
`browse-local-protocols`
Specifies which protocols to use for local printer sharing.

Defaults to '`dnssd`'.

`boolean browse-web-if?` [`cups-configuration` parameter]
Specifies whether the CUPS web interface is advertised.

Defaults to '`#f`'.

`boolean browsing?` [`cups-configuration` parameter]
Specifies whether shared printers are advertised.

Defaults to '`#f`'.

`string classification` [`cups-configuration` parameter]
Specifies the security classification of the server. Any valid banner name can be used, including `"classified"`, `"confidential"`, `"secret"`, `"topsecret"`, and `"unclassified"`, or the banner can be omitted to disable secure printing functions.

Defaults to '`""`'.

`boolean classify-override?` [`cups-configuration` parameter]
Specifies whether users may override the classification (cover page) of individual print jobs using the `job-sheets` option.

Defaults to '`#f`'.

`default-auth-type default-auth-type` [`cups-configuration` parameter]
Specifies the default type of authentication to use.

Defaults to '`Basic`'.

`default-encryption default-encryption` [`cups-configuration` parameter]
Specifies whether encryption will be used for authenticated requests.

Defaults to '`Required`'.

`string default-language` [`cups-configuration` parameter]
Specifies the default language to use for text and web content.

Defaults to '`"en"`'.

`string default-paper-size` [`cups-configuration` parameter]
Specifies the default paper size for new print queues. '`"Auto"`' uses a locale-specific default, while '`"None"`' specifies there is no default paper size. Specific size names are typically '`"Letter"`' or '`"A4"`'.

Defaults to '`"Auto"`'.

`string default-policy` [`cups-configuration` parameter]
Specifies the default access policy to use.

Defaults to '`"default"`'.

`boolean default-shared?` [`cups-configuration` parameter]
Specifies whether local printers are shared by default.

Defaults to '`#t`'.

`non-negative-integer dirty-clean-interval` [`cups-configuration` parameter]
Specifies the delay for updating of configuration and state files, in seconds. A value of 0 causes the update to happen as soon as possible, typically within a few milliseconds.

Defaults to '`30`'.

`error-policy error-policy` [`cups-configuration` parameter]
Specifies what to do when an error occurs. Possible values are `abort-job`, which will discard the failed print job; `retry-job`, which will retry the job at a later time; `retry-this-job`, which retries the failed job immediately; and `stop-printer`, which stops the printer.

Defaults to '`stop-printer`'.

`non-negative-integer filter-limit` [`cups-configuration` parameter]
Specifies the maximum cost of filters that are run concurrently, which can be used to minimize disk, memory, and CPU resource problems. A limit of 0 disables filter limiting. An average print to a non-PostScript printer needs a filter limit of about 200. A PostScript printer needs about half that (100). Setting the limit below these thresholds will effectively limit the scheduler to printing a single job at any time.

Defaults to '`0`'.

`non-negative-integer filter-nice` [`cups-configuration` parameter]
Specifies the scheduling priority of filters that are run to print a job. The nice value ranges from 0, the highest priority, to 19, the lowest priority.

Defaults to '`0`'.

`host-name-lookups host-name-lookups` [`cups-configuration` parameter]
Specifies whether to do reverse lookups on connecting clients. The `double` setting causes `cupsd` to verify that the hostname resolved from the address matches one of the addresses returned for that hostname. Double lookups also prevent clients with unregistered addresses from connecting to your server. Only set this option to `#t` or `double` if absolutely required.

Defaults to '`#f`'.

`non-negative-integer job-kill-delay` [`cups-configuration` parameter]
Specifies the number of seconds to wait before killing the filters and backend associated with a canceled or held job.

Defaults to '`30`'.

`non-negative-integer job-retry-interval` [`cups-configuration` parameter]
Specifies the interval between retries of jobs in seconds. This is typically used for fax queues but can also be used with normal print queues whose error policy is `retry-job` or `retry-current-job`.

Defaults to '`30`'.

`non-negative-integer job-retry-limit` [`cups-configuration` parameter]
Specifies the number of retries that are done for jobs. This is typically used for fax queues but can also be used with normal print queues whose error policy is `retry-job` or `retry-current-job`.

Defaults to '`5`'.

`boolean keep-alive?` [`cups-configuration` parameter]
Specifies whether to support HTTP keep-alive connections.

Defaults to '`#t`'.

`non-negative-integer keep-alive-timeout` [`cups-configuration` parameter]
Specifies how long an idle client connection remains open, in seconds.

Defaults to '`30`'.

`non-negative-integer limit-request-body` [`cups-configuration` parameter]
Specifies the maximum size of print files, IPP requests, and HTML form data. A limit of 0 disables the limit check.

Defaults to '`0`'.

`multiline-string-list listen` [`cups-configuration` parameter]
Listens on the specified interfaces for connections. Valid values are of the form *address*:*port*, where *address* is either an IPv6 address enclosed in brackets, an IPv4 address, or `*` to indicate all addresses. Values can also be file names of local UNIX domain sockets. The Listen directive is similar to the Port directive but allows you to restrict access to specific interfaces or networks.

`non-negative-integer listen-back-log` [`cups-configuration` parameter]
Specifies the number of pending connections that will be allowed. This normally only affects very busy servers that have reached the MaxClients limit, but can also be triggered by large numbers of simultaneous connections. When the limit is reached, the operating system will refuse additional connections until the scheduler can accept the pending ones.

Defaults to '`128`'.

`location-access-control-list location-access-controls` [`cups-configuration` parameter]
Specifies a set of additional access controls.

Available `location-access-controls` fields are:

`file-name path` [`location-access-controls` parameter]
Specifies the URI path to which the access control applies.

`access-control-list` [`location-access-controls` parameter]
`access-controls`
Access controls for all access to this path, in the same format as the `access-controls` of `operation-access-control`.

Defaults to '`()`'.

`method-access-control-list` [`location-access-controls` parameter]
`method-access-controls`
Access controls for method-specific access to this path.

Defaults to '`()`'.

Available `method-access-controls` fields are:

`boolean reverse?` [`method-access-controls` parameter]
If `#t`, apply access controls to all methods except the listed methods. Otherwise apply to only the listed methods.

Defaults to '`#f`'.

`method-list methods` [`method-access-controls` parameter]
Methods to which this access control applies.

Defaults to '`()`'.

`access-control-list` [`method-access-controls` parameter]
`access-controls`
Access control directives, as a list of strings. Each string should be one directive, such as `"Order allow,deny"`.

Defaults to '`()`'.

`non-negative-integer log-debug-history` [`cups-configuration` parameter]
Specifies the number of debugging messages that are retained for logging if an error occurs in a print job. Debug messages are logged regardless of the LogLevel setting.

Defaults to '`100`'.

`log-level log-level` [`cups-configuration` parameter]
Specifies the level of logging for the ErrorLog file. The value `none` stops all logging while `debug2` logs everything.

Defaults to '`info`'.

`log-time-format log-time-format` [`cups-configuration` parameter]
Specifies the format of the date and time in the log files. The value `standard` logs whole seconds while `usecs` logs microseconds.

Defaults to '`standard`'.

`non-negative-integer max-clients` [`cups-configuration` parameter]
Specifies the maximum number of simultaneous clients that are allowed by the scheduler.

Defaults to '`100`'.

`non-negative-integer max-clients-per-host` [`cups-configuration` parameter]
Specifies the maximum number of simultaneous clients that are allowed from a single address.

Defaults to '`100`'.

`non-negative-integer max-copies` [`cups-configuration` parameter]
Specifies the maximum number of copies that a user can print of each job.

Defaults to '`9999`'.

`non-negative-integer max-hold-time` [`cups-configuration` parameter]
Specifies the maximum time a job may remain in the `indefinite` hold state before it is canceled. A value of 0 disables cancellation of held jobs.

Defaults to '`0`'.

`non-negative-integer max-jobs` [`cups-configuration` parameter]
Specifies the maximum number of simultaneous jobs that are allowed. Set to 0 to allow an unlimited number of jobs.

Defaults to '`500`'.

`non-negative-integer max-jobs-per-printer` [`cups-configuration` parameter]
Specifies the maximum number of simultaneous jobs that are allowed per printer. A value of 0 allows up to MaxJobs jobs per printer.

Defaults to '`0`'.

`non-negative-integer max-jobs-per-user` [`cups-configuration` parameter]
Specifies the maximum number of simultaneous jobs that are allowed per user. A value of 0 allows up to MaxJobs jobs per user.

Defaults to '`0`'.

`non-negative-integer max-job-time` [`cups-configuration` parameter]
Specifies the maximum time a job may take to print before it is canceled, in seconds. Set to 0 to disable cancellation of "stuck" jobs.

Defaults to '`10800`'.

`non-negative-integer max-log-size` [`cups-configuration` parameter]
Specifies the maximum size of the log files before they are rotated, in bytes. The value 0 disables log rotation.

Defaults to '`1048576`'.

`non-negative-integer multiple-operation-timeout` [`cups-configuration` parameter]
Specifies the maximum amount of time to allow between files in a multiple file print job, in seconds.

Defaults to '`300`'.

`string page-log-format` [`cups-configuration` parameter]

Specifies the format of PageLog lines. Sequences beginning with percent (‘`%`’) characters are replaced with the corresponding information, while all other characters are copied literally. The following percent sequences are recognized:

‘`%%`’ insert a single percent character

‘`%{name}`’ insert the value of the specified IPP attribute

‘`%C`’ insert the number of copies for the current page

‘`%P`’ insert the current page number

‘`%T`’ insert the current date and time in common log format

‘`%j`’ insert the job ID

‘`%p`’ insert the printer name

‘`%u`’ insert the username

A value of the empty string disables page logging. The string `%p %u %j %T %P %C %{job-billing} %{job-originating-host-name} %{job-name} %{media} %{sides}` creates a page log with the standard items.

Defaults to ‘`""`’.

`environment-variables` [`cups-configuration` parameter]

`environment-variables`

Passes the specified environment variable(s) to child processes; a list of strings.

Defaults to ‘`()`’.

`policy-configuration-list policies` [`cups-configuration` parameter]

Specifies named access control policies.

Available `policy-configuration` fields are:

`string name` [`policy-configuration` parameter]

Name of the policy.

`string job-private-access` [`policy-configuration` parameter]

Specifies an access list for a job's private values. `@ACL` maps to the printer's requesting-user-name-allowed or requesting-user-name-denied values. `@OWNER` maps to the job's owner. `@SYSTEM` maps to the groups listed for the `system-group` field of the `files-config` configuration, which is reified into the `cups-files.conf(5)` file. Other possible elements of the access list include specific user names, and `@`*`group`* to indicate members of a specific group. The access list may also be simply `all` or `default`.

Defaults to ‘`"@OWNER @SYSTEM"`’.

`string job-private-values` [`policy-configuration` parameter]

Specifies the list of job values to make private, or `all`, `default`, or `none`.

Defaults to ‘`"job-name job-originating-host-name job-originating-user-name phone"`’.

`string` [`policy-configuration` parameter]
`subscription-private-access`

Specifies an access list for a subscription's private values. `@ACL` maps to the printer's requesting-user-name-allowed or requesting-user-name-denied values. `@OWNER` maps to the job's owner. `@SYSTEM` maps to the groups listed for the `system-group` field of the `files-config` configuration, which is reified into the `cups-files.conf(5)` file. Other possible elements of the access list include specific user names, and `@group` to indicate members of a specific group. The access list may also be simply `all` or `default`.

Defaults to '`"@OWNER @SYSTEM"`'.

`string` [`policy-configuration` parameter]
`subscription-private-values`

Specifies the list of job values to make private, or `all`, `default`, or `none`.

Defaults to '`"notify-events notify-pull-method notify-recipient-uri notify-subscriber-user-name notify-user-data"`'.

`operation-access-control-list` [`policy-configuration` parameter]
`access-controls`

Access control by IPP operation.

Defaults to '`()`'.

`boolean-or-non-negative-integer` [`cups-configuration` parameter]
`preserve-job-files`

Specifies whether job files (documents) are preserved after a job is printed. If a numeric value is specified, job files are preserved for the indicated number of seconds after printing. Otherwise a boolean value applies indefinitely.

Defaults to '`86400`'.

`boolean-or-non-negative-integer` [`cups-configuration` parameter]
`preserve-job-history`

Specifies whether the job history is preserved after a job is printed. If a numeric value is specified, the job history is preserved for the indicated number of seconds after printing. If `#t`, the job history is preserved until the MaxJobs limit is reached.

Defaults to '`#t`'.

`non-negative-integer reload-timeout` [`cups-configuration` parameter]

Specifies the amount of time to wait for job completion before restarting the scheduler.

Defaults to '`30`'.

`string rip-cache` [`cups-configuration` parameter]

Specifies the maximum amount of memory to use when converting documents into bitmaps for a printer.

Defaults to '`"128m"`'.

`string server-admin` [`cups-configuration` parameter]

Specifies the email address of the server administrator.

Defaults to '`"root@localhost.localdomain"`'.

`host-name-list-or-* server-alias` [`cups-configuration` parameter]
The ServerAlias directive is used for HTTP Host header validation when clients connect to the scheduler from external interfaces. Using the special name `*` can expose your system to known browser-based DNS rebinding attacks, even when accessing sites through a firewall. If the auto-discovery of alternate names does not work, we recommend listing each alternate name with a ServerAlias directive instead of using `*`.

Defaults to '`*`'.

`string server-name` [`cups-configuration` parameter]
Specifies the fully-qualified host name of the server.

Defaults to '`"localhost"`'.

`server-tokens server-tokens` [`cups-configuration` parameter]
Specifies what information is included in the Server header of HTTP responses. `None` disables the Server header. `ProductOnly` reports `CUPS`. `Major` reports `CUPS 2`. `Minor` reports `CUPS 2.0`. `Minimal` reports `CUPS 2.0.0`. `OS` reports `CUPS 2.0.0 (`*uname*`)` where *uname* is the output of the `uname` command. `Full` reports `CUPS 2.0.0 (`*uname*`) IPP/2.0`.

Defaults to '`Minimal`'.

`string set-env` [`cups-configuration` parameter]
Set the specified environment variable to be passed to child processes.

Defaults to '`"variable value"`'.

`multiline-string-list ssl-listen` [`cups-configuration` parameter]
Listens on the specified interfaces for encrypted connections. Valid values are of the form *address*:*port*, where *address* is either an IPv6 address enclosed in brackets, an IPv4 address, or `*` to indicate all addresses.

Defaults to '`()`'.

`ssl-options ssl-options` [`cups-configuration` parameter]
Sets encryption options. By default, CUPS only supports encryption using TLS v1.0 or higher using known secure cipher suites. The `AllowRC4` option enables the 128-bit RC4 cipher suites, which are required for some older clients that do not implement newer ones. The `AllowSSL3` option enables SSL v3.0, which is required for some older clients that do not support TLS v1.0.

Defaults to '`()`'.

`boolean strict-conformance?` [`cups-configuration` parameter]
Specifies whether the scheduler requires clients to strictly adhere to the IPP specifications.

Defaults to '`#f`'.

`non-negative-integer timeout` [`cups-configuration` parameter]
Specifies the HTTP request timeout, in seconds.

Defaults to '`300`'.

`boolean web-interface?` [`cups-configuration` parameter]
Specifies whether the web interface is enabled.

Defaults to '`#f`'.

At this point you're probably thinking "oh dear, Guix manual, I like you but you can stop already with the configuration options". Indeed. However, one more point: it could be that you have an existing `cupsd.conf` that you want to use. In that case, you can pass an `opaque-cups-configuration` as the configuration of a `cups-service-type`.

Available `opaque-cups-configuration` fields are:

`package cups` [`opaque-cups-configuration` parameter]
The CUPS package.

`string cupsd.conf` [`opaque-cups-configuration` parameter]
The contents of the `cupsd.conf`, as a string.

`string cups-files.conf` [`opaque-cups-configuration` parameter]
The contents of the `cups-files.conf` file, as a string.

For example, if your `cupsd.conf` and `cups-files.conf` are in strings of the same name, you could instantiate a CUPS service like this:

```
(service cups-service-type
         (opaque-cups-configuration
           (cupsd.conf cupsd.conf)
           (cups-files.conf cups-files.conf)))
```

6.2.7.7 Desktop Services

The `(gnu services desktop)` module provides services that are usually useful in the context of a "desktop" setup—that is, on a machine running a graphical display server, possibly with graphical user interfaces, etc. It also defines services that provide specific desktop environments like GNOME and XFCE.

To simplify things, the module defines a variable containing the set of services that users typically expect on a machine with a graphical environment and networking:

`%desktop-services` [Scheme Variable]
This is a list of services that builds upon *%base-services* and adds or adjusts services for a typical "desktop" setup.

In particular, it adds a graphical login manager (see Section 6.2.7.5 [X Window], page 139), screen lockers, a network management tool (see Section 6.2.7.4 [Networking Services], page 132), energy and color management services, the `elogind` login and seat manager, the Polkit privilege service, the GeoClue location service, an NTP client (see Section 6.2.7.4 [Networking Services], page 132), the Avahi daemon, and has the name service switch service configured to be able to use `nss-mdns` (see Section 6.2.10 [Name Service Switch], page 207).

The *%desktop-services* variable can be used as the `services` field of an `operating-system` declaration (see Section 6.2.2 [operating-system Reference], page 109).

Additionally, the `gnome-desktop-service` and `xfce-desktop-service` procedures can add GNOME and/or XFCE to a system. To "add GNOME" means that system-level services like the backlight adjustment helpers and the power management utilities are added to the system, extending `polkit` and `dbus` appropriately, allowing GNOME to operate with elevated privileges on a limited number of special-purpose system interfaces. Additionally, adding a service made by `gnome-desktop-service` adds the GNOME metapackage to the system profile. Likewise, adding the XFCE service not only adds the `xfce` metapackage to the system profile, but it also gives the Thunar file manager the ability to open a "root-mode" file management window, if the user authenticates using the administrator's password via the standard polkit graphical interface.

`gnome-desktop-service` [Scheme Procedure]

Return a service that adds the `gnome` package to the system profile, and extends polkit with the actions from `gnome-settings-daemon`.

`xfce-desktop-service` [Scheme Procedure]

Return a service that adds the `xfce` package to the system profile, and extends polkit with the ability for `thunar` to manipulate the file system as root from within a user session, after the user has authenticated with the administrator's password.

Because the GNOME and XFCE desktop services pull in so many packages, the default `%desktop-services` variable doesn't include either of them by default. To add GNOME or XFCE, just `cons` them onto `%desktop-services` in the `services` field of your `operating-system`:

```
(use-modules (gnu))
(use-service-modules desktop)
(operating-system
  ...
  ;; cons* adds items to the list given as its last argument.
  (services (cons* (gnome-desktop-service)
                   (xfce-desktop-service)
                   %desktop-services))
  ...)
```

These desktop environments will then be available as options in the graphical login window.

The actual service definitions included in `%desktop-services` and provided by `(gnu services dbus)` and `(gnu services desktop)` are described below.

`dbus-service` [*#:dbus dbus*] [*#:services '()*] [Scheme Procedure]

Return a service that runs the "system bus", using *dbus*, with support for *services*.

D-Bus (`http://dbus.freedesktop.org/`) is an inter-process communication facility. Its system bus is used to allow system services to communicate and to be notified of system-wide events.

services must be a list of packages that provide an `etc/dbus-1/system.d` directory containing additional D-Bus configuration and policy files. For example, to allow avahi-daemon to use the system bus, *services* must be equal to `(list avahi)`.

`elogind-service` [*#:config config*] [Scheme Procedure]

Return a service that runs the `elogind` login and seat management daemon. Elogind (`https://github.com/andywingo/elogind`) exposes a D-Bus interface that can be used to know which users are logged in, know what kind of sessions they have open, suspend the system, inhibit system suspend, reboot the system, and other tasks.

Elogind handles most system-level power events for a computer, for example suspending the system when a lid is closed, or shutting it down when the power button is pressed.

The *config* keyword argument specifies the configuration for elogind, and should be the result of an `(elogind-configuration (`*parameter value*`)...)` invocation. Available parameters and their default values are:

```
kill-user-processes?
          #f

kill-only-users
          ()

kill-exclude-users
          ("root")

inhibit-delay-max-seconds
          5

handle-power-key
          poweroff

handle-suspend-key
          suspend

handle-hibernate-key
          hibernate

handle-lid-switch
          suspend

handle-lid-switch-docked
          ignore

power-key-ignore-inhibited?
          #f

suspend-key-ignore-inhibited?
          #f

hibernate-key-ignore-inhibited?
          #f

lid-switch-ignore-inhibited?
          #t

holdoff-timeout-seconds
          30
```

```
idle-action
          ignore

idle-action-seconds
          (* 30 60)

runtime-directory-size-percent
          10

runtime-directory-size
          #f

remove-ipc?
          #t

suspend-state
          ("mem" "standby" "freeze")

suspend-mode
          ()

hibernate-state
          ("disk")

hibernate-mode
          ("platform" "shutdown")

hybrid-sleep-state
          ("disk")

hybrid-sleep-mode
          ("suspend" "platform" "shutdown")
```

`polkit-service` [*#:polkit `polkit`*] [Scheme Procedure]
Return a service that runs the Polkit privilege management service (`http://www.freedesktop.org/wiki/Software/polkit/`), which allows system administrators to grant access to privileged operations in a structured way. By querying the Polkit service, a privileged system component can know when it should grant additional capabilities to ordinary users. For example, an ordinary user can be granted the capability to suspend the system if the user is logged in locally.

`upower-service` [*#:upower `upower`*] [*#:watts-up-pro? #f*] [*#:poll-batteries? #t*] [*#:ignore-lid? #f*] [*#:use-percentage-for-policy? #f*] [*#:percentage-low 10*] [*#:percentage-critical 3*] [*#:percentage-action 2*] [*#:time-low 1200*] [*#:time-critical 300*] [*#:time-action 120*] [*#:critical-power-action 'hybrid-sleep*] [Scheme Procedure]
Return a service that runs `upowerd` (`http://upower.freedesktop.org/`), a system-wide monitor for power consumption and battery levels, with the given configuration settings. It implements the `org.freedesktop.UPower` D-Bus interface, and is notably used by GNOME.

`udisks-service` [*#:udisks `udisks`*] [Scheme Procedure]
Return a service for UDisks (`http://udisks.freedesktop.org/docs/latest/`), a *disk management* daemon that provides user interfaces with notifications and

ways to mount/unmount disks. Programs that talk to UDisks include the `udisksctl` command, part of UDisks, and GNOME Disks.

`colord-service` [*#:colord* `colord`] [Scheme Procedure]
Return a service that runs `colord`, a system service with a D-Bus interface to manage the color profiles of input and output devices such as screens and scanners. It is notably used by the GNOME Color Manager graphical tool. See the colord web site (`http://www.freedesktop.org/software/colord/`) for more information.

`geoclue-application` *name* [*#:allowed? #t*] [*#:system? #f*] [*#:users '()*] [Scheme Procedure]
Return a configuration allowing an application to access GeoClue location data. *name* is the Desktop ID of the application, without the `.desktop` part. If *allowed?* is true, the application will have access to location information by default. The boolean *system?* value indicates whether an application is a system component or not. Finally *users* is a list of UIDs of all users for which this application is allowed location info access. An empty users list means that all users are allowed.

`%standard-geoclue-applications` [Scheme Variable]
The standard list of well-known GeoClue application configurations, granting authority to the GNOME date-and-time utility to ask for the current location in order to set the time zone, and allowing the IceCat and Epiphany web browsers to request location information. IceCat and Epiphany both query the user before allowing a web page to know the user's location.

`geoclue-service` [*#:colord* `colord`] [*#:whitelist '()*] [*#:wifi-geolocation-url "https://location.services.mozilla.com/v1/geolocate?key=geoclue"*] [*#:submit-data? #f*] [Scheme Procedure]
[#:wifi-submission-url "https://location.services.mozilla.com/v1/submit?key=geoclue"] [#:submission-nick "geoclue"] [#:applications %standard-geoclue-applications]
Return a service that runs the GeoClue location service. This service provides a D-Bus interface to allow applications to request access to a user's physical location, and optionally to add information to online location databases. See the GeoClue web site (`https://wiki.freedesktop.org/www/Software/GeoClue/`) for more information.

`bluetooth-service` [*#:bluez* `bluez`] [Scheme Procedure]
Return a service that runs the `bluetoothd` daemon, which manages all the Bluetooth devices and provides a number of D-Bus interfaces.

Users need to be in the `lp` group to access the D-Bus service.

6.2.7.8 Database Services

The `(gnu services databases)` module provides the following services.

`postgresql-service` [*#:postgresql postgresql*] [*#:config-file*] [*#:data-directory "/var/lib/postgresql/data"*] [*#:port 5432*] [*#:locale "en_US.utf8"*] [Scheme Procedure]
Return a service that runs *postgresql*, the PostgreSQL database server.

The PostgreSQL daemon loads its runtime configuration from *config-file*, creates a database cluster with *locale* as the default locale, stored in *data-directory*. It then listens on *port*.

`mysql-service` [*#:config* (*mysql-configuration*)] [Scheme Procedure]
Return a service that runs `mysqld`, the MySQL or MariaDB database server.

The optional *config* argument specifies the configuration for `mysqld`, which should be a `<mysql-configuration>` object.

`mysql-configuration` [Data Type]
Data type representing the configuration of *mysql-service*.

`mysql` (default: *mariadb*)
Package object of the MySQL database server, can be either *mariadb* or *mysql*.

For MySQL, a temporary root password will be displayed at activation time. For MariaDB, the root password is empty.

`port` (default: `3306`)
TCP port on which the database server listens for incoming connections.

`redis-service-type` [Scheme Variable]
This is the service type for the Redis (`https://redis.io/`) key/value store, whose value is a `redis-configuration` object.

`redis-configuration` [Data Type]
Data type representing the configuration of redis.

`redis` (default: `redis`)
The Redis package to use.

`bind` (default: `"127.0.0.1"`)
Network interface on which to listen.

`port` (default: `6379`)
Port on which to accept connections on, a value of 0 will disable listening on a TCP socket.

`working-directory` (default: `"/var/lib/redis"`)
Directory in which to store the database and related files.

6.2.7.9 Mail Services

The `(gnu services mail)` module provides Guix service definitions for email services: IMAP, POP3, and LMTP servers, as well as mail transport agents (MTAs). Lots of acronyms! These services are detailed in the subsections below.

Dovecot Service

`dovecot-service` [*#:config* (*dovecot-configuration*)] [Scheme Procedure]
Return a service that runs the Dovecot IMAP/POP3/LMTP mail server.

By default, Dovecot does not need much configuration; the default configuration object created by `(dovecot-configuration)` will suffice if your mail is delivered to `~/Maildir`. A self-signed certificate will be generated for TLS-protected connections, though Dovecot will also listen on cleartext ports by default. There are a number of options, though, which mail administrators might need to change, and as is the case with other services, Guix allows the system administrator to specify these parameters via a uniform Scheme interface.

For example, to specify that mail is located at `maildir~/.mail`, one would instantiate the Dovecot service like this:

```
(dovecot-service #:config
                 (dovecot-configuration
                  (mail-location "maildir:~/.mail")))
```

The available configuration parameters follow. Each parameter definition is preceded by its type; for example, '`string-list foo`' indicates that the `foo` parameter should be specified as a list of strings. There is also a way to specify the configuration as a string, if you have an old `dovecot.conf` file that you want to port over from some other system; see the end for more details.

Available `dovecot-configuration` fields are:

`package dovecot` [`dovecot-configuration` parameter]
The dovecot package.

`comma-separated-string-list listen` [`dovecot-configuration` parameter]
A list of IPs or hosts where to listen for connections. '`*`' listens on all IPv4 interfaces, '`::`' listens on all IPv6 interfaces. If you want to specify non-default ports or anything more complex, customize the address and port fields of the '`inet-listener`' of the specific services you are interested in.

`protocol-configuration-list protocols` [`dovecot-configuration` parameter]
List of protocols we want to serve. Available protocols include '`imap`', '`pop3`', and '`lmtp`'.

Available `protocol-configuration` fields are:

`string name` [`protocol-configuration` parameter]
The name of the protocol.

`string auth-socket-path` [`protocol-configuration` parameter]
UNIX socket path to the master authentication server to find users. This is used by imap (for shared users) and lda. It defaults to '`"/var/run/dovecot/auth-userdb"`'.

`space-separated-string-list mail-plugins` [`protocol-configuration` parameter]
Space separated list of plugins to load.

`non-negative-integer mail-max-userip-connections` [`protocol-configuration` parameter]
Maximum number of IMAP connections allowed for a user from each IP address. NOTE: The username is compared case-sensitively. Defaults to '`10`'.

`service-configuration-list services` [`dovecot-configuration` parameter]
List of services to enable. Available services include '`imap`', '`imap-login`', '`pop3`', '`pop3-login`', '`auth`', and '`lmtp`'.

Available `service-configuration` fields are:

`string kind` [`service-configuration` parameter]
The service kind. Valid values include `director`, `imap-login`, `pop3-login`, `lmtp`, `imap`, `pop3`, `auth`, `auth-worker`, `dict`, `tcpwrap`, `quota-warning`, or anything else.

`listener-configuration-list listeners` [`service-configuration` parameter]
Listeners for the service. A listener is either a `unix-listener-configuration`, a `fifo-listener-configuration`, or an `inet-listener-configuration`. Defaults to '`()`'.

Available `unix-listener-configuration` fields are:

`string path` [`unix-listener-configuration` parameter]
Path to the file, relative to `base-dir` field. This is also used as the section name.

`string mode` [`unix-listener-configuration` parameter]
The access mode for the socket. Defaults to '`"0600"`'.

`string user` [`unix-listener-configuration` parameter]
The user to own the socket. Defaults to '`""`'.

`string group` [`unix-listener-configuration` parameter]
The group to own the socket. Defaults to '`""`'.

Available `fifo-listener-configuration` fields are:

`string path` [`fifo-listener-configuration` parameter]
Path to the file, relative to `base-dir` field. This is also used as the section name.

`string mode` [`fifo-listener-configuration` parameter]
The access mode for the socket. Defaults to '`"0600"`'.

`string user` [`fifo-listener-configuration` parameter]
The user to own the socket. Defaults to '`""`'.

`string group` [`fifo-listener-configuration` parameter]
The group to own the socket. Defaults to '`""`'.

Available `inet-listener-configuration` fields are:

`string protocol` [`inet-listener-configuration` parameter]
The protocol to listen for.

`string address` [`inet-listener-configuration` parameter]
The address on which to listen, or empty for all addresses. Defaults to '`""`'.

`non-negative-integer port` [`inet-listener-configuration` parameter]
The port on which to listen.

`boolean ssl?` [`inet-listener-configuration` parameter]
Whether to use SSL for this service; '`yes`', '`no`', or '`required`'. Defaults to '`#t`'.

`non-negative-integer service-count` [`service-configuration` parameter]
Number of connections to handle before starting a new process. Typically the only useful values are 0 (unlimited) or 1. 1 is more secure, but 0 is faster. <doc/wiki/LoginProcess.txt>. Defaults to '`1`'.

`non-negative-integer process-min-avail` [`service-configuration` parameter]
Number of processes to always keep waiting for more connections. Defaults to '`0`'.

`non-negative-integer vsz-limit` [`service-configuration` parameter]
If you set '`service-count 0`', you probably need to grow this. Defaults to '`256000000`'.

`dict-configuration dict` [`dovecot-configuration` parameter]
Dict configuration, as created by the `dict-configuration` constructor.

Available `dict-configuration` fields are:

`free-form-fields entries` [`dict-configuration` parameter]
A list of key-value pairs that this dict should hold. Defaults to '`()`'.

`passdb-configuration-list passdbs` [`dovecot-configuration` parameter]
A list of passdb configurations, each one created by the `passdb-configuration` constructor.

Available `passdb-configuration` fields are:

`string driver` [`passdb-configuration` parameter]
The driver that the passdb should use. Valid values include '`pam`', '`passwd`', '`shadow`', '`bsdauth`', and '`static`'. Defaults to '`"pam"`'.

`space-separated-string-list args` [`passdb-configuration` parameter]
Space separated list of arguments to the passdb driver. Defaults to '`""`'.

`userdb-configuration-list userdbs` [`dovecot-configuration` parameter]
List of userdb configurations, each one created by the `userdb-configuration` constructor.

Available `userdb-configuration` fields are:

`string driver` [`userdb-configuration` parameter]
The driver that the userdb should use. Valid values include '`passwd`' and '`static`'. Defaults to '`"passwd"`'.

`space-separated-string-list args` [`userdb-configuration` parameter]
Space separated list of arguments to the userdb driver. Defaults to '`""`'.

`free-form-args override-fields` [`userdb-configuration` parameter]
Override fields from passwd. Defaults to '`()`'.

`plugin-configuration` [`dovecot-configuration` parameter]
`plugin-configuration`
Plug-in configuration, created by the `plugin-configuration` constructor.

`list-of-namespace-configuration` [`dovecot-configuration` parameter]
`namespaces`
List of namespaces. Each item in the list is created by the `namespace-configuration` constructor.

Available `namespace-configuration` fields are:

`string name` [`namespace-configuration` parameter]
Name for this namespace.

`string type` [`namespace-configuration` parameter]
Namespace type: '`private`', '`shared`' or '`public`'. Defaults to '`"private"`'.

`string separator` [`namespace-configuration` parameter]
Hierarchy separator to use. You should use the same separator for all namespaces or some clients get confused. '`/`' is usually a good one. The default however depends on the underlying mail storage format. Defaults to '`""`'.

`string prefix` [`namespace-configuration` parameter]
Prefix required to access this namespace. This needs to be different for all namespaces. For example '`Public/`'. Defaults to '`""`'.

`string location` [`namespace-configuration` parameter]
Physical location of the mailbox. This is in the same format as mail_location, which is also the default for it. Defaults to '`""`'.

`boolean inbox?` [`namespace-configuration` parameter]
There can be only one INBOX, and this setting defines which namespace has it. Defaults to '`#f`'.

`boolean hidden?` [`namespace-configuration` parameter]
If namespace is hidden, it's not advertised to clients via NAMESPACE extension. You'll most likely also want to set '`list? #f`'. This is mostly useful when converting from another server with different namespaces which you want to deprecate but still keep working. For example you can create hidden namespaces with prefixes '`~/mail/`', '`~%u/mail/`' and '`mail/`'. Defaults to '`#f`'.

`boolean list?` [`namespace-configuration` parameter]
Show the mailboxes under this namespace with the LIST command. This makes the namespace visible for clients that do not support the NAMESPACE extension. The special `children` value lists child mailboxes, but hides the namespace prefix. Defaults to '`#t`'.

`boolean subscriptions?` [`namespace-configuration` parameter]
Namespace handles its own subscriptions. If set to `#f`, the parent namespace handles them. The empty prefix should always have this as `#t`). Defaults to '`#t`'.

`mailbox-configuration-list mailboxes` [`namespace-configuration` parameter]
List of predefined mailboxes in this namespace. Defaults to '`()`'.

Available `mailbox-configuration` fields are:

`string name` [`mailbox-configuration` parameter]
Name for this mailbox.

`string auto` [`mailbox-configuration` parameter]
'`create`' will automatically create this mailbox. '`subscribe`' will both create and subscribe to the mailbox. Defaults to '`"no"`'.

`space-separated-string-list special-use` [`mailbox-configuration` parameter]
List of IMAP `SPECIAL-USE` attributes as specified by RFC 6154. Valid values are `\All`, `\Archive`, `\Drafts`, `\Flagged`, `\Junk`, `\Sent`, and `\Trash`. Defaults to '`()`'.

`file-name base-dir` [`dovecot-configuration` parameter]
Base directory where to store runtime data. Defaults to '`"/var/run/dovecot/"`'.

`string login-greeting` [`dovecot-configuration` parameter]
Greeting message for clients. Defaults to '`"Dovecot ready."`'.

`space-separated-string-list login-trusted-networks` [`dovecot-configuration` parameter]
List of trusted network ranges. Connections from these IPs are allowed to override their IP addresses and ports (for logging and for authentication checks). '`disable-plaintext-auth`' is also ignored for these networks. Typically you would specify your IMAP proxy servers here. Defaults to '`()`'.

`space-separated-string-list login-access-sockets` [`dovecot-configuration` parameter]
List of login access check sockets (e.g. tcpwrap). Defaults to '`()`'.

`boolean verbose-proctitle?` [`dovecot-configuration` parameter]
Show more verbose process titles (in ps). Currently shows user name and IP address. Useful for seeing who is actually using the IMAP processes (e.g. shared mailboxes or if the same uid is used for multiple accounts). Defaults to '`#f`'.

`boolean shutdown-clients?` [`dovecot-configuration` parameter]
Should all processes be killed when Dovecot master process shuts down. Setting this to `#f` means that Dovecot can be upgraded without forcing existing client connections to close (although that could also be a problem if the upgrade is e.g. due to a security fix). Defaults to '`#t`'.

`non-negative-integer` [`dovecot-configuration` parameter]
`doveadm-worker-count`
If non-zero, run mail commands via this many connections to doveadm server, instead of running them directly in the same process. Defaults to '`0`'.

`string doveadm-socket-path` [`dovecot-configuration` parameter]
UNIX socket or host:port used for connecting to doveadm server. Defaults to '`"doveadm-server"`'.

`space-separated-string-list` [`dovecot-configuration` parameter]
`import-environment`
List of environment variables that are preserved on Dovecot startup and passed down to all of its child processes. You can also give key=value pairs to always set specific settings.

`boolean disable-plaintext-auth?` [`dovecot-configuration` parameter]
Disable LOGIN command and all other plaintext authentications unless SSL/TLS is used (LOGINDISABLED capability). Note that if the remote IP matches the local IP (i.e. you're connecting from the same computer), the connection is considered secure and plaintext authentication is allowed. See also ssl=required setting. Defaults to '`#t`'.

`non-negative-integer auth-cache-size` [`dovecot-configuration` parameter]
Authentication cache size (e.g. '`#e10e6`'). 0 means it's disabled. Note that bsdauth, PAM and vpopmail require '`cache-key`' to be set for caching to be used. Defaults to '`0`'.

`string auth-cache-ttl` [`dovecot-configuration` parameter]
Time to live for cached data. After TTL expires the cached record is no longer used, *except* if the main database lookup returns internal failure. We also try to handle password changes automatically: If user's previous authentication was successful, but this one wasn't, the cache isn't used. For now this works only with plaintext authentication. Defaults to '`"1 hour"`'.

`string auth-cache-negative-ttl` [`dovecot-configuration` parameter]
TTL for negative hits (user not found, password mismatch). 0 disables caching them completely. Defaults to '`"1 hour"`'.

`space-separated-string-list` [`dovecot-configuration` parameter]
`auth-realms`
List of realms for SASL authentication mechanisms that need them. You can leave it empty if you don't want to support multiple realms. Many clients simply use the first one listed here, so keep the default realm first. Defaults to '`()`'.

`string auth-default-realm` [`dovecot-configuration` parameter]
Default realm/domain to use if none was specified. This is used for both SASL realms and appending @domain to username in plaintext logins. Defaults to '`""`'.

`string auth-username-chars` [`dovecot-configuration` parameter]
List of allowed characters in username. If the user-given username contains a character not listed in here, the login automatically fails. This is just an extra check to make sure user can't exploit any potential quote escaping vulnerabilities with SQL/LDAP databases. If you want to allow all characters, set this value to empty. Defaults to '`"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ01234567890.-_@"`'.

`string auth-username-translation` [`dovecot-configuration` parameter]
Username character translations before it's looked up from databases. The value contains series of from `->` to characters. For example '`#@/@`' means that '`#`' and '`/`' characters are translated to '`@`'. Defaults to '`""`'.

`string auth-username-format` [`dovecot-configuration` parameter]
Username formatting before it's looked up from databases. You can use the standard variables here, e.g. %Lu would lowercase the username, %n would drop away the domain if it was given, or '`%n-AT-%d`' would change the '`@`' into '`-AT-`'. This translation is done after '`auth-username-translation`' changes. Defaults to '`"%Lu"`'.

`string auth-master-user-separator` [`dovecot-configuration` parameter]
If you want to allow master users to log in by specifying the master username within the normal username string (i.e. not using SASL mechanism's support for it), you can specify the separator character here. The format is then <username><separator><master username>. UW-IMAP uses '`*`' as the separator, so that could be a good choice. Defaults to '`""`'.

`string auth-anonymous-username` [`dovecot-configuration` parameter]
Username to use for users logging in with ANONYMOUS SASL mechanism. Defaults to '`"anonymous"`'.

`non-negative-integer` [`dovecot-configuration` parameter]
`auth-worker-max-count`
Maximum number of dovecot-auth worker processes. They're used to execute blocking passdb and userdb queries (e.g. MySQL and PAM). They're automatically created and destroyed as needed. Defaults to '`30`'.

`string auth-gssapi-hostname` [`dovecot-configuration` parameter]
Host name to use in GSSAPI principal names. The default is to use the name returned by gethostname(). Use '`$ALL`' (with quotes) to allow all keytab entries. Defaults to '`""`'.

`string auth-krb5-keytab` [`dovecot-configuration` parameter]
Kerberos keytab to use for the GSSAPI mechanism. Will use the system default (usually `/etc/krb5.keytab`) if not specified. You may need to change the auth service to run as root to be able to read this file. Defaults to '`""`'.

`boolean auth-use-winbind?` [`dovecot-configuration` parameter]
Do NTLM and GSS-SPNEGO authentication using Samba's winbind daemon and '`ntlm-auth`' helper. <doc/wiki/Authentication/Mechanisms/Winbind.txt>. Defaults to '`#f`'.

`file-name auth-winbind-helper-path` [`dovecot-configuration` parameter]
Path for Samba's '`ntlm-auth`' helper binary. Defaults to '`"/usr/bin/ntlm_auth"`'.

`string auth-failure-delay` [`dovecot-configuration` parameter]
Time to delay before replying to failed authentications. Defaults to '`"2 secs"`'.

`boolean auth-ssl-require-client-cert?` [`dovecot-configuration` parameter]
Require a valid SSL client certificate or the authentication fails. Defaults to '`#f`'.

`boolean auth-ssl-username-from-cert?` [`dovecot-configuration` parameter]
Take the username from client's SSL certificate, using `X509_NAME_get_text_by_NID()` which returns the subject's DN's CommonName. Defaults to '`#f`'.

`space-separated-string-list` [`dovecot-configuration` parameter]
`auth-mechanisms`
List of wanted authentication mechanisms. Supported mechanisms are: '`plain`', '`login`', '`digest-md5`', '`cram-md5`', '`ntlm`', '`rpa`', '`apop`', '`anonymous`', '`gssapi`', '`otp`', '`skey`', and '`gss-spnego`'. NOTE: See also '`disable-plaintext-auth`' setting.

`space-separated-string-list` [`dovecot-configuration` parameter]
`director-servers`
List of IPs or hostnames to all director servers, including ourself. Ports can be specified as ip:port. The default port is the same as what director service's '`inet-listener`' is using. Defaults to '`()`'.

`space-separated-string-list` [`dovecot-configuration` parameter]
`director-mail-servers`
List of IPs or hostnames to all backend mail servers. Ranges are allowed too, like 10.0.0.10-10.0.0.30. Defaults to '`()`'.

`string director-user-expire` [`dovecot-configuration` parameter]
How long to redirect users to a specific server after it no longer has any connections. Defaults to '`"15 min"`'.

`non-negative-integer` [`dovecot-configuration` parameter]
`director-doveadm-port`
TCP/IP port that accepts doveadm connections (instead of director connections) If you enable this, you'll also need to add '`inet-listener`' for the port. Defaults to '`0`'.

`string director-username-hash` [`dovecot-configuration` parameter]
How the username is translated before being hashed. Useful values include %Ln if user can log in with or without @domain, %Ld if mailboxes are shared within domain. Defaults to '`"%Lu"`'.

`string log-path` [`dovecot-configuration` parameter]
Log file to use for error messages. '`syslog`' logs to syslog, '`/dev/stderr`' logs to stderr. Defaults to '`"syslog"`'.

`string info-log-path` [`dovecot-configuration` parameter]
Log file to use for informational messages. Defaults to '`log-path`'. Defaults to '`""`'.

`string debug-log-path` [`dovecot-configuration` parameter]
Log file to use for debug messages. Defaults to '`info-log-path`'. Defaults to '`""`'.

`string syslog-facility` [`dovecot-configuration` parameter]
Syslog facility to use if you're logging to syslog. Usually if you don't want to use '`mail`', you'll use local0..local7. Also other standard facilities are supported. Defaults to '`"mail"`'.

`boolean auth-verbose?` [`dovecot-configuration` parameter]
Log unsuccessful authentication attempts and the reasons why they failed. Defaults to '`#f`'.

`boolean auth-verbose-passwords?` [`dovecot-configuration` parameter]
In case of password mismatches, log the attempted password. Valid values are no, plain and sha1. sha1 can be useful for detecting brute force password attempts vs. user simply trying the same password over and over again. You can also truncate the value to n chars by appending `":n"` (e.g. sha1:6). Defaults to '`#f`'.

`boolean auth-debug?` [`dovecot-configuration` parameter]
Even more verbose logging for debugging purposes. Shows for example SQL queries. Defaults to '`#f`'.

`boolean auth-debug-passwords?` [`dovecot-configuration` parameter]
In case of password mismatches, log the passwords and used scheme so the problem can be debugged. Enabling this also enables '`auth-debug`'. Defaults to '`#f`'.

`boolean mail-debug?` [`dovecot-configuration` parameter]
Enable mail process debugging. This can help you figure out why Dovecot isn't finding your mails. Defaults to '`#f`'.

`boolean verbose-ssl?` [`dovecot-configuration` parameter]
Show protocol level SSL errors. Defaults to '`#f`'.

`string log-timestamp` [`dovecot-configuration` parameter]
Prefix for each line written to log file. % codes are in strftime(3) format. Defaults to '`"\"%b %d %H:%M:%S \""`'.

`space-separated-string-list login-log-format-elements` [`dovecot-configuration` parameter]
List of elements we want to log. The elements which have a non-empty variable value are joined together to form a comma-separated string.

`string login-log-format` [`dovecot-configuration` parameter]
Login log format. %s contains '`login-log-format-elements`' string, %$ contains the data we want to log. Defaults to '`"%$: %s"`'.

`string mail-log-prefix` [`dovecot-configuration` parameter]
Log prefix for mail processes. See doc/wiki/Variables.txt for list of possible variables you can use. Defaults to '`"\"%s(%u): \""`'.

`string deliver-log-format` [`dovecot-configuration` parameter]

Format to use for logging mail deliveries. You can use variables:

`%$`	Delivery status message (e.g. '`saved to INBOX`')
`%m`	Message-ID
`%s`	Subject
`%f`	From address
`%p`	Physical size
`%w`	Virtual size.

Defaults to '`"msgid=%m: %$"`'.

`string mail-location` [`dovecot-configuration` parameter]

Location for users' mailboxes. The default is empty, which means that Dovecot tries to find the mailboxes automatically. This won't work if the user doesn't yet have any mail, so you should explicitly tell Dovecot the full location.

If you're using mbox, giving a path to the INBOX file (e.g. /var/mail/%u) isn't enough. You'll also need to tell Dovecot where the other mailboxes are kept. This is called the "root mail directory", and it must be the first path given in the '`mail-location`' setting.

There are a few special variables you can use, eg.:

'`%u`'	username
'`%n`'	user part in user@domain, same as %u if there's no domain
'`%d`'	domain part in user@domain, empty if there's no domain
'`%h`'	home director

See doc/wiki/Variables.txt for full list. Some examples:

'`maildir:~/Maildir`'
'`mbox:~/mail:INBOX=/var/mail/%u`'
'`mbox:/var/mail/%d/%1n/%n:INDEX=/var/indexes/%d/%1n/%`'

Defaults to '`""`'.

`string mail-uid` [`dovecot-configuration` parameter]

System user and group used to access mails. If you use multiple, userdb can override these by returning uid or gid fields. You can use either numbers or names. <doc/wiki/UserIds.txt>. Defaults to '`""`'.

`string mail-gid` [`dovecot-configuration` parameter]

Defaults to '`""`'.

`string mail-privileged-group` [`dovecot-configuration` parameter]

Group to enable temporarily for privileged operations. Currently this is used only with INBOX when either its initial creation or dotlocking fails. Typically this is set to "mail" to give access to /var/mail. Defaults to '`""`'.

`string mail-access-groups` [`dovecot-configuration` parameter]
Grant access to these supplementary groups for mail processes. Typically these are used to set up access to shared mailboxes. Note that it may be dangerous to set these if users can create symlinks (e.g. if "mail" group is set here, ln -s /var/mail ~/mail/var could allow a user to delete others' mailboxes, or ln -s /secret/shared/box ~/mail/mybox would allow reading it). Defaults to '`""`'.

`boolean mail-full-filesystem-access?` [`dovecot-configuration` parameter]
Allow full file system access to clients. There's no access checks other than what the operating system does for the active UID/GID. It works with both maildir and mboxes, allowing you to prefix mailboxes names with e.g. /path/ or ~user/. Defaults to '`#f`'.

`boolean mmap-disable?` [`dovecot-configuration` parameter]
Don't use mmap() at all. This is required if you store indexes to shared file systems (NFS or clustered file system). Defaults to '`#f`'.

`boolean dotlock-use-excl?` [`dovecot-configuration` parameter]
Rely on '`O_EXCL`' to work when creating dotlock files. NFS supports '`O_EXCL`' since version 3, so this should be safe to use nowadays by default. Defaults to '`#t`'.

`string mail-fsync` [`dovecot-configuration` parameter]
When to use fsync() or fdatasync() calls:

`optimized`
: Whenever necessary to avoid losing important data

`always`
: Useful with e.g. NFS when write()s are delayed

`never`
: Never use it (best performance, but crashes can lose data).

Defaults to '`"optimized"`'.

`boolean mail-nfs-storage?` [`dovecot-configuration` parameter]
Mail storage exists in NFS. Set this to yes to make Dovecot flush NFS caches whenever needed. If you're using only a single mail server this isn't needed. Defaults to '`#f`'.

`boolean mail-nfs-index?` [`dovecot-configuration` parameter]
Mail index files also exist in NFS. Setting this to yes requires '`mmap-disable? #t`' and '`fsync-disable? #f`'. Defaults to '`#f`'.

`string lock-method` [`dovecot-configuration` parameter]
Locking method for index files. Alternatives are fcntl, flock and dotlock. Dotlocking uses some tricks which may create more disk I/O than other locking methods. NFS users: flock doesn't work, remember to change '`mmap-disable`'. Defaults to '`"fcntl"`'.

`file-name mail-temp-dir` [`dovecot-configuration` parameter]
Directory in which LDA/LMTP temporarily stores incoming mails >128 kB. Defaults to '`"/tmp"`'.

`non-negative-integer first-valid-uid` [`dovecot-configuration` parameter]
Valid UID range for users. This is mostly to make sure that users can't log in as daemons or other system users. Note that denying root logins is hardcoded to dovecot binary and can't be done even if '`first-valid-uid`' is set to 0. Defaults to '`500`'.

`non-negative-integer last-valid-uid` [`dovecot-configuration` parameter]
Defaults to '`0`'.

`non-negative-integer first-valid-gid` [`dovecot-configuration` parameter]
Valid GID range for users. Users having non-valid GID as primary group ID aren't allowed to log in. If user belongs to supplementary groups with non-valid GIDs, those groups are not set. Defaults to '`1`'.

`non-negative-integer last-valid-gid` [`dovecot-configuration` parameter]
Defaults to '`0`'.

`non-negative-integer mail-max-keyword-length` [`dovecot-configuration` parameter]
Maximum allowed length for mail keyword name. It's only forced when trying to create new keywords. Defaults to '`50`'.

`colon-separated-file-name-list valid-chroot-dirs` [`dovecot-configuration` parameter]
List of directories under which chrooting is allowed for mail processes (i.e. /var/mail will allow chrooting to /var/mail/foo/bar too). This setting doesn't affect '`login-chroot`' '`mail-chroot`' or auth chroot settings. If this setting is empty, `"/./"` in home dirs are ignored. WARNING: Never add directories here which local users can modify, that may lead to root exploit. Usually this should be done only if you don't allow shell access for users. <doc/wiki/Chrooting.txt>. Defaults to '`()`'.

`string mail-chroot` [`dovecot-configuration` parameter]
Default chroot directory for mail processes. This can be overridden for specific users in user database by giving /./ in user's home directory (e.g. /home/./user chroots into /home). Note that usually there is no real need to do chrooting, Dovecot doesn't allow users to access files outside their mail directory anyway. If your home directories are prefixed with the chroot directory, append `"/."` to '`mail-chroot`'. <doc/wiki/Chrooting.txt>. Defaults to '`""`'.

`file-name auth-socket-path` [`dovecot-configuration` parameter]
UNIX socket path to master authentication server to find users. This is used by imap (for shared users) and lda. Defaults to '`"/var/run/dovecot/auth-userdb"`'.

`file-name mail-plugin-dir` [`dovecot-configuration` parameter]
Directory where to look up mail plugins. Defaults to '`"/usr/lib/dovecot"`'.

`space-separated-string-list mail-plugins` [`dovecot-configuration` parameter]
List of plugins to load for all services. Plugins specific to IMAP, LDA, etc. are added to this list in their own .conf files. Defaults to '`()`'.

`non-negative-integer` [`dovecot-configuration` parameter]
`mail-cache-min-mail-count`
The minimum number of mails in a mailbox before updates are done to cache file. This allows optimizing Dovecot's behavior to do less disk writes at the cost of more disk reads. Defaults to '`0`'.

`string mailbox-idle-check-interval` [`dovecot-configuration` parameter]
When IDLE command is running, mailbox is checked once in a while to see if there are any new mails or other changes. This setting defines the minimum time to wait between those checks. Dovecot can also use dnotify, inotify and kqueue to find out immediately when changes occur. Defaults to '`"30 secs"`'.

`boolean mail-save-crlf?` [`dovecot-configuration` parameter]
Save mails with CR+LF instead of plain LF. This makes sending those mails take less CPU, especially with sendfile() syscall with Linux and FreeBSD. But it also creates a bit more disk I/O which may just make it slower. Also note that if other software reads the mboxes/maildirs, they may handle the extra CRs wrong and cause problems. Defaults to '`#f`'.

`boolean maildir-stat-dirs?` [`dovecot-configuration` parameter]
By default LIST command returns all entries in maildir beginning with a dot. Enabling this option makes Dovecot return only entries which are directories. This is done by stat()ing each entry, so it causes more disk I/O. (For systems setting struct '`dirent->d_type`' this check is free and it's done always regardless of this setting). Defaults to '`#f`'.

`boolean maildir-copy-with-hardlinks?` [`dovecot-configuration` parameter]
When copying a message, do it with hard links whenever possible. This makes the performance much better, and it's unlikely to have any side effects. Defaults to '`#t`'.

`boolean maildir-very-dirty-syncs?` [`dovecot-configuration` parameter]
Assume Dovecot is the only MUA accessing Maildir: Scan cur/ directory only when its mtime changes unexpectedly or when we can't find the mail otherwise. Defaults to '`#f`'.

`space-separated-string-list` [`dovecot-configuration` parameter]
`mbox-read-locks`
Which locking methods to use for locking mbox. There are four available:

`dotlock` Create <mailbox>.lock file. This is the oldest and most NFS-safe solution. If you want to use /var/mail/ like directory, the users will need write access to that directory.

`dotlock-try`
Same as dotlock, but if it fails because of permissions or because there isn't enough disk space, just skip it.

`fcntl` Use this if possible. Works with NFS too if lockd is used.

`flock` May not exist in all systems. Doesn't work with NFS.

`lockf` May not exist in all systems. Doesn't work with NFS.

You can use multiple locking methods; if you do the order they're declared in is important to avoid deadlocks if other MTAs/MUAs are using multiple locking methods as well. Some operating systems don't allow using some of them simultaneously.

`space-separated-string-list` [`dovecot-configuration` parameter]
`mbox-write-locks`

`string mbox-lock-timeout` [`dovecot-configuration` parameter]
Maximum time to wait for lock (all of them) before aborting. Defaults to '`"5 mins"`'.

`string mbox-dotlock-change-timeout` [`dovecot-configuration` parameter]
If dotlock exists but the mailbox isn't modified in any way, override the lock file after this much time. Defaults to '`"2 mins"`'.

`boolean mbox-dirty-syncs?` [`dovecot-configuration` parameter]
When mbox changes unexpectedly we have to fully read it to find out what changed. If the mbox is large this can take a long time. Since the change is usually just a newly appended mail, it'd be faster to simply read the new mails. If this setting is enabled, Dovecot does this but still safely fallbacks to re-reading the whole mbox file whenever something in mbox isn't how it's expected to be. The only real downside to this setting is that if some other MUA changes message flags, Dovecot doesn't notice it immediately. Note that a full sync is done with SELECT, EXAMINE, EXPUNGE and CHECK commands. Defaults to '`#t`'.

`boolean mbox-very-dirty-syncs?` [`dovecot-configuration` parameter]
Like '`mbox-dirty-syncs`', but don't do full syncs even with SELECT, EXAMINE, EXPUNGE or CHECK commands. If this is set, '`mbox-dirty-syncs`' is ignored. Defaults to '`#f`'.

`boolean mbox-lazy-writes?` [`dovecot-configuration` parameter]
Delay writing mbox headers until doing a full write sync (EXPUNGE and CHECK commands and when closing the mailbox). This is especially useful for POP3 where clients often delete all mails. The downside is that our changes aren't immediately visible to other MUAs. Defaults to '`#t`'.

`non-negative-integer` [`dovecot-configuration` parameter]
`mbox-min-index-size`
If mbox size is smaller than this (e.g. 100k), don't write index files. If an index file already exists it's still read, just not updated. Defaults to '`0`'.

`non-negative-integer` [`dovecot-configuration` parameter]
`mdbox-rotate-size`
Maximum dbox file size until it's rotated. Defaults to '`2000000`'.

`string mdbox-rotate-interval` [`dovecot-configuration` parameter]
Maximum dbox file age until it's rotated. Typically in days. Day begins from midnight, so 1d = today, 2d = yesterday, etc. 0 = check disabled. Defaults to '`"1d"`'.

`boolean mdbox-preallocate-space?` [`dovecot-configuration` parameter]
When creating new mdbox files, immediately preallocate their size to '`mdbox-rotate-size`'. This setting currently works only in Linux with some file systems (ext4, xfs). Defaults to '`#f`'.

`string mail-attachment-dir` [`dovecot-configuration` parameter]
sdbox and mdbox support saving mail attachments to external files, which also allows single instance storage for them. Other backends don't support this for now.

WARNING: This feature hasn't been tested much yet. Use at your own risk.

Directory root where to store mail attachments. Disabled, if empty. Defaults to '`""`'.

`non-negative-integer` [`dovecot-configuration` parameter]
`mail-attachment-min-size`
Attachments smaller than this aren't saved externally. It's also possible to write a plugin to disable saving specific attachments externally. Defaults to '`128000`'.

`string mail-attachment-fs` [`dovecot-configuration` parameter]
File system backend to use for saving attachments:

`posix` No SiS done by Dovecot (but this might help FS's own deduplication)

`sis posix` SiS with immediate byte-by-byte comparison during saving

`sis-queue posix`
SiS with delayed comparison and deduplication.

Defaults to '`"sis posix"`'.

`string mail-attachment-hash` [`dovecot-configuration` parameter]
Hash format to use in attachment filenames. You can add any text and variables: `%{md4}`, `%{md5}`, `%{sha1}`, `%{sha256}`, `%{sha512}`, `%{size}`. Variables can be truncated, e.g. `%{sha256:80}` returns only first 80 bits. Defaults to '`"%{sha1}"`'.

`non-negative-integer` [`dovecot-configuration` parameter]
`default-process-limit`
Defaults to '`100`'.

`non-negative-integer` [`dovecot-configuration` parameter]
`default-client-limit`
Defaults to '`1000`'.

`non-negative-integer` [`dovecot-configuration` parameter]
`default-vsz-limit`
Default VSZ (virtual memory size) limit for service processes. This is mainly intended to catch and kill processes that leak memory before they eat up everything. Defaults to '`256000000`'.

`string default-login-user` [`dovecot-configuration` parameter]
Login user is internally used by login processes. This is the most untrusted user in Dovecot system. It shouldn't have access to anything at all. Defaults to '`"dovenull"`'.

`string default-internal-user` [`dovecot-configuration` parameter]
Internal user is used by unprivileged processes. It should be separate from login user, so that login processes can't disturb other processes. Defaults to '`"dovecot"`'.

`string ssl?` [`dovecot-configuration` parameter]
SSL/TLS support: yes, no, required. <doc/wiki/SSL.txt>. Defaults to '`"required"`'.

`string ssl-cert` [`dovecot-configuration` parameter]
PEM encoded X.509 SSL/TLS certificate (public key). Defaults to '`"</etc/dovecot/default.pem"`'.

`string ssl-key` [`dovecot-configuration` parameter]
PEM encoded SSL/TLS private key. The key is opened before dropping root privileges, so keep the key file unreadable by anyone but root. Defaults to '`"</etc/dovecot/private/default.pem"`'.

`string ssl-key-password` [`dovecot-configuration` parameter]
If key file is password protected, give the password here. Alternatively give it when starting dovecot with -p parameter. Since this file is often world-readable, you may want to place this setting instead to a different. Defaults to '`""`'.

`string ssl-ca` [`dovecot-configuration` parameter]
PEM encoded trusted certificate authority. Set this only if you intend to use '`ssl-verify-client-cert? #t`'. The file should contain the CA certificate(s) followed by the matching CRL(s). (e.g. '`ssl-ca </etc/ssl/certs/ca.pem`'). Defaults to '`""`'.

`boolean ssl-require-crl?` [`dovecot-configuration` parameter]
Require that CRL check succeeds for client certificates. Defaults to '`#t`'.

`boolean ssl-verify-client-cert?` [`dovecot-configuration` parameter]
Request client to send a certificate. If you also want to require it, set '`auth-ssl-require-client-cert? #t`' in auth section. Defaults to '`#f`'.

`string ssl-cert-username-field` [`dovecot-configuration` parameter]
Which field from certificate to use for username. commonName and x500UniqueIdentifier are the usual choices. You'll also need to set '`auth-ssl-username-from-cert? #t`'. Defaults to '`"commonName"`'.

`hours ssl-parameters-regenerate` [`dovecot-configuration` parameter]
How often to regenerate the SSL parameters file. Generation is quite CPU intensive operation. The value is in hours, 0 disables regeneration entirely. Defaults to '`168`'.

`string ssl-protocols` [`dovecot-configuration` parameter]
SSL protocols to use. Defaults to '`"!SSLv2"`'.

`string ssl-cipher-list` [`dovecot-configuration` parameter]
SSL ciphers to use. Defaults to '`"ALL:!LOW:!SSLv2:!EXP:!aNULL"`'.

`string ssl-crypto-device` [`dovecot-configuration` parameter]
SSL crypto device to use, for valid values run "openssl engine". Defaults to '`""`'.

`string postmaster-address` [`dovecot-configuration` parameter]
Address to use when sending rejection mails. %d expands to recipient domain. Defaults to '`"postmaster@%d"`'.

`string hostname` [`dovecot-configuration` parameter]
Hostname to use in various parts of sent mails (e.g. in Message-Id) and in LMTP replies. Default is the system's real hostname@domain. Defaults to '`""`'.

`boolean quota-full-tempfail?` [`dovecot-configuration` parameter]
If user is over quota, return with temporary failure instead of bouncing the mail. Defaults to '`#f`'.

`file-name sendmail-path` [`dovecot-configuration` parameter]
Binary to use for sending mails. Defaults to '`"/usr/sbin/sendmail"`'.

`string submission-host` [`dovecot-configuration` parameter]
If non-empty, send mails via this SMTP host[:port] instead of sendmail. Defaults to '`""`'.

`string rejection-subject` [`dovecot-configuration` parameter]
Subject: header to use for rejection mails. You can use the same variables as for '`rejection-reason`' below. Defaults to '`"Rejected: %s"`'.

`string rejection-reason` [`dovecot-configuration` parameter]
Human readable error message for rejection mails. You can use variables:

`%n` CRLF

`%r` reason

`%s` original subject

`%t` recipient

Defaults to '`"Your message to <%t> was automatically rejected:%n%r"`'.

`string recipient-delimiter` [`dovecot-configuration` parameter]
Delimiter character between local-part and detail in email address. Defaults to '`"+"`'.

`string lda-original-recipient-header` [`dovecot-configuration` parameter]
Header where the original recipient address (SMTP's RCPT TO: address) is taken from if not available elsewhere. With dovecot-lda -a parameter overrides this. A commonly used header for this is X-Original-To. Defaults to '`""`'.

`boolean lda-mailbox-autocreate?` [`dovecot-configuration` parameter]
Should saving a mail to a nonexistent mailbox automatically create it?. Defaults to '`#f`'.

`boolean lda-mailbox-autosubscribe?` [`dovecot-configuration` parameter]
Should automatically created mailboxes be also automatically subscribed?. Defaults to '`#f`'.

`non-negative-integer` [`dovecot-configuration` parameter]
`imap-max-line-length`
Maximum IMAP command line length. Some clients generate very long command lines with huge mailboxes, so you may need to raise this if you get "Too long argument" or "IMAP command line too large" errors often. Defaults to '`64000`'.

`string imap-logout-format` [`dovecot-configuration` parameter]
IMAP logout format string:

`%i` total number of bytes read from client

`%o` total number of bytes sent to client.

Defaults to ‘`"in=%i out=%o"`’.

`string imap-capability` [`dovecot-configuration` parameter]
Override the IMAP CAPABILITY response. If the value begins with ’`+`’, add the given capabilities on top of the defaults (e.g. `+XFOO XBAR`). Defaults to ‘`""`’.

`string imap-idle-notify-interval` [`dovecot-configuration` parameter]
How long to wait between "OK Still here" notifications when client is IDLEing. Defaults to ‘`"2 mins"`’.

`string imap-id-send` [`dovecot-configuration` parameter]
ID field names and values to send to clients. Using * as the value makes Dovecot use the default value. The following fields have default values currently: name, version, os, os-version, support-url, support-email. Defaults to ‘`""`’.

`string imap-id-log` [`dovecot-configuration` parameter]
ID fields sent by client to log. * means everything. Defaults to ‘`""`’.

`space-separated-string-list imap-client-workarounds` [`dovecot-configuration` parameter]
Workarounds for various client bugs:

`delay-newmail`
Send EXISTS/RECENT new mail notifications only when replying to NOOP and CHECK commands. Some clients ignore them otherwise, for example OSX Mail (<v2.1). Outlook Express breaks more badly though, without this it may show user "Message no longer in server" errors. Note that OE6 still breaks even with this workaround if synchronization is set to "Headers Only".

`tb-extra-mailbox-sep`
Thunderbird gets somehow confused with LAYOUT=fs (mbox and dbox) and adds extra ‘`/`’ suffixes to mailbox names. This option causes Dovecot to ignore the extra ‘`/`’ instead of treating it as invalid mailbox name.

`tb-lsub-flags`
Show \Noselect flags for LSUB replies with LAYOUT=fs (e.g. mbox). This makes Thunderbird realize they aren’t selectable and show them greyed out, instead of only later giving "not selectable" popup error.

Defaults to ‘`()`’.

`string imap-urlauth-host` [`dovecot-configuration` parameter]
Host allowed in URLAUTH URLs sent by client. "*" allows all. Defaults to ‘`""`’.

Whew! Lots of configuration options. The nice thing about it though is that GuixSD has a complete interface to Dovecot’s configuration language. This allows not only a nice way to declare configurations, but also offers reflective capabilities as well: users can write code to inspect and transform configurations from within Scheme.

However, it could be that you just want to get a `dovecot.conf` up and running. In that case, you can pass an `opaque-dovecot-configuration` as the `#:config` parameter to `dovecot-service`. As its name indicates, an opaque configuration does not have easy reflective capabilities.

Available `opaque-dovecot-configuration` fields are:

`package dovecot` [`opaque-dovecot-configuration` parameter]
The dovecot package.

`string string` [`opaque-dovecot-configuration` parameter]
The contents of the `dovecot.conf`, as a string.

For example, if your `dovecot.conf` is just the empty string, you could instantiate a dovecot service like this:

```
(dovecot-service #:config
                 (opaque-dovecot-configuration
                  (string "")))
```

OpenSMTPD Service

`opensmtpd-service-type` [Scheme Variable]
This is the type of the OpenSMTPD (`https://www.opensmtpd.org`) service, whose value should be an `opensmtpd-configuration` object as in this example:

```
(service opensmtpd-service-type
         (opensmtpd-configuration
           (config-file (local-file "./my-smtpd.conf"))))
```

`opensmtpd-configuration` [Data Type]
Data type representing the configuration of opensmtpd.

`package` (default: *opensmtpd*)
Package object of the OpenSMTPD SMTP server.

`config-file` (default: *%default-opensmtpd-file*)
File-like object of the OpenSMTPD configuration file to use. By default it listens on the loopback network interface, and allows for mail from users and daemons on the local machine, as well as permitting email to remote servers. Run `man smtpd.conf` for more information.

Exim Service

`exim-service-type` [Scheme Variable]
This is the type of the Exim (`https://exim.org`) mail transfer agent (MTA), whose value should be an `exim-configuration` object as in this example:

```
(service exim-service-type
         (exim-configuration
           (config-file (local-file "./my-exim.conf"))))
```

In order to use an `exim-service-type` service you must also have a `mail-aliases-service-type` service present in your `operating-system` (even if it has no aliases).

`exim-configuration` [Data Type]

Data type representing the configuration of exim.

`package` (default: *exim*)
: Package object of the Exim server.

`config-file` (default: `#f`)
: File-like object of the Exim configuration file to use. If its value is `#f` then use the default configuration file from the package provided in `package`. The resulting configuration file is loaded after setting the `exim_user` and `exim_group` configuration variables.

Mail Aliases Service

`mail-aliases-service-type` [Scheme Variable]

This is the type of the service which provides `/etc/aliases`, specifying how to deliver mail to users on this system.

```
(service mail-aliases-service-type
         '(("postmaster" "bob")
           ("bob" "bob@example.com" "bob@example2.com")))
```

The configuration for a `mail-aliases-service-type` service is an association list denoting how to deliver mail that comes to this system. Each entry is of the form `(alias addresses ...)`, with `alias` specifying the local alias and `addresses` specifying where to deliver this user's mail.

The aliases aren't required to exist as users on the local system. In the above example, there doesn't need to be a `postmaster` entry in the `operating-system`'s `user-accounts` in order to deliver the `postmaster` mail to `bob` (which subsequently would deliver mail to `bob@example.com` and `bob@example2.com`).

6.2.7.10 Messaging Services

The `(gnu services messaging)` module provides Guix service definitions for messaging services: currently only Prosody is supported.

Prosody Service

`prosody-service-type` [Scheme Variable]

This is the type for the Prosody XMPP communication server (`http://prosody.im`). Its value must be a `prosody-configuration` record as in this example:

```
(service prosody-service-type
         (prosody-configuration
          (modules-enabled (cons "groups" %default-modules-enabled))
          (int-components
           (list
            (int-component-configuration
             (hostname "conference.example.net")
             (plugin "muc")
             (mod-muc (mod-muc-configuration)))))
```

```
(virtualhosts
 (list
  (virtualhost-configuration
   (domain "example.net"))))))
```

See below for details about `prosody-configuration`.

By default, Prosody does not need much configuration. Only one `virtualhosts` field is needed: it specifies the domain you wish Prosody to serve.

Prosodyctl will help you generate X.509 certificates and keys:

```
prosodyctl cert request example.net
```

The available configuration parameters follow. Each parameter definition is preceded by its type; for example, '`string-list foo`' indicates that the `foo` parameter should be specified as a list of strings. Types starting with `maybe-` denote parameters that won't show up in `prosody.cfg.lua` when their value is `'disabled`.

There is also a way to specify the configuration as a string, if you have an old `prosody.cfg.lua` file that you want to port over from some other system; see the end for more details.

Available `prosody-configuration` fields are:

`package prosody` [`prosody-configuration` parameter]
The Prosody package.

`file-name data-path` [`prosody-configuration` parameter]
Location of the Prosody data storage directory. See `http://prosody.im/doc/configure`. Defaults to '`"/var/lib/prosody"`'.

`file-name-list plugin-paths` [`prosody-configuration` parameter]
Additional plugin directories. They are searched in all the specified paths in order. See `http://prosody.im/doc/plugins_directory`. Defaults to '`()`'.

`string-list admins` [`prosody-configuration` parameter]
This is a list of accounts that are admins for the server. Note that you must create the accounts separately. See `http://prosody.im/doc/admins` and `http://prosody.im/doc/creating_accounts`. Example: `(admins '("user1@example.com" "user2@example.net"))` Defaults to '`()`'.

`boolean use-libevent?` [`prosody-configuration` parameter]
Enable use of libevent for better performance under high load. See `http://prosody.im/doc/libevent`. Defaults to '`#f`'.

`module-list modules-enabled` [`prosody-configuration` parameter]
This is the list of modules Prosody will load on startup. It looks for `mod_modulename.lua` in the plugins folder, so make sure that exists too. Documentation on modules can be found at: `http://prosody.im/doc/modules`. Defaults to '`%default-modules-enabled`'.

`string-list modules-disabled` [`prosody-configuration` parameter]
'`"offline"`', '`"c2s"`' and '`"s2s"`' are auto-loaded, but should you want to disable them then add them to this list. Defaults to '`()`'.

`file-name groups-file` [`prosody-configuration` parameter]
Path to a text file where the shared groups are defined. If this path is empty then '`mod_groups`' does nothing. See `http://prosody.im/doc/modules/mod_groups`. Defaults to '`"/var/lib/prosody/sharedgroups.txt"`'.

`boolean allow-registration?` [`prosody-configuration` parameter]
Disable account creation by default, for security. See `http://prosody.im/doc/creating_accounts`. Defaults to '`#f`'.

`maybe-ssl-configuration ssl` [`prosody-configuration` parameter]
These are the SSL/TLS-related settings. Most of them are disabled so to use Prosody's defaults. If you do not completely understand these options, do not add them to your config, it is easy to lower the security of your server using them. See `http://prosody.im/doc/advanced_ssl_config`.

Available `ssl-configuration` fields are:

`maybe-string protocol` [`ssl-configuration` parameter]
This determines what handshake to use.

`file-name key` [`ssl-configuration` parameter]
Path to your private key file, relative to `/etc/prosody`. Defaults to '`"/etc/prosody/certs/key.pem"`'.

`file-name certificate` [`ssl-configuration` parameter]
Path to your certificate file, relative to `/etc/prosody`. Defaults to '`"/etc/prosody/certs/cert.pem"`'.

`file-name capath` [`ssl-configuration` parameter]
Path to directory containing root certificates that you wish Prosody to trust when verifying the certificates of remote servers. Defaults to '`"/etc/ssl/certs"`'.

`maybe-file-name cafile` [`ssl-configuration` parameter]
Path to a file containing root certificates that you wish Prosody to trust. Similar to `capath` but with all certificates concatenated together.

`maybe-string-list verify` [`ssl-configuration` parameter]
A list of verification options (these mostly map to OpenSSL's `set_verify()` flags).

`maybe-string-list options` [`ssl-configuration` parameter]
A list of general options relating to SSL/TLS. These map to OpenSSL's `set_options()`. For a full list of options available in LuaSec, see the LuaSec source.

`maybe-non-negative-integer depth` [`ssl-configuration` parameter]
How long a chain of certificate authorities to check when looking for a trusted root certificate.

`maybe-string ciphers` [`ssl-configuration` parameter]
An OpenSSL cipher string. This selects what ciphers Prosody will offer to clients, and in what order.

`maybe-file-name dhparam` [`ssl-configuration` parameter]
A path to a file containing parameters for Diffie-Hellman key exchange. You can create such a file with: `openssl dhparam -out /etc/prosody/certs/dh-2048.pem 2048`

`maybe-string curve` [`ssl-configuration` parameter]
Curve for Elliptic curve Diffie-Hellman. Prosody's default is '`"secp384r1"`'.

`maybe-string-list verifyext` [`ssl-configuration` parameter]
A list of "extra" verification options.

`maybe-string password` [`ssl-configuration` parameter]
Password for encrypted private keys.

`boolean c2s-require-encryption?` [`prosody-configuration` parameter]
Whether to force all client-to-server connections to be encrypted or not. See `http://prosody.im/doc/modules/mod_tls`. Defaults to '`#f`'.

`boolean s2s-require-encryption?` [`prosody-configuration` parameter]
Whether to force all server-to-server connections to be encrypted or not. See `http://prosody.im/doc/modules/mod_tls`. Defaults to '`#f`'.

`boolean s2s-secure-auth?` [`prosody-configuration` parameter]
Whether to require encryption and certificate authentication. This provides ideal security, but requires servers you communicate with to support encryption AND present valid, trusted certificates. See `http://prosody.im/doc/s2s#security`. Defaults to '`#f`'.

`string-list s2s-insecure-domains` [`prosody-configuration` parameter]
Many servers don't support encryption or have invalid or self-signed certificates. You can list domains here that will not be required to authenticate using certificates. They will be authenticated using DNS. See `http://prosody.im/doc/s2s#security`. Defaults to '`()`'.

`string-list s2s-secure-domains` [`prosody-configuration` parameter]
Even if you leave `s2s-secure-auth?` disabled, you can still require valid certificates for some domains by specifying a list here. See `http://prosody.im/doc/s2s#security`. Defaults to '`()`'.

`string authentication` [`prosody-configuration` parameter]
Select the authentication backend to use. The default provider stores passwords in plaintext and uses Prosody's configured data storage to store the authentication data. If you do not trust your server please see `http://prosody.im/doc/modules/mod_auth_internal_hashed` for information about using the hashed backend. See also `http://prosody.im/doc/authentication` Defaults to '`"internal_plain"`'.

`maybe-string log` [`prosody-configuration` parameter]
Set logging options. Advanced logging configuration is not yet supported by the GuixSD Prosody Service. See `http://prosody.im/doc/logging`. Defaults to '`"*syslog"`'.

`file-name pidfile` [`prosody-configuration` parameter]
File to write pid in. See `http://prosody.im/doc/modules/mod_posix`. Defaults to '`"/var/run/prosody/prosody.pid"`'.

`virtualhost-configuration-list` [`prosody-configuration` parameter]
`virtualhosts`
A host in Prosody is a domain on which user accounts can be created. For example if you want your users to have addresses like '`"john.smith@example.com"`' then you need to add a host '`"example.com"`'. All options in this list will apply only to this host.

Note: the name "virtual" host is used in configuration to avoid confusion with the actual physical host that Prosody is installed on. A single Prosody instance can serve many domains, each one defined as a VirtualHost entry in Prosody's configuration. Conversely a server that hosts a single domain would have just one VirtualHost entry.

See `http://prosody.im/doc/configure#virtual_host_settings`.

Available `virtualhost-configuration` fields are:

all these `prosody-configuration` fields: `admins`, `use-libevent?`, `modules-enabled`, `modules-disabled`, `groups-file`, `allow-registration?`, `ssl`, `c2s-require-encryption?`, `s2s-require-encryption?`, `s2s-secure-auth?`, `s2s-insecure-domains`, `s2s-secure-domains`, `authentication`, `log`, plus:

`string domain` [`virtualhost-configuration` parameter]
Domain you wish Prosody to serve.

`int-component-configuration-list` [`prosody-configuration` parameter]
`int-components`
Components are extra services on a server which are available to clients, usually on a subdomain of the main server (such as '`"mycomponent.example.com"`'). Example components might be chatroom servers, user directories, or gateways to other protocols.

Internal components are implemented with Prosody-specific plugins. To add an internal component, you simply fill the hostname field, and the plugin you wish to use for the component.

See `http://prosody.im/doc/components`. Defaults to '`()`'.

Available `int-component-configuration` fields are:

all these `prosody-configuration` fields: `admins`, `use-libevent?`, `modules-enabled`, `modules-disabled`, `groups-file`, `allow-registration?`, `ssl`, `c2s-require-encryption?`, `s2s-require-encryption?`, `s2s-secure-auth?`, `s2s-insecure-domains`, `s2s-secure-domains`, `authentication`, `log`, plus:

`string hostname` [`int-component-configuration` parameter]
Hostname of the component.

`string plugin` [`int-component-configuration` parameter]
Plugin you wish to use for the component.

`maybe-mod-muc-configuration` [`int-component-configuration` parameter]
`mod-muc`

Multi-user chat (MUC) is Prosody's module for allowing you to create hosted chatrooms/conferences for XMPP users.

General information on setting up and using multi-user chatrooms can be found in the "Chatrooms" documentation (`http://prosody.im/doc/chatrooms`), which you should read if you are new to XMPP chatrooms.

See also `http://prosody.im/doc/modules/mod_muc`.

Available `mod-muc-configuration` fields are:

`string name` [`mod-muc-configuration` parameter]

The name to return in service discovery responses. Defaults to '`"Prosody Chatrooms"`'.

`string-or-boolean` [`mod-muc-configuration` parameter]
`restrict-room-creation`

If '`#t`', this will only allow admins to create new chatrooms. Otherwise anyone can create a room. The value '`"local"`' restricts room creation to users on the service's parent domain. E.g. '`user@example.com`' can create rooms on '`rooms.example.com`'. The value '`"admin"`' restricts to service administrators only. Defaults to '`#f`'.

`non-negative-integer` [`mod-muc-configuration` parameter]
`max-history-messages`

Maximum number of history messages that will be sent to the member that has just joined the room. Defaults to '`20`'.

`ext-component-configuration-list` [`prosody-configuration` parameter]
`ext-components`

External components use XEP-0114, which most standalone components support. To add an external component, you simply fill the hostname field. See `http://prosody.im/doc/components`. Defaults to '`()`'.

Available `ext-component-configuration` fields are:

all these `prosody-configuration` fields: `admins`, `use-libevent?`, `modules-enabled`, `modules-disabled`, `groups-file`, `allow-registration?`, `ssl`, `c2s-require-encryption?`, `s2s-require-encryption?`, `s2s-secure-auth?`, `s2s-insecure-domains`, `s2s-secure-domains`, `authentication`, `log`, plus:

`string component-secret` [`ext-component-configuration` parameter]

Password which the component will use to log in.

`string hostname` [`ext-component-configuration` parameter]

Hostname of the component.

`non-negative-integer-list` [`prosody-configuration` parameter]
`component-ports`

Port(s) Prosody listens on for component connections.

`string component-interface` [`prosody-configuration` parameter]
Interface Prosody listens on for component connections. Defaults to '`"127.0.0.1"`'.

It could be that you just want to get a `prosody.cfg.lua` up and running. In that case, you can pass an `opaque-prosody-configuration` record as the value of `prosody-service-type`. As its name indicates, an opaque configuration does not have easy reflective capabilities. Available `opaque-prosody-configuration` fields are:

`package prosody` [`opaque-prosody-configuration` parameter]
The prosody package.

`string prosody.cfg.lua` [`opaque-prosody-configuration` parameter]
The contents of the `prosody.cfg.lua` to use.

For example, if your `prosody.cfg.lua` is just the empty string, you could instantiate a prosody service like this:

```
(service prosody-service-type
         (opaque-prosody-configuration
          (prosody.cfg.lua "")))
```

6.2.7.11 Kerberos Services

The `(gnu services kerberos)` module provides services relating to the authentication protocol *Kerberos*.

Krb5 Service

Programs using a Kerberos client library normally expect a configuration file in `/etc/krb5.conf`. This service generates such a file from a definition provided in the operating system declaration. It does not cause any daemon to be started.

No "keytab" files are provided by this service—you must explicitly create them. This service is known to work with the MIT client library, `mit-krb5`. Other implementations have not been tested.

`krb5-service-type` [Scheme Variable]
A service type for Kerberos 5 clients.

Here is an example of its use:

```
(service krb5-service-type
         (krb5-configuration
          (default-realm "EXAMPLE.COM")
          (allow-weak-crypto? #t)
          (realms (list
                   (krb5-realm
                    (name "EXAMPLE.COM")
                    (admin-server "groucho.example.com")
                    (kdc "karl.example.com"))
                   (krb5-realm
                    (name "ARGRX.EDU")
                    (admin-server "kerb-admin.argrx.edu")
```

```
(kdc "keys.argrx.edu"))))))
```

This example provides a Kerberos 5 client configuration which:

- Recognizes two realms, *viz:* "EXAMPLE.COM" and "ARGRX.EDU", both of which have distinct administration servers and key distribution centers;
- Will default to the realm "EXAMPLE.COM" if the realm is not explicitly specified by clients;
- Accepts services which only support encryption types known to be weak.

The `krb5-realm` and `krb5-configuration` types have many fields. Only the most commonly used ones are described here. For a full list, and more detailed explanation of each, see the MIT krb5.conf documentation.

`krb5-realm` [Data Type]

`name` This field is a string identifying the name of the realm. A common convention is to use the fully qualified DNS name of your organization, converted to upper case.

`admin-server` This field is a string identifying the host where the administration server is running.

`kdc` This field is a string identifying the key distribution center for the realm.

`krb5-configuration` [Data Type]

`allow-weak-crypto?` (default: `#f`) If this flag is `#t` then services which only offer encryption algorithms known to be weak will be accepted.

`default-realm` (default: `#f`) This field should be a string identifying the default Kerberos realm for the client. You should set this field to the name of your Kerberos realm. If this value is `#f` then a realm must be specified with every Kerberos principal when invoking programs such as `kinit`.

`realms` This should be a non-empty list of `krb5-realm` objects, which clients may access. Normally, one of them will have a `name` field matching the `default-realm` field.

PAM krb5 Service

The `pam-krb5` service allows for login authentication and password management via Kerberos. You will need this service if you want PAM enabled applications to authenticate users using Kerberos.

`pam-krb5-service-type` [Scheme Variable]

A service type for the Kerberos 5 PAM module.

`pam-krb5-configuration` [Data Type]

Data type representing the configuration of the Kerberos 5 PAM module This type has the following parameters:

`pam-krb5` (default: `pam-krb5`) The pam-krb5 package to use.

`minimum-uid` (default: `1000`)
: The smallest user ID for which Kerberos authentications should be attempted. Local accounts with lower values will silently fail to authenticate.

6.2.7.12 Web Services

The `(gnu services web)` module provides the following service:

`nginx-service` [*#:nginx nginx*] [*#:log-directory "/var/log/nginx"*] [*#:run-directory "/var/run/nginx"*] [*#:server-list '()*] [*#:upstream-list '()*] [*#:config-file* `#f`] [Scheme Procedure]

Return a service that runs *nginx*, the nginx web server.

The nginx daemon loads its runtime configuration from *config-file*. Log files are written to *log-directory* and temporary runtime data files are written to *run-directory*. For proper operation, these arguments should match what is in *config-file* to ensure that the directories are created when the service is activated.

As an alternative to using a *config-file*, *server-list* can be used to specify the list of *server blocks* required on the host and *upstream-list* can be used to specify a list of *upstream blocks* to configure. For this to work, use the default value for *config-file*.

At startup, `nginx` has not yet read its configuration file, so it uses a default file to log error messages. If it fails to load its configuration file, that is where error messages are logged. After the configuration file is loaded, the default error log file changes as per configuration. In our case, startup error messages can be found in `/var/run/nginx/logs/error.log`, and after configuration in `/var/log/nginx/error.log`. The second location can be changed with the *log-directory* configuration option.

`nginx-service-type` [Scheme Variable]

This is type for the nginx web server.

This service can be extended to add server blocks in addition to the default one, as in this example:

```
(simple-service 'my-extra-server nginx-service-type
                (list (nginx-server-configuration
                        (https-port #f)
                        (root "/srv/http/extra-website"))))
```

`nginx-server-configuration` [Data Type]

Data type representing the configuration of an nginx server block. This type has the following parameters:

`http-port` (default: `80`)
: Nginx will listen for HTTP connection on this port. Set it at `#f` if nginx should not listen for HTTP (non secure) connection for this *server block*.

`https-port` (default: `443`)
: Nginx will listen for HTTPS connection on this port. Set it at `#f` if nginx should not listen for HTTPS (secure) connection for this *server block*.

Note that nginx can listen for HTTP and HTTPS connections in the same *server block*.

`server-name` (default: `(list 'default)`)
A list of server names this server represents. `'default` represents the default server for connections matching no other server.

`root` (default: `"/srv/http"`)
Root of the website nginx will serve.

`locations` (default: `'()`)
A list of *nginx-location-configuration* or *nginx-named-location-configuration* records to use within this server block.

`index` (default: `(list "index.html")`)
Index files to look for when clients ask for a directory. If it cannot be found, Nginx will send the list of files in the directory.

`ssl-certificate` (default: `"/etc/nginx/cert.pem"`)
Where to find the certificate for secure connections. Set it to `#f` if you don't have a certificate or you don't want to use HTTPS.

`ssl-certificate-key` (default: `"/etc/nginx/key.pem"`)
Where to find the private key for secure connections. Set it to `#f` if you don't have a key or you don't want to use HTTPS.

`server-tokens?` (default: `#f`)
Whether the server should add its configuration to response.

6.2.7.13 VPN Services

The `(gnu services vpn)` module provides services related to *virtual private networks* (VPNs). It provides a *client* service for your machine to connect to a VPN, and a *servire* service for your machine to host a VPN. Both services use OpenVPN (`https://openvpn.net/`).

`openvpn-client-service` [*#:config* (*openvpn-client-configuration*)] [Scheme Procedure]
Return a service that runs `openvpn`, a VPN daemon, as a client.

`openvpn-server-service` [*#:config* (*openvpn-server-configuration*)] [Scheme Procedure]
Return a service that runs `openvpn`, a VPN daemon, as a server.

Both can be run simultaneously.

Available `openvpn-client-configuration` fields are:

`package openvpn` [`openvpn-client-configuration` parameter]
The OpenVPN package.

`string pid-file` [`openvpn-client-configuration` parameter]
The OpenVPN pid file.

Defaults to '`"/var/run/openvpn/openvpn.pid"`'.

`proto proto` [`openvpn-client-configuration` parameter]
The protocol (UDP or TCP) used to open a channel between clients and servers.

Defaults to '`udp`'.

`dev dev` [`openvpn-client-configuration` parameter]
The device type used to represent the VPN connection.

Defaults to '`tun`'.

`string ca` [`openvpn-client-configuration` parameter]
The certificate authority to check connections against.

Defaults to '`"/etc/openvpn/ca.crt"`'.

`string cert` [`openvpn-client-configuration` parameter]
The certificate of the machine the daemon is running on. It should be signed by the authority given in `ca`.

Defaults to '`"/etc/openvpn/client.crt"`'.

`string key` [`openvpn-client-configuration` parameter]
The key of the machine the daemon is running on. It must be the key whose certificate is `cert`.

Defaults to '`"/etc/openvpn/client.key"`'.

`boolean comp-lzo?` [`openvpn-client-configuration` parameter]
Whether to use the lzo compression algorithm.

Defaults to '`#t`'.

`boolean persist-key?` [`openvpn-client-configuration` parameter]
Don't re-read key files across SIGUSR1 or –ping-restart.

Defaults to '`#t`'.

`boolean persist-tun?` [`openvpn-client-configuration` parameter]
Don't close and reopen TUN/TAP device or run up/down scripts across SIGUSR1 or –ping-restart restarts.

Defaults to '`#t`'.

`number verbosity` [`openvpn-client-configuration` parameter]
Verbosity level.

Defaults to '`3`'.

`tls-auth-client tls-auth` [`openvpn-client-configuration` parameter]
Add an additional layer of HMAC authentication on top of the TLS control channel to protect against DoS attacks.

Defaults to '`#f`'.

`key-usage verify-key-usage?` [`openvpn-client-configuration` parameter]
Whether to check the server certificate has server usage extension.

Defaults to '`#t`'.

`bind bind?` [`openvpn-client-configuration` parameter]
Bind to a specific local port number.

Defaults to '`#f`'.

`resolv-retry resolv-retry?` [`openvpn-client-configuration` parameter]
Retry resolving server address.

Defaults to '`#t`'.

`openvpn-remote-list remote` [`openvpn-client-configuration` parameter]
A list of remote servers to connect to.

Defaults to '`()`'.

Available `openvpn-remote-configuration` fields are:

`string name` [`openvpn-remote-configuration` parameter]
Server name.

Defaults to '`"my-server"`'.

`number port` [`openvpn-remote-configuration` parameter]
Port number the server listens to.

Defaults to '`1194`'.

Available `openvpn-server-configuration` fields are:

`package openvpn` [`openvpn-server-configuration` parameter]
The OpenVPN package.

`string pid-file` [`openvpn-server-configuration` parameter]
The OpenVPN pid file.

Defaults to '`"/var/run/openvpn/openvpn.pid"`'.

`proto proto` [`openvpn-server-configuration` parameter]
The protocol (UDP or TCP) used to open a channel between clients and servers.

Defaults to '`udp`'.

`dev dev` [`openvpn-server-configuration` parameter]
The device type used to represent the VPN connection.

Defaults to '`tun`'.

`string ca` [`openvpn-server-configuration` parameter]
The certificate authority to check connections against.

Defaults to '`"/etc/openvpn/ca.crt"`'.

`string cert` [`openvpn-server-configuration` parameter]
The certificate of the machine the daemon is running on. It should be signed by the authority given in `ca`.

Defaults to '`"/etc/openvpn/client.crt"`'.

`string key` [`openvpn-server-configuration` parameter]
The key of the machine the daemon is running on. It must be the key whose certificate is `cert`.

Defaults to '`"/etc/openvpn/client.key"`'.

`boolean comp-lzo?` [`openvpn-server-configuration` parameter]
Whether to use the lzo compression algorithm.

Defaults to '`#t`'.

`boolean persist-key?` [`openvpn-server-configuration` parameter]
Don't re-read key files across SIGUSR1 or –ping-restart.

Defaults to '`#t`'.

`boolean persist-tun?` [`openvpn-server-configuration` parameter]
Don't close and reopen TUN/TAP device or run up/down scripts across SIGUSR1 or –ping-restart restarts.

Defaults to '`#t`'.

`number verbosity` [`openvpn-server-configuration` parameter]
Verbosity level.

Defaults to '`3`'.

`tls-auth-server tls-auth` [`openvpn-server-configuration` parameter]
Add an additional layer of HMAC authentication on top of the TLS control channel to protect against DoS attacks.

Defaults to '`#f`'.

`number port` [`openvpn-server-configuration` parameter]
Specifies the port number on which the server listens.

Defaults to '`1194`'.

`ip-mask server` [`openvpn-server-configuration` parameter]
An ip and mask specifying the subnet inside the virtual network.

Defaults to '`"10.8.0.0 255.255.255.0"`'.

`cidr6 server-ipv6` [`openvpn-server-configuration` parameter]
A CIDR notation specifying the IPv6 subnet inside the virtual network.

Defaults to '`#f`'.

`string dh` [`openvpn-server-configuration` parameter]
The Diffie-Hellman parameters file.

Defaults to '`"/etc/openvpn/dh2048.pem"`'.

`string ifconfig-pool-persist` [`openvpn-server-configuration` parameter]
The file that records client IPs.

Defaults to '`"/etc/openvpn/ipp.txt"`'.

`gateway redirect-gateway?` [`openvpn-server-configuration` parameter]
When true, the server will act as a gateway for its clients.

Defaults to '`#f`'.

`boolean client-to-client?` [`openvpn-server-configuration` parameter]
When true, clients are allowed to talk to each other inside the VPN.

Defaults to '`#f`'.

`keepalive keepalive` [`openvpn-server-configuration` parameter]
Causes ping-like messages to be sent back and forth over the link so that each side knows when the other side has gone down. `keepalive` requires a pair. The first element is the period of the ping sending, and the second element is the timeout before considering the other side down.

`number max-clients` [`openvpn-server-configuration` parameter]
The maximum number of clients.

Defaults to '`100`'.

`string status` [`openvpn-server-configuration` parameter]
The status file. This file shows a small report on current connection. It is truncated and rewritten every minute.

Defaults to '`"/var/run/openvpn/status"`'.

`openvpn-ccd-list client-config-dir` [`openvpn-server-configuration` parameter]
The list of configuration for some clients.

Defaults to '`()`'.

Available `openvpn-ccd-configuration` fields are:

`string name` [`openvpn-ccd-configuration` parameter]
Client name.

Defaults to '`"client"`'.

`ip-mask iroute` [`openvpn-ccd-configuration` parameter]
Client own network

Defaults to '`#f`'.

`ip-mask ifconfig-push` [`openvpn-ccd-configuration` parameter]
Client VPN IP.

Defaults to '`#f`'.

`nginx-upstream-configuration` [Data Type]
Data type representing the configuration of an nginx `upstream` block. This type has the following parameters:

`name` Name for this group of servers.

`servers` Specify the addresses of the servers in the group. The address can be specified as a IP address (e.g. '`127.0.0.1`'), domain name (e.g. '`backend1.example.com`') or a path to a UNIX socket using the prefix '`unix:`'. For addresses using an IP address or domain name, the default port is 80, and a different port can be specified explicitly.

`nginx-location-configuration` [Data Type]
Data type representing the configuration of an nginx `location` block. This type has the following parameters:

`uri` URI which this location block matches.

`body` Body of the location block, specified as a string. This can contain many configuration directives. For example, to pass requests to a upstream server group defined using an `nginx-upstream-configuration` block, the following directive would be specified in the body '`proxy_pass http://upstream-name;`'.

`nginx-named-location-configuration` [Data Type]
Data type representing the configuration of an nginx named location block. Named location blocks are used for request redirection, and not used for regular request processing. This type has the following parameters:

`name` Name to identify this location block.

`body` See [nginx-location-configuration body], page 193, as the body for named location blocks can be used in a similar way to the `nginx-location-configuration body`. One restriction is that the body of a named location block cannot contain location blocks.

6.2.7.14 Network File System

The `(gnu services nfs)` module provides the following services, which are most commonly used in relation to mounting or exporting directory trees as *network file systems* (NFS).

RPC Bind Service

The RPC Bind service provides a facility to map program numbers into universal addresses. Many NFS related services use this facility. Hence it is automatically started when a dependent service starts.

`rpcbind-service-type` [Scheme Variable]
A service type for the RPC portmapper daemon.

`rpcbind-configuration` [Data Type]
Data type representing the configuration of the RPC Bind Service. This type has the following parameters:

`rpcbind` (default: `rpcbind`)
The rpcbind package to use.

`warm-start?` (default: `#t`)
If this parameter is `#t`, then the daemon will read a state file on startup thus reloading state information saved by a previous instance.

Pipefs Pseudo File System

The pipefs file system is used to transfer NFS related data between the kernel and user space programs.

`pipefs-service-type` [Scheme Variable]
A service type for the pipefs pseudo file system.

`pipefs-configuration` [Data Type]
Data type representing the configuration of the pipefs pseudo file system service. This type has the following parameters:

`mount-point` (default: `"/var/lib/nfs/rpc_pipefs"`)
The directory to which the file system is to be attached.

GSS Daemon Service

The *global security system* (GSS) daemon provides strong security for RPC based protocols. Before exchanging RPC requests an RPC client must establish a security context. Typically this is done using the Kerberos command `kinit` or automatically at login time using PAM services (see Section 6.2.7.11 [Kerberos Services], page 185).

`gss-service-type` [Scheme Variable]
A service type for the Global Security System (GSS) daemon.

`gss-configuration` [Data Type]
Data type representing the configuration of the GSS daemon service. This type has the following parameters:

`nfs-utils` (default: `nfs-utils`)
The package in which the `rpc.gssd` command is to be found.

`pipefs-directory` (default: `"/var/lib/nfs/rpc_pipefs"`)
The directory where the pipefs file system is mounted.

IDMAP Daemon Service

The idmap daemon service provides mapping between user IDs and user names. Typically it is required in order to access file systems mounted via NFSv4.

`idmap-service-type` [Scheme Variable]
A service type for the Identity Mapper (IDMAP) daemon.

`idmap-configuration` [Data Type]
Data type representing the configuration of the IDMAP daemon service. This type has the following parameters:

`nfs-utils` (default: `nfs-utils`)
The package in which the `rpc.idmapd` command is to be found.

`pipefs-directory` (default: `"/var/lib/nfs/rpc_pipefs"`)
The directory where the pipefs file system is mounted.

`domain` (default: `#f`)
The local NFSv4 domain name. This must be a string or `#f`. If it is `#f` then the daemon will use the host's fully qualified domain name.

6.2.7.15 Continuous Integration

Cuirass (`https://notabug.org/mthl/cuirass`) is a continuous integration tool for Guix. It can be used both for development and for providing substitutes to others (see Section 3.3 [Substitutes], page 25).

The `(gnu services cuirass)` module provides the following service.

`cuirass-service-type` [Scheme Procedure]
The type of the Cuirass service. Its value must be a `cuirass-configuration` object, as described below.

To add build jobs, you have to set the `specifications` field of the configuration. Here is an example of a service defining a build job based on a specification that can be found in Cuirass source tree. This service polls the Guix repository and builds a subset of the Guix packages, as prescribed in the `gnu-system.scm` example spec:

```
(let ((spec #~((#:name . "guix")
               (#:url . "git://git.savannah.gnu.org/guix.git")
               (#:load-path . ".")

               ;; Here we must provide an absolute file name.
               ;; We take jobs from one of the examples provided
               ;; by Cuirass.
               (#:file . #$(file-append
                            cuirass
                            "/tests/gnu-system.scm"))

               (#:proc . hydra-jobs)
               (#:arguments (subset . "hello"))
               (#:branch . "master"))))
  (service cuirass-service-type
           (cuirass-configuration
            (specifications #~(list #$spec)))))
```

While information related to build jobs is located directly in the specifications, global settings for the `cuirass` process are accessible in other `cuirass-configuration` fields.

`cuirass-configuration` [Data Type]
Data type representing the configuration of Cuirass.

`log-file` (default: `"/var/log/cuirass.log"`)
Location of the log file.

`cache-directory` (default: `"/var/cache/cuirass"`)
Location of the repository cache.

`user` (default: `"cuirass"`)
Owner of the `cuirass` process.

`group` (default: `"cuirass"`)
Owner's group of the `cuirass` process.

`interval` (default: `60`)
: Number of seconds between the poll of the repositories followed by the Cuirass jobs.

`database` (default: `"/var/run/cuirass/cuirass.db"`)
: Location of sqlite database which contains the build results and previously added specifications.

`port` (default: `8080`)
: Port number used by the HTTP server.

`specifications` (default: `#~'()`)
: A gexp (see Section 4.6 [G-Expressions], page 56) that evaluates to a list of specifications, where a specification is an association list (see Section "Associations Lists" in *GNU Guile Reference Manual*) whose keys are keywords (`#:keyword-example`) as shown in the example above.

`use-substitutes?` (default: `#f`)
: This allows using substitutes to avoid building every dependencies of a job from source.

`one-shot?` (default: `#f`)
: Only evaluate specifications and build derivations once.

`load-path` (default: `'()`)
: This allows users to define their own packages and make them visible to cuirass as in `guix build` command.

`cuirass` (default: `cuirass`)
: The Cuirass package to use.

6.2.7.16 Power management Services

The `(gnu services pm)` module provides a Guix service definition for the Linux power management tool TLP.

TLP enables various powersaving modes in userspace and kernel. Contrary to `upower-service`, it is not a passive, monitoring tool, as it will apply custom settings each time a new power source is detected. More information can be found at TLP home page (`http://linrunner.de/en/tlp/tlp.html`).

`tlp-service-type` [Scheme Variable]
: The service type for the TLP tool. Its value should be a valid TLP configuration (see below). To use the default settings, simply write:

```
(service tlp-service-type)
```

By default TLP does not need much configuration but most TLP parameters can be tweaked using `tlp-configuration`.

Each parameter definition is preceded by its type; for example, '`boolean foo`' indicates that the `foo` parameter should be specified as a boolean. Types starting with `maybe-` denote parameters that won't show up in TLP config file when their value is `'disabled`.

Available `tlp-configuration` fields are:

`package tlp` [`tlp-configuration` parameter]
The TLP package.

`boolean tlp-enable?` [`tlp-configuration` parameter]
Set to true if you wish to enable TLP.
Defaults to '`#t`'.

`string tlp-default-mode` [`tlp-configuration` parameter]
Default mode when no power supply can be detected. Alternatives are AC and BAT.
Defaults to '`"AC"`'.

`non-negative-integer disk-idle-secs-on-ac` [`tlp-configuration` parameter]
Number of seconds Linux kernel has to wait after the disk goes idle, before syncing on AC.
Defaults to '`0`'.

`non-negative-integer` [`tlp-configuration` parameter]
`disk-idle-secs-on-bat`
Same as `disk-idle-ac` but on BAT mode.
Defaults to '`2`'.

`non-negative-integer` [`tlp-configuration` parameter]
`max-lost-work-secs-on-ac`
Dirty pages flushing periodicity, expressed in seconds.
Defaults to '`15`'.

`non-negative-integer` [`tlp-configuration` parameter]
`max-lost-work-secs-on-bat`
Same as `max-lost-work-secs-on-ac` but on BAT mode.
Defaults to '`60`'.

`maybe-space-separated-string-list` [`tlp-configuration` parameter]
`cpu-scaling-governor-on-ac`
CPU frequency scaling governor on AC mode. With intel_pstate driver, alternatives are powersave and performance. With acpi-cpufreq driver, alternatives are ondemand, powersave, performance and conservative.
Defaults to '`disabled`'.

`maybe-space-separated-string-list` [`tlp-configuration` parameter]
`cpu-scaling-governor-on-bat`
Same as `cpu-scaling-governor-on-ac` but on BAT mode.
Defaults to '`disabled`'.

`maybe-non-negative-integer` [`tlp-configuration` parameter]
`cpu-scaling-min-freq-on-ac`
Set the min available frequency for the scaling governor on AC.
Defaults to '`disabled`'.

`maybe-non-negative-integer` [`tlp-configuration` parameter]
`cpu-scaling-max-freq-on-ac`

Set the max available frequency for the scaling governor on AC.

Defaults to '`disabled`'.

`maybe-non-negative-integer` [`tlp-configuration` parameter]
`cpu-scaling-min-freq-on-bat`

Set the min available frequency for the scaling governor on BAT.

Defaults to '`disabled`'.

`maybe-non-negative-integer` [`tlp-configuration` parameter]
`cpu-scaling-max-freq-on-bat`

Set the max available frequency for the scaling governor on BAT.

Defaults to '`disabled`'.

`maybe-non-negative-integer` [`tlp-configuration` parameter]
`cpu-min-perf-on-ac`

Limit the min P-state to control the power dissipation of the CPU, in AC mode. Values are stated as a percentage of the available performance.

Defaults to '`disabled`'.

`maybe-non-negative-integer` [`tlp-configuration` parameter]
`cpu-max-perf-on-ac`

Limit the max P-state to control the power dissipation of the CPU, in AC mode. Values are stated as a percentage of the available performance.

Defaults to '`disabled`'.

`maybe-non-negative-integer` [`tlp-configuration` parameter]
`cpu-min-perf-on-bat`

Same as `cpu-min-perf-on-ac` on BAT mode.

Defaults to '`disabled`'.

`maybe-non-negative-integer` [`tlp-configuration` parameter]
`cpu-max-perf-on-bat`

Same as `cpu-max-perf-on-ac` on BAT mode.

Defaults to '`disabled`'.

`maybe-boolean cpu-boost-on-ac?` [`tlp-configuration` parameter]

Enable CPU turbo boost feature on AC mode.

Defaults to '`disabled`'.

`maybe-boolean cpu-boost-on-bat?` [`tlp-configuration` parameter]

Same as `cpu-boost-on-ac?` on BAT mode.

Defaults to '`disabled`'.

`boolean sched-powersave-on-ac?` [`tlp-configuration` parameter]

Allow Linux kernel to minimize the number of CPU cores/hyper-threads used under light load conditions.

Defaults to '`#f`'.

`boolean sched-powersave-on-bat?` [`tlp-configuration` parameter]
Same as `sched-powersave-on-ac?` but on BAT mode.

Defaults to '`#t`'.

`boolean nmi-watchdog?` [`tlp-configuration` parameter]
Enable Linux kernel NMI watchdog.

Defaults to '`#f`'.

`maybe-string phc-controls` [`tlp-configuration` parameter]
For Linux kernels with PHC patch applied, change CPU voltages. An example value would be '`"F:V F:V F:V F:V"`'.

Defaults to '`disabled`'.

`string energy-perf-policy-on-ac` [`tlp-configuration` parameter]
Set CPU performance versus energy saving policy on AC. Alternatives are performance, normal, powersave.

Defaults to '`"performance"`'.

`string energy-perf-policy-on-bat` [`tlp-configuration` parameter]
Same as `energy-perf-policy-ac` but on BAT mode.

Defaults to '`"powersave"`'.

`space-separated-string-list disks-devices` [`tlp-configuration` parameter]
Hard disk devices.

`space-separated-string-list disk-apm-level-on-ac` [`tlp-configuration` parameter]
Hard disk advanced power management level.

`space-separated-string-list disk-apm-level-on-bat` [`tlp-configuration` parameter]
Same as `disk-apm-bat` but on BAT mode.

`maybe-space-separated-string-list disk-spindown-timeout-on-ac` [`tlp-configuration` parameter]
Hard disk spin down timeout. One value has to be specified for each declared hard disk.

Defaults to '`disabled`'.

`maybe-space-separated-string-list disk-spindown-timeout-on-bat` [`tlp-configuration` parameter]
Same as `disk-spindown-timeout-on-ac` but on BAT mode.

Defaults to '`disabled`'.

`maybe-space-separated-string-list disk-iosched` [`tlp-configuration` parameter]
Select IO scheduler for disk devices. One value has to be specified for each declared hard disk. Example alternatives are cfq, deadline and noop.

Defaults to '`disabled`'.

`string sata-linkpwr-on-ac` [`tlp-configuration` parameter]
SATA aggressive link power management (ALPM) level. Alternatives are min_power, medium_power, max_performance.

Defaults to '`"max_performance"`'.

`string sata-linkpwr-on-bat` [`tlp-configuration` parameter]
Same as `sata-linkpwr-ac` but on BAT mode.

Defaults to '`"min_power"`'.

`maybe-string sata-linkpwr-blacklist` [`tlp-configuration` parameter]
Exclude specified SATA host devices for link power management.

Defaults to '`disabled`'.

`maybe-on-off-boolean` [`tlp-configuration` parameter]
`ahci-runtime-pm-on-ac?`
Enable Runtime Power Management for AHCI controller and disks on AC mode.

Defaults to '`disabled`'.

`maybe-on-off-boolean` [`tlp-configuration` parameter]
`ahci-runtime-pm-on-bat?`
Same as `ahci-runtime-pm-on-ac` on BAT mode.

Defaults to '`disabled`'.

`non-negative-integer` [`tlp-configuration` parameter]
`ahci-runtime-pm-timeout`
Seconds of inactivity before disk is suspended.

Defaults to '`15`'.

`string pcie-aspm-on-ac` [`tlp-configuration` parameter]
PCI Express Active State Power Management level. Alternatives are default, performance, powersave.

Defaults to '`"performance"`'.

`string pcie-aspm-on-bat` [`tlp-configuration` parameter]
Same as `pcie-aspm-ac` but on BAT mode.

Defaults to '`"powersave"`'.

`string radeon-power-profile-on-ac` [`tlp-configuration` parameter]
Radeon graphics clock speed level. Alternatives are low, mid, high, auto, default.

Defaults to '`"high"`'.

`string radeon-power-profile-on-bat` [`tlp-configuration` parameter]
Same as `radeon-power-ac` but on BAT mode.

Defaults to '`"low"`'.

`string radeon-dpm-state-on-ac` [`tlp-configuration` parameter]
Radeon dynamic power management method (DPM). Alternatives are battery, performance.

Defaults to '`"performance"`'.

`string radeon-dpm-state-on-bat` [`tlp-configuration` parameter]
Same as `radeon-dpm-state-ac` but on BAT mode.

Defaults to '`"battery"`'.

`string radeon-dpm-perf-level-on-ac` [`tlp-configuration` parameter]
Radeon DPM performance level. Alternatives are auto, low, high.

Defaults to '`"auto"`'.

`string radeon-dpm-perf-level-on-bat` [`tlp-configuration` parameter]
Same as `radeon-dpm-perf-ac` but on BAT mode.

Defaults to '`"auto"`'.

`on-off-boolean wifi-pwr-on-ac?` [`tlp-configuration` parameter]
Wifi power saving mode.

Defaults to '`#f`'.

`on-off-boolean wifi-pwr-on-bat?` [`tlp-configuration` parameter]
Same as `wifi-power-ac?` but on BAT mode.

Defaults to '`#t`'.

`y-n-boolean wol-disable?` [`tlp-configuration` parameter]
Disable wake on LAN.

Defaults to '`#t`'.

`non-negative-integer sound-power-save-on-ac` [`tlp-configuration` parameter]
Timeout duration in seconds before activating audio power saving on Intel HDA and AC97 devices. A value of 0 disables power saving.

Defaults to '`0`'.

`non-negative-integer sound-power-save-on-bat` [`tlp-configuration` parameter]
Same as `sound-powersave-ac` but on BAT mode.

Defaults to '`1`'.

`y-n-boolean sound-power-save-controller?` [`tlp-configuration` parameter]
Disable controller in powersaving mode on Intel HDA devices.

Defaults to '`#t`'.

`boolean bay-poweroff-on-bat?` [`tlp-configuration` parameter]
Enable optical drive in UltraBay/MediaBay on BAT mode. Drive can be powered on again by releasing (and reinserting) the eject lever or by pressing the disc eject button on newer models.

Defaults to '`#f`'.

`string bay-device` [`tlp-configuration` parameter]
Name of the optical drive device to power off.

Defaults to '`"sr0"`'.

`string runtime-pm-on-ac` [`tlp-configuration` parameter]
Runtime Power Management for PCI(e) bus devices. Alternatives are on and auto.

Defaults to '`"on"`'.

`string runtime-pm-on-bat` [`tlp-configuration` parameter]
Same as `runtime-pm-ac` but on BAT mode.

Defaults to '`"auto"`'.

`boolean runtime-pm-all?` [`tlp-configuration` parameter]
Runtime Power Management for all PCI(e) bus devices, except blacklisted ones.

Defaults to '`#t`'.

`maybe-space-separated-string-list runtime-pm-blacklist` [`tlp-configuration` parameter]
Exclude specified PCI(e) device addresses from Runtime Power Management.

Defaults to '`disabled`'.

`space-separated-string-list runtime-pm-driver-blacklist` [`tlp-configuration` parameter]
Exclude PCI(e) devices assigned to the specified drivers from Runtime Power Management.

`boolean usb-autosuspend?` [`tlp-configuration` parameter]
Enable USB autosuspend feature.

Defaults to '`#t`'.

`maybe-string usb-blacklist` [`tlp-configuration` parameter]
Exclude specified devices from USB autosuspend.

Defaults to '`disabled`'.

`boolean usb-blacklist-wwan?` [`tlp-configuration` parameter]
Exclude WWAN devices from USB autosuspend.

Defaults to '`#t`'.

`maybe-string usb-whitelist` [`tlp-configuration` parameter]
Include specified devices into USB autosuspend, even if they are already excluded by the driver or via `usb-blacklist-wwan?`.

Defaults to '`disabled`'.

`maybe-boolean usb-autosuspend-disable-on-shutdown?` [`tlp-configuration` parameter]
Enable USB autosuspend before shutdown.

Defaults to '`disabled`'.

`boolean restore-device-state-on-startup?` [`tlp-configuration` parameter]
Restore radio device state (bluetooth, wifi, wwan) from previous shutdown on system startup.

Defaults to '`#f`'.

The `(gnu services pm)` module provides an interface to thermald, a CPU frequency scaling service which helps prevent overheating.

`thermald-service-type` [Scheme Variable]
This is the service type for thermald (`https://01.org/linux-thermal-daemon/`), the Linux Thermal Daemon, which is responsible for controlling the thermal state of processors and preventing overheating.

`thermald-configuration` [Data Type]
Data type representing the configuration of `thermald-service-type`.

`ignore-cpuid-check?` (default: `#f`)
Ignore cpuid check for supported CPU models.

`thermald` (default: *thermald*)
Package object of thermald.

6.2.7.17 Miscellaneous Services

Lirc Service

The `(gnu services lirc)` module provides the following service.

`lirc-service` [*#:lirc lirc*] [*#:device #f*] [*#:driver #f*] [*#:config-file #f*] [*#:extra-options '()*] [Scheme Procedure]
Return a service that runs LIRC (`http://www.lirc.org`), a daemon that decodes infrared signals from remote controls.

Optionally, *device*, *driver* and *config-file* (configuration file name) may be specified. See `lircd` manual for details.

Finally, *extra-options* is a list of additional command-line options passed to `lircd`.

Spice Service

The `(gnu services spice)` module provides the following service.

`spice-vdagent-service` [*#:spice-vdagent*] [Scheme Procedure]
Returns a service that runs VDAGENT (`http://www.spice-space.org`), a daemon that enables sharing the clipboard with a vm and setting the guest display resolution when the graphical console window resizes.

6.2.7.18 Dictionary Services

The `(gnu services dict)` module provides the following service:

`dicod-service` [*#:config* (*dicod-configuration*)] [Scheme Procedure]
Return a service that runs the `dicod` daemon, an implementation of DICT server (see Section "Dicod" in *GNU Dico Manual*).

The optional *config* argument specifies the configuration for `dicod`, which should be a `<dicod-configuration>` object, by default it serves the GNU Collaborative International Dictonary of English.

You can add `open localhost` to your `~/.dico` file to make `localhost` the default server for `dico` client (see Section "Initialization File" in *GNU Dico Manual*).

`dicod-configuration` [Data Type]

Data type representing the configuration of dicod.

`dico` (default: *dico*)
: Package object of the GNU Dico dictionary server.

`interfaces` (default: *'("localhost")*)
: This is the list of IP addresses and ports and possibly socket file names to listen to (see Section "Server Settings" in *GNU Dico Manual*).

`handlers` (default: *'()*)
: List of `<dicod-handler>` objects denoting handlers (module instances).

`databases` (default: *(list %dicod-database:gcide)*)
: List of `<dicod-database>` objects denoting dictionaries to be served.

`dicod-handler` [Data Type]

Data type representing a dictionary handler (module instance).

`name` Name of the handler (module instance).

`module` (default: *#f*)
: Name of the dicod module of the handler (instance). If it is `#f`, the module has the same name as the handler. (see Section "Modules" in *GNU Dico Manual*).

`options` List of strings or gexps representing the arguments for the module handler

`dicod-database` [Data Type]

Data type representing a dictionary database.

`name` Name of the database, will be used in DICT commands.

`handler` Name of the dicod handler (module instance) used by this database (see Section "Handlers" in *GNU Dico Manual*).

`complex?` (default: *#f*)
: Whether the database configuration complex. The complex configuration will need a corresponding `<dicod-handler>` object, otherwise not.

`options` List of strings or gexps representing the arguments for the database (see Section "Databases" in *GNU Dico Manual*).

`%dicod-database:gcide` [Scheme Variable]

A `<dicod-database>` object serving the GNU Collaborative International Dictionary of English using the `gcide` package.

The following is an example `dicod-service` configuration.

```
(dicod-service #:config
  (dicod-configuration
   (handlers (list (dicod-handler
                    (name "wordnet")
                    (module "dictorg")
                    (options
```

```
                    (list #~(string-append "dbdir=" #$wordnet))))))
      (databases (list (dicod-database
                    (name "wordnet")
                    (complex? #t)
                    (handler "wordnet")
                    (options '("database=wn")))
                   %dicod-database:gcide))))
```

6.2.7.19 Version Control

The `(gnu services version-control)` module provides the following services:

Git daemon service

`git-daemon-service` [*#:config* (*git-daemon-configuration*)] [Scheme Procedure]
Return a service that runs `git daemon`, a simple TCP server to expose repositories over the Git protocol for anonymous access.

The optional *config* argument should be a `<git-daemon-configuration>` object, by default it allows read-only access to exported[9] repositories under `/srv/git`.

`git-daemon-configuration` [Data Type]
Data type representing the configuration for `git-daemon-service`.

`package` (default: *git*)
Package object of the Git distributed version control system.

`export-all?` (default: *#f*)
Whether to allow access for all Git repositories, even if they do not have the `git-daemon-export-ok` file.

`base-path` (default: `/srv/git`)
Whether to remap all the path requests as relative to the given path. If you run git daemon with *(base-path "/srv/git")* on example.com, then if you later try to pull `git://example.com/hello.git`, git daemon will interpret the path as `/srv/git/hello.git`.

`user-path` (default: *#f*)
Whether to allow `~user` notation to be used in requests. When specified with empty string, requests to `git://host/~alice/foo` is taken as a request to access `foo` repository in the home directory of user `alice`. If *(user-path "path")* is specified, the same request is taken as a request to access `path/foo` repository in the home directory of user `alice`.

`listen` (default: *'()*)
Whether to listen on specific IP addresses or hostnames, defaults to all.

`port` (default: *#f*)
Whether to listen on an alternative port, which defaults to 9418.

`whitelist` (default: *'()*)
If not empty, only allow access to this list of directories.

[9] By creating the magic file "git-daemon-export-ok" in the repository directory.

`extra-options` (default: *'()*)
Extra options will be passed to `git daemon`, please run `man git-daemon` for more information.

6.2.8 Setuid Programs

Some programs need to run with "root" privileges, even when they are launched by unprivileged users. A notorious example is the `passwd` program, which users can run to change their password, and which needs to access the `/etc/passwd` and `/etc/shadow` files—something normally restricted to root, for obvious security reasons. To address that, these executables are *setuid-root*, meaning that they always run with root privileges (see Section "How Change Persona" in *The GNU C Library Reference Manual*, for more info about the setuid mechanism.)

The store itself *cannot* contain setuid programs: that would be a security issue since any user on the system can write derivations that populate the store (see Section 4.3 [The Store], page 48). Thus, a different mechanism is used: instead of changing the setuid bit directly on files that are in the store, we let the system administrator *declare* which programs should be setuid root.

The `setuid-programs` field of an `operating-system` declaration contains a list of G-expressions denoting the names of programs to be setuid-root (see Section 6.2.1 [Using the Configuration System], page 103). For instance, the `passwd` program, which is part of the Shadow package, can be designated by this G-expression (see Section 4.6 [G-Expressions], page 56):

```
#~(string-append #$shadow "/bin/passwd")
```

A default set of setuid programs is defined by the `%setuid-programs` variable of the `(gnu system)` module.

`%setuid-programs` [Scheme Variable]
A list of G-expressions denoting common programs that are setuid-root.

The list includes commands such as `passwd`, `ping`, `su`, and `sudo`.

Under the hood, the actual setuid programs are created in the `/run/setuid-programs` directory at system activation time. The files in this directory refer to the "real" binaries, which are in the store.

6.2.9 X.509 Certificates

Web servers available over HTTPS (that is, HTTP over the transport-layer security mechanism, TLS) send client programs an *X.509 certificate* that the client can then use to *authenticate* the server. To do that, clients verify that the server's certificate is signed by a so-called *certificate authority* (CA). But to verify the CA's signature, clients must have first acquired the CA's certificate.

Web browsers such as GNU IceCat include their own set of CA certificates, such that they are able to verify CA signatures out-of-the-box.

However, most other programs that can talk HTTPS—`wget`, `git`, `w3m`, etc.—need to be told where CA certificates can be found.

In GuixSD, this is done by adding a package that provides certificates to the `packages` field of the `operating-system` declaration (see Section 6.2.2 [operating-system Reference],

page 109). GuixSD includes one such package, `nss-certs`, which is a set of CA certificates provided as part of Mozilla's Network Security Services.

Note that it is *not* part of *%base-packages*, so you need to explicitly add it. The `/etc/ssl/certs` directory, which is where most applications and libraries look for certificates by default, points to the certificates installed globally.

Unprivileged users, including users of Guix on a foreign distro, can also install their own certificate package in their profile. A number of environment variables need to be defined so that applications and libraries know where to find them. Namely, the OpenSSL library honors the `SSL_CERT_DIR` and `SSL_CERT_FILE` variables. Some applications add their own environment variables; for instance, the Git version control system honors the certificate bundle pointed to by the `GIT_SSL_CAINFO` environment variable. Thus, you would typically run something like:

```
$ guix package -i nss-certs
$ export SSL_CERT_DIR="$HOME/.guix-profile/etc/ssl/certs"
$ export SSL_CERT_FILE="$HOME/.guix-profile/etc/ssl/certs/ca-certificates.crt"
$ export GIT_SSL_CAINFO="$SSL_CERT_FILE"
```

As another example, R requires the `CURL_CA_BUNDLE` environment variable to point to a certificate bundle, so you would have to run something like this:

```
$ guix package -i nss-certs
$ export CURL_CA_BUNDLE="$HOME/.guix-profile/etc/ssl/certs/ca-certificates.crt"
```

For other applications you may want to look up the required environment variable in the relevant documentation.

6.2.10 Name Service Switch

The `(gnu system nss)` module provides bindings to the configuration file of the libc *name service switch* or *NSS* (see Section "NSS Configuration File" in *The GNU C Library Reference Manual*). In a nutshell, the NSS is a mechanism that allows libc to be extended with new "name" lookup methods for system databases, which includes host names, service names, user accounts, and more (see Section "Name Service Switch" in *The GNU C Library Reference Manual*).

The NSS configuration specifies, for each system database, which lookup method is to be used, and how the various methods are chained together—for instance, under which circumstances NSS should try the next method in the list. The NSS configuration is given in the `name-service-switch` field of `operating-system` declarations (see Section 6.2.2 [operating-system Reference], page 109).

As an example, the declaration below configures the NSS to use the `nss-mdns` back-end (`http://0pointer.de/lennart/projects/nss-mdns/`), which supports host name lookups over multicast DNS (mDNS) for host names ending in `.local`:

```
(name-service-switch
   (hosts (list %files    ;first, check /etc/hosts

                ;; If the above did not succeed, try
                ;; with 'mdns_minimal'.
                (name-service
                  (name "mdns_minimal")
```

```
    ;; 'mdns_minimal' is authoritative for
    ;; '.local'.  When it returns "not found",
    ;; no need to try the next methods.
    (reaction (lookup-specification
               (not-found => return))))

  ;; Then fall back to DNS.
  (name-service
    (name "dns"))

  ;; Finally, try with the "full" 'mdns'.
  (name-service
    (name "mdns")))))
```

Do not worry: the `%mdns-host-lookup-nss` variable (see below) contains this configuration, so you will not have to type it if all you want is to have `.local` host lookup working.

Note that, in this case, in addition to setting the `name-service-switch` of the `operating-system` declaration, you also need to use `avahi-service` (see Section 6.2.7.4 [Networking Services], page 132), or *%desktop-services*, which includes it (see Section 6.2.7.7 [Desktop Services], page 154). Doing this makes `nss-mdns` accessible to the name service cache daemon (see Section 6.2.7.1 [Base Services], page 120).

For convenience, the following variables provide typical NSS configurations.

`%default-nss` [Scheme Variable]

This is the default name service switch configuration, a `name-service-switch` object.

`%mdns-host-lookup-nss` [Scheme Variable]

This is the name service switch configuration with support for host name lookup over multicast DNS (mDNS) for host names ending in `.local`.

The reference for name service switch configuration is given below. It is a direct mapping of the configuration file format of the C library , so please refer to the C library manual for more information (see Section "NSS Configuration File" in *The GNU C Library Reference Manual*). Compared to the configuration file format of libc NSS, it has the advantage not only of adding this warm parenthetic feel that we like, but also static checks: you will know about syntax errors and typos as soon as you run `guix system`.

`name-service-switch` [Data Type]

This is the data type representation the configuration of libc's name service switch (NSS). Each field below represents one of the supported system databases.

`aliases`
`ethers`
`group`
`gshadow`
`hosts`
`initgroups`
`netgroup`
`networks`
`password`
`public-key`
`rpc`
`services`
`shadow` The system databases handled by the NSS. Each of these fields must be a list of `<name-service>` objects (see below).

`name-service` [Data Type]
This is the data type representing an actual name service and the associated lookup action.

`name` A string denoting the name service (see Section "Services in the NSS configuration" in *The GNU C Library Reference Manual*).

Note that name services listed here must be visible to nscd. This is achieved by passing the `#:name-services` argument to `nscd-service` the list of packages providing the needed name services (see Section 6.2.7.1 [Base Services], page 120).

`reaction` An action specified using the `lookup-specification` macro (see Section "Actions in the NSS configuration" in *The GNU C Library Reference Manual*). For example:

```
(lookup-specification (unavailable => continue)
                      (success => return))
```

6.2.11 Initial RAM Disk

For bootstrapping purposes, the Linux-Libre kernel is passed an *initial RAM disk*, or *initrd*. An initrd contains a temporary root file system as well as an initialization script. The latter is responsible for mounting the real root file system, and for loading any kernel modules that may be needed to achieve that.

The `initrd` field of an `operating-system` declaration allows you to specify which initrd you would like to use. The `(gnu system linux-initrd)` module provides three ways to build an initrd: the high-level `base-initrd` procedure and the low-level `raw-initrd` and `expression->initrd` procedures.

The `base-initrd` procedure is intended to cover most common uses. For example, if you want to add a bunch of kernel modules to be loaded at boot time, you can define the `initrd` field of the operating system declaration like this:

```
(initrd (lambda (file-systems . rest)
          ;; Create a standard initrd that has modules "foo.ko"
          ;; and "bar.ko", as well as their dependencies, in
```

```
;; addition to the modules available by default.
(apply base-initrd file-systems
       #:extra-modules '("foo" "bar")
       rest)))
```

The `base-initrd` procedure also handles common use cases that involves using the system as a QEMU guest, or as a "live" system with volatile root file system.

The `base-initrd` procedure is built from `raw-initrd` procedure. Unlike `base-initrd`, `raw-initrd` doesn't do anything high-level, such as trying to guess which kernel modules and packages should be included to the initrd. An example use of `raw-initrd` is when a user has a custom Linux kernel configuration and default kernel modules included by `base-initrd` are not available.

The initial RAM disk produced by `base-initrd` or `raw-initrd` honors several options passed on the Linux kernel command line (that is, arguments passed *via* the `linux` command of GRUB, or the `-append` option of QEMU), notably:

`--load=`*boot*
: Tell the initial RAM disk to load *boot*, a file containing a Scheme program, once it has mounted the root file system.

  GuixSD uses this option to yield control to a boot program that runs the service activation programs and then spawns the GNU Shepherd, the initialization system.

`--root=`*root*
: Mount *root* as the root file system. *root* can be a device name like `/dev/sda1`, a partition label, or a partition UUID.

`--system=`*system*
: Have `/run/booted-system` and `/run/current-system` point to *system*.

`modprobe.blacklist=`*modules...*
: Instruct the initial RAM disk as well as the `modprobe` command (from the kmod package) to refuse to load *modules*. *modules* must be a comma-separated list of module names—e.g., `usbkbd,9pnet`.

`--repl`
: Start a read-eval-print loop (REPL) from the initial RAM disk before it tries to load kernel modules and to mount the root file system. Our marketing team calls it *boot-to-Guile*. The Schemer in you will love it. See Section "Using Guile Interactively" in *GNU Guile Reference Manual*, for more information on Guile's REPL.

Now that you know all the features that initial RAM disks produced by `base-initrd` and `raw-initrd` provide, here is how to use it and customize it further.

`raw-initrd` *file-systems* [*#:linux-modules '()*] [Monadic Procedure]
[*#:mapped-devices '()*] [*#:helper-packages '()*] [*#:qemu-networking? #f*]
[*#:volatile-root? #f*]

Return a monadic derivation that builds a raw initrd. *file-systems* is a list of file systems to be mounted by the initrd, possibly in addition to the root file system specified on the kernel command line via `--root`. *linux-modules* is a list of kernel modules to be

loaded at boot time. *mapped-devices* is a list of device mappings to realize before *file-systems* are mounted (see Section 6.2.4 [Mapped Devices], page 114). *helper-packages* is a list of packages to be copied in the initrd. It may include `e2fsck/static` or other packages needed by the initrd to check root partition.

When *qemu-networking?* is true, set up networking with the standard QEMU parameters. When *virtio?* is true, load additional modules so that the initrd can be used as a QEMU guest with para-virtualized I/O drivers.

When *volatile-root?* is true, the root file system is writable but any changes to it are lost.

`base-initrd` *file-systems* [*#:mapped-devices '()*] [Monadic Procedure]
[*#:qemu-networking? #f*] [*#:volatile-root? #f*] [*#:virtio? #t*]
[*#:extra-modules '()*]

Return a monadic derivation that builds a generic initrd. *file-systems* is a list of file systems to be mounted by the initrd like for `raw-initrd`. *mapped-devices*, *qemu-networking?* and *volatile-root?* also behaves as in `raw-initrd`.

When *virtio?* is true, load additional modules so that the initrd can be used as a QEMU guest with para-virtualized I/O drivers.

The initrd is automatically populated with all the kernel modules necessary for *file-systems* and for the given options. However, additional kernel modules can be listed in *extra-modules*. They will be added to the initrd, and loaded at boot time in the order in which they appear.

Needless to say, the initrds we produce and use embed a statically-linked Guile, and the initialization program is a Guile program. That gives a lot of flexibility. The `expression->initrd` procedure builds such an initrd, given the program to run in that initrd.

`expression->initrd` *exp* [*#:guile %guile-static-stripped*] [Monadic Procedure]
[*#:name "guile-initrd"*]

Return a derivation that builds a Linux initrd (a gzipped cpio archive) containing *guile* and that evaluates *exp*, a G-expression, upon booting. All the derivations referenced by *exp* are automatically copied to the initrd.

6.2.12 GRUB Configuration

The operating system uses GNU GRUB as its boot loader (see Section "Overview" in *GNU GRUB Manual*). It is configured using a `grub-configuration` declaration. This data type is exported by the `(gnu system grub)` module and described below.

`grub-configuration` [Data Type]

The type of a GRUB configuration declaration.

`device` This is a string denoting the boot device. It must be a device name understood by the `grub-install` command, such as `/dev/sda` or `(hd0)` (see Section "Invoking grub-install" in *GNU GRUB Manual*).

`menu-entries` (default: `()`)
: A possibly empty list of `menu-entry` objects (see below), denoting entries to appear in the GRUB boot menu, in addition to the current system entry and the entry pointing to previous system generations.

`default-entry` (default: `0`)
: The index of the default boot menu entry. Index 0 is for the entry of the current system.

`timeout` (default: `5`)
: The number of seconds to wait for keyboard input before booting. Set to 0 to boot immediately, and to -1 to wait indefinitely.

`theme` (default: *%default-theme*)
: The `grub-theme` object describing the theme to use.

`grub` (default: `grub`)
: The GRUB package to use.

`terminal-outputs` (default: `'gfxterm`)
: The output terminals used for the GRUB boot menu, as a list of symbols. These values are accepted: `console`, `serial`, `serial_{0-3}`, `gfxterm`, `vga_text`, `mda_text`, `morse`, and `pkmodem`. This field corresponds to the GRUB variable GRUB_TERMINAL_OUTPUT (see Section "Simple configuration" in *GNU GRUB manual*).

`terminal-inputs` (default: `'()`)
: The input terminals used for the GRUB boot menu, as a list of symbols. The default is the native platform terminal as determined by GRUB at run-time. These values are accepted: `console`, `serial`, `serial_{0-3}`, `at_keyboard`, and `usb_keyboard`. This field corresponds to the GRUB variable GRUB_TERMINAL_INPUT (see Section "Simple configuration" in *GNU GRUB manual*).

`serial-unit` (default: `#f`)
: The serial unit used by GRUB, as an integer from 0 to 3. The default value is chosen by GRUB at run-time; currently GRUB chooses 0, which corresponds to COM1 (see Section "Serial terminal" in *GNU GRUB manual*).

`serial-speed` (default: `#f`)
: The speed of the serial interface, as an integer. The default value is chosen by GRUB at run-time; currently GRUB chooses 9600 bps (see Section "Serial terminal" in *GNU GRUB manual*).

Should you want to list additional boot menu entries *via* the `menu-entries` field above, you will need to create them with the `menu-entry` form. For example, imagine you want to be able to boot another distro (hard to imagine!), you can define a menu entry along these lines:

```
(menu-entry
  (label "The Other Distro")
```

```
  (linux "/boot/old/vmlinux-2.6.32")
  (linux-arguments '("root=/dev/sda2"))
  (initrd "/boot/old/initrd"))
```

Details below.

`menu-entry` [Data Type]

The type of an entry in the GRUB boot menu.

`label` The label to show in the menu—e.g., `"GNU"`.

`linux` The Linux kernel image to boot, for example:

```
(file-append linux-libre "/bzImage")
```

It is also possible to specify a device explicitly in the file path using GRUB's device naming convention (see Section "Naming convention" in *GNU GRUB manual*), for example:

```
"(hd0,msdos1)/boot/vmlinuz"
```

If the device is specified explicitly as above, then the `device` field is ignored entirely.

`linux-arguments` (default: `()`)
The list of extra Linux kernel command-line arguments—e.g., `("console=ttyS0")`.

`initrd` A G-Expression or string denoting the file name of the initial RAM disk to use (see Section 4.6 [G-Expressions], page 56).

`device` (default: `#f`)
The device where the kernel and initrd are to be found—i.e., the GRUB *root* for this menu entry (see Section "root" in *GNU GRUB manual*).

This may be a file system label (a string), a file system UUID (a bytevector, see Section 6.2.3 [File Systems], page 112), or `#f`, in which case GRUB will search the device containing the file specified by the `linux` field (see Section "search" in *GNU GRUB manual*). It must *not* be an OS device name such as `/dev/sda1`.

`device-mount-point` (default: `"/"`)
The mount point of the above device on the system. You probably do not need to change the default value. GuixSD uses it to strip the prefix of store file names for systems where `/gnu` or `/gnu/store` is on a separate partition.

Themes are created using the `grub-theme` form, which is not documented yet.

`%default-theme` [Scheme Variable]

This is the default GRUB theme used by the operating system, with a fancy background image displaying the GNU and Guix logos.

6.2.13 Invoking `guix system`

Once you have written an operating system declaration as seen in the previous section, it can be *instantiated* using the `guix system` command. The synopsis is:

```
guix system options... action file
```

file must be the name of a file containing an `operating-system` declaration. *action* specifies how the operating system is instantiated. Currently the following values are supported:

`reconfigure`

Build the operating system described in *file*, activate it, and switch to it[10].

This effects all the configuration specified in *file*: user accounts, system services, global package list, setuid programs, etc. The command starts system services specified in *file* that are not currently running; if a service is currently running, it does not attempt to upgrade it since this would not be possible without stopping it first.

This command creates a new generation whose number is one greater than the current generation (as reported by `guix system list-generations`). If that generation already exists, it will be overwritten. This behavior mirrors that of `guix package` (see Section 3.2 [Invoking guix package], page 18).

It also adds a GRUB menu entry for the new OS configuration, and moves entries for older configurations to a submenu—unless `--no-bootloader` is passed.

> **Note:** It is highly recommended to run `guix pull` once before you run `guix system reconfigure` for the first time (see Section 3.6 [Invoking guix pull], page 29). Failing to do that you would see an older version of Guix once `reconfigure` has completed.

`switch-generation`

Switch to an existing system generation. This action atomically switches the system profile to the specified system generation. It also rearranges the system's existing GRUB menu entries. It makes the menu entry for the specified system generation the default, and it moves the entries for the other generations to a submenu. The next time the system boots, it will use the specified system generation.

The target generation can be specified explicitly by its generation number. For example, the following invocation would switch to system generation 7:

```
guix system switch-generation 7
```

The target generation can also be specified relative to the current generation with the form `+N` or `-N`, where `+3` means "3 generations ahead of the current generation," and `-1` means "1 generation prior to the current generation." When specifying a negative value such as `-1`, you must precede it with `--` to prevent it from being parsed as an option. For example:

```
guix system switch-generation -- -1
```

[10] This action (and the related actions `switch-generation` and `roll-back`) are usable only on systems already running GuixSD.

Currently, the effect of invoking this action is *only* to switch the system profile to an existing generation and rearrange the GRUB menu entries. To actually start using the target system generation, you must reboot after running this action. In the future, it will be updated to do the same things as `reconfigure`, like activating and deactivating services.

This action will fail if the specified generation does not exist.

`roll-back`

Switch to the preceding system generation. The next time the system boots, it will use the preceding system generation. This is the inverse of `reconfigure`, and it is exactly the same as invoking `switch-generation` with an argument of `-1`.

Currently, as with `switch-generation`, you must reboot after running this action to actually start using the preceding system generation.

`build` Build the derivation of the operating system, which includes all the configuration files and programs needed to boot and run the system. This action does not actually install anything.

`init` Populate the given directory with all the files necessary to run the operating system specified in *file*. This is useful for first-time installations of GuixSD. For instance:

```
guix system init my-os-config.scm /mnt
```

copies to `/mnt` all the store items required by the configuration specified in `my-os-config.scm`. This includes configuration files, packages, and so on. It also creates other essential files needed for the system to operate correctly—e.g., the `/etc`, `/var`, and `/run` directories, and the `/bin/sh` file.

This command also installs GRUB on the device specified in `my-os-config`, unless the `--no-bootloader` option was passed.

`vm` Build a virtual machine that contains the operating system declared in *file*, and return a script to run that virtual machine (VM). Arguments given to the script are passed to QEMU.

The VM shares its store with the host system.

Additional file systems can be shared between the host and the VM using the `--share` and `--expose` command-line options: the former specifies a directory to be shared with write access, while the latter provides read-only access to the shared directory.

The example below creates a VM in which the user's home directory is accessible read-only, and where the `/exchange` directory is a read-write mapping of `$HOME/tmp` on the host:

```
guix system vm my-config.scm \
   --expose=$HOME --share=$HOME/tmp=/exchange
```

On GNU/Linux, the default is to boot directly to the kernel; this has the advantage of requiring only a very tiny root disk image since the store of the host can then be mounted.

The `--full-boot` option forces a complete boot sequence, starting with the bootloader. This requires more disk space since a root image containing at least the kernel, initrd, and bootloader data files must be created. The `--image-size` option can be used to specify the size of the image.

`vm-image`
`disk-image`
: Return a virtual machine or disk image of the operating system declared in *file* that stands alone. Use the `--image-size` option to specify the size of the image.

  When using `vm-image`, the returned image is in qcow2 format, which the QEMU emulator can efficiently use. See Section 6.2.14 [Running GuixSD in a VM], page 218, for more information on how to run the image in a virtual machine.

  When using `disk-image`, a raw disk image is produced; it can be copied as is to a USB stick, for instance. Assuming `/dev/sdc` is the device corresponding to a USB stick, one can copy the image to it using the following command:

  ```
  # dd if=$(guix system disk-image my-os.scm) of=/dev/sdc
  ```

`container`
: Return a script to run the operating system declared in *file* within a container. Containers are a set of lightweight isolation mechanisms provided by the kernel Linux-libre. Containers are substantially less resource-demanding than full virtual machines since the kernel, shared objects, and other resources can be shared with the host system; this also means they provide thinner isolation.

  Currently, the script must be run as root in order to support more than a single user and group. The container shares its store with the host system.

  As with the `vm` action (see [guix system vm], page 215), additional file systems to be shared between the host and container can be specified using the `--share` and `--expose` options:

  ```
  guix system container my-config.scm \
     --expose=$HOME --share=$HOME/tmp=/exchange
  ```

  Note: This option requires Linux-libre 3.19 or newer.

options can contain any of the common build options (see Section 5.1.1 [Common Build Options], page 64). In addition, *options* can contain one of the following:

`--system=`*system*
`-s` *system*
: Attempt to build for *system* instead of the host system type. This works as per `guix build` (see Section 5.1 [Invoking guix build], page 64).

`--derivation`
`-d`
: Return the derivation file name of the given operating system without building anything.

`--image-size=`*size*
: For the `vm-image` and `disk-image` actions, create an image of the given *size*. *size* may be a number of bytes, or it may include a unit as a suffix (see Section "Block size" in *GNU Coreutils*).

`--root=`*file*
`-r` *file* Make *file* a symlink to the result, and register it as a garbage collector root.

`--on-error=`*strategy*
Apply *strategy* when an error occurs when reading *file*. *strategy* may be one of the following:

`nothing-special`
Report the error concisely and exit. This is the default strategy.

`backtrace`
Likewise, but also display a backtrace.

`debug` Report the error and enter Guile's debugger. From there, you can run commands such as `,bt` to get a backtrace, `,locals` to display local variable values, and more generally inspect the state of the program. See Section "Debug Commands" in *GNU Guile Reference Manual*, for a list of available debugging commands.

Note: All the actions above, except `build` and `init`, can use KVM support in the Linux-libre kernel. Specifically, if the machine has hardware virtualization support, the corresponding KVM kernel module should be loaded, and the `/dev/kvm` device node must exist and be readable and writable by the user and by the build users of the daemon (see Section 2.4.1 [Build Environment Setup], page 7).

Once you have built, configured, re-configured, and re-re-configured your GuixSD installation, you may find it useful to list the operating system generations available on disk—and that you can choose from the GRUB boot menu:

`list-generations`
List a summary of each generation of the operating system available on disk, in a human-readable way. This is similar to the `--list-generations` option of `guix package` (see Section 3.2 [Invoking guix package], page 18).

Optionally, one can specify a pattern, with the same syntax that is used in `guix package --list-generations`, to restrict the list of generations displayed. For instance, the following command displays generations that are up to 10 days old:

```
$ guix system list-generations 10d
```

The `guix system` command has even more to offer! The following sub-commands allow you to visualize how your system services relate to each other:

`extension-graph`
Emit in Dot/Graphviz format to standard output the *service extension graph* of the operating system defined in *file* (see Section 6.2.15.1 [Service Composition], page 219, for more information on service extensions.)

The command:

```
$ guix system extension-graph file | dot -Tpdf > services.pdf
```

produces a PDF file showing the extension relations among services.

`shepherd-graph`
: Emit in Dot/Graphviz format to standard output the *dependency graph* of shepherd services of the operating system defined in *file*. See Section 6.2.15.4 [Shepherd Services], page 226, for more information and for an example graph.

6.2.14 Running GuixSD in a Virtual Machine

To run GuixSD in a virtual machine (VM), one can either use the pre-built GuixSD VM image distributed at '`ftp://alpha.gnu.org/guix/guixsd-vm-image-0.13.0.`*system*`.tar.xz`' , or build their own virtual machine image using `guix system vm-image` (see Section 6.2.13 [Invoking guix system], page 214). The returned image is in qcow2 format, which the QEMU emulator (`http://qemu.org/`) can efficiently use.

If you built your own image, you must copy it out of the store (see Section 4.3 [The Store], page 48) and give yourself permission to write to the copy before you can use it. When invoking QEMU, you must choose a system emulator that is suitable for your hardware platform. Here is a minimal QEMU invocation that will boot the result of `guix system vm-image` on x86_64 hardware:

```
$ qemu-system-x86_64 \
   -net user -net nic,model=virtio \
   -enable-kvm -m 256 /tmp/qemu-image
```

Here is what each of these options means:

`qemu-system-x86_64`
: This specifies the hardware platform to emulate. This should match the host.

`-net user` Enable the unprivileged user-mode network stack. The guest OS can access the host but not vice versa. This is the simplest way to get the guest OS online.

`-net nic,model=virtio`
: You must create a network interface of a given model. If you do not create a NIC, the boot will fail. Assuming your hardware platform is x86_64, you can get a list of available NIC models by running `qemu-system-x86_64 -net nic,model=help`.

`-enable-kvm`
: If your system has hardware virtualization extensions, enabling the virtual machine support (KVM) of the Linux kernel will make things run faster.

`-m 256` RAM available to the guest OS, in mebibytes. Defaults to 128 MiB, which may be insufficient for some operations.

`/tmp/qemu-image`
: The file name of the qcow2 image.

The default `run-vm.sh` script that is returned by an invocation of `guix system vm` does not add a `-net user` flag by default. To get network access from within the vm add the `(dhcp-client-service)` to your system definition and start the VM using '`guix system vm config.scm` `-net user`'. An important caveat of using `-net user` for networking is that `ping` will not work, because it uses the ICMP protocol. You'll have to use a different command to check for network connectivity, for example `guix download`.

6.2.14.1 Connecting Through SSH

To enable SSH inside a VM you need to add a SSH server like `(dropbear-service)` or `(lsh-service)` to your VM. The `(lsh-service)` doesn't currently boot unsupervised. It requires you to type some characters to initialize the randomness generator. In addition you need to forward the SSH port, 22 by default, to the host. You can do this with

```
‘guix system vm config.scm‘ -net user,hostfwd=tcp::10022-:22
```

To connect to the VM you can run

```
ssh -o UserKnownHostsFile=/dev/null -o StrictHostKeyChecking=no -p 10022
```

The `-p` tells `ssh` the port you want to connect to. `-o UserKnownHostsFile=/dev/null` prevents `ssh` from complaining every time you modify your `config.scm` file and the `-o StrictHostKeyChecking=no` prevents you from having to allow a connection to an unknown host every time you connect.

6.2.14.2 Using `virt-viewer` with Spice

As an alternative to the default `qemu` graphical client you can use the `remote-viewer` from the `virt-viewer` package. To connect pass the `-spice port=5930,disable-ticketing` flag to `qemu`. See previous section for further information on how to do this.

Spice also allows you to do some nice stuff like share your clipboard with your VM. To enable that you'll also have to pass the following flags to `qemu`:

```
-device virtio-serial-pci,id=virtio-serial0,max_ports=16,bus=pci.0,addr=0x5
-chardev spicevmc,name=vdagent,id=vdagent
-device virtserialport,nr=1,bus=virtio-serial0.0,chardev=vdagent,
name=com.redhat.spice.0
```

You'll also need to add the see Section 6.2.7.17 [Miscellaneous Services], page 203.

6.2.15 Defining Services

The previous sections show the available services and how one can combine them in an `operating-system` declaration. But how do we define them in the first place? And what is a service anyway?

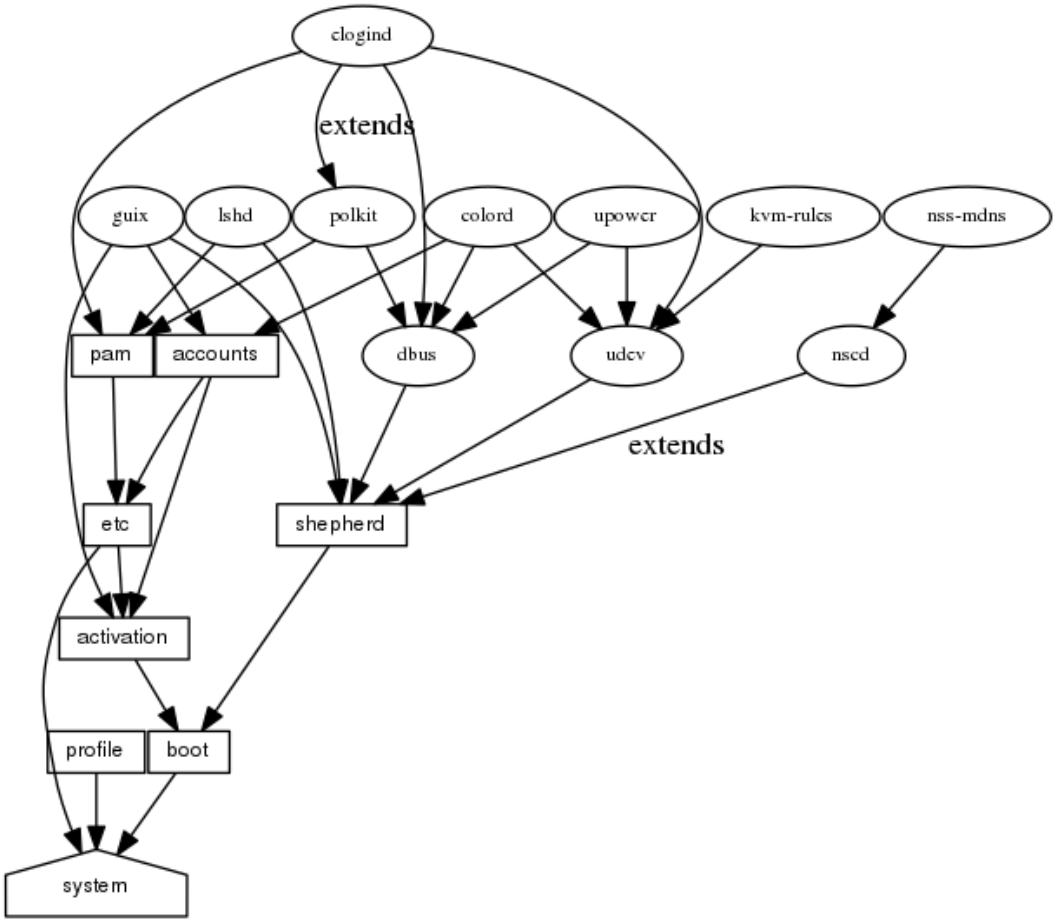
6.2.15.1 Service Composition

Here we define a *service* as, broadly, something that extends the functionality of the operating system. Often a service is a process—a *daemon*—started when the system boots: a secure shell server, a Web server, the Guix build daemon, etc. Sometimes a service is a daemon whose execution can be triggered by another daemon—e.g., an FTP server started by `inetd` or a D-Bus service activated by `dbus-daemon`. Occasionally, a service does not map to a daemon. For instance, the "account" service collects user accounts and makes sure they exist when the system runs; the "udev" service collects device management rules and makes them available to the eudev daemon; the `/etc` service populates the `/etc` directory of the system.

GuixSD services are connected by *extensions*. For instance, the secure shell service *extends* the Shepherd—the GuixSD initialization system, running as PID 1—by giving it the command lines to start and stop the secure shell daemon (see Section 6.2.7.4 [Networking Services], page 132); the UPower service extends the D-Bus service by passing it

its `.service` specification, and extends the udev service by passing it device management rules (see Section 6.2.7.7 [Desktop Services], page 154); the Guix daemon service extends the Shepherd by passing it the command lines to start and stop the daemon, and extends the account service by passing it a list of required build user accounts (see Section 6.2.7.1 [Base Services], page 120).

All in all, services and their "extends" relations form a directed acyclic graph (DAG). If we represent services as boxes and extensions as arrows, a typical system might provide something like this:



At the bottom, we see the *system service*, which produces the directory containing everything to run and boot the system, as returned by the `guix system build` command. See Section 6.2.15.3 [Service Reference], page 222, to learn about the other service types shown here. See [system-extension-graph], page 217, for information on how to generate this representation for a particular operating system definition.

Technically, developers can define *service types* to express these relations. There can be any number of services of a given type on the system—for instance, a system running two instances of the GNU secure shell server (lsh) has two instances of *lsh-service-type*, with different parameters.

The following section describes the programming interface for service types and services.

6.2.15.2 Service Types and Services

A *service type* is a node in the DAG described above. Let us start with a simple example, the service type for the Guix build daemon (see Section 2.5 [Invoking guix-daemon], page 11):

```
(define guix-service-type
  (service-type
   (name 'guix)
   (extensions
    (list (service-extension shepherd-root-service-type guix-shepherd-service)
          (service-extension account-service-type guix-accounts)
          (service-extension activation-service-type guix-activation)))
   (default-value (guix-configuration))))
```

It defines three things:

1. A name, whose sole purpose is to make inspection and debugging easier.
2. A list of *service extensions*, where each extension designates the target service type and a procedure that, given the parameters of the service, returns a list of objects to extend the service of that type.

   Every service type has at least one service extension. The only exception is the *boot service type*, which is the ultimate service.
3. Optionally, a default value for instances of this type.

In this example, *guix-service-type* extends three services:

shepherd-root-service-type
: The *guix-shepherd-service* procedure defines how the Shepherd service is extended. Namely, it returns a `<shepherd-service>` object that defines how `guix-daemon` is started and stopped (see Section 6.2.15.4 [Shepherd Services], page 226).

account-service-type
: This extension for this service is computed by *guix-accounts*, which returns a list of `user-group` and `user-account` objects representing the build user accounts (see Section 2.5 [Invoking guix-daemon], page 11).

activation-service-type
: Here *guix-activation* is a procedure that returns a gexp, which is a code snippet to run at "activation time"—e.g., when the service is booted.

A service of this type is instantiated like this:

```
(service guix-service-type
         (guix-configuration
           (build-accounts 5)
           (use-substitutes? #f)))
```

The second argument to the `service` form is a value representing the parameters of this specific service instance. See [guix-configuration-type], page 126, for information about the

`guix-configuration` data type. When the value is omitted, the default value specified by `guix-service-type` is used:

```
(service guix-service-type)
```

guix-service-type is quite simple because it extends other services but is not extensible itself.

The service type for an *extensible* service looks like this:

```
(define udev-service-type
  (service-type (name 'udev)
                (extensions
                 (list (service-extension shepherd-root-service-type
                                          udev-shepherd-service)))

                (compose concatenate)       ;concatenate the list of rules
                (extend (lambda (config rules)
                          (match config
                            (($ <udev-configuration> udev initial-rules)
                             (udev-configuration
                              (udev udev)   ;the udev package to use
                              (rules (append initial-rules rules)))))))))
```

This is the service type for the eudev device management daemon (`https://wiki.gentoo.org/wiki/Project:Eudev`). Compared to the previous example, in addition to an extension of *shepherd-root-service-type*, we see two new fields:

`compose` This is the procedure to *compose* the list of extensions to services of this type.

Services can extend the udev service by passing it lists of rules; we compose those extensions simply by concatenating them.

`extend` This procedure defines how the value of the service is *extended* with the composition of the extensions.

Udev extensions are composed into a list of rules, but the udev service value is itself a `<udev-configuration>` record. So here, we extend that record by appending the list of rules it contains to the list of contributed rules.

There can be only one instance of an extensible service type such as *udev-service-type*. If there were more, the `service-extension` specifications would be ambiguous.

Still here? The next section provides a reference of the programming interface for services.

6.2.15.3 Service Reference

We have seen an overview of service types (see Section 6.2.15.2 [Service Types and Services], page 221). This section provides a reference on how to manipulate services and service types. This interface is provided by the `(gnu services)` module.

`service` *type* [*value*] [Scheme Procedure]

Return a new service of *type*, a `<service-type>` object (see below.) *value* can be any object; it represents the parameters of this particular service instance.

When *value* is omitted, the default value specified by *type* is used; if *type* does not specify a default value, an error is raised.

For instance, this:

```
(service openssh-service-type)
```

is equivalent to this:

```
(service openssh-service-type
         (openssh-configuration))
```

In both cases the result is an instance of `openssh-service-type` with the default configuration.

`service? obj` [Scheme Procedure]
Return true if *obj* is a service.

`service-kind service` [Scheme Procedure]
Return the type of *service*—i.e., a `<service-type>` object.

`service-value service` [Scheme Procedure]
Return the value associated with *service*. It represents its parameters.

Here is an example of how a service is created and manipulated:

```
(define s
  (service nginx-service-type
           (nginx-configuration
            (nginx nginx)
            (log-directory log-directory)
            (run-directory run-directory)
            (file config-file))))

(service? s)
⇒ #t

(eq? (service-kind s) nginx-service-type)
⇒ #t
```

The `modify-services` form provides a handy way to change the parameters of some of the services of a list such as *%base-services* (see Section 6.2.7.1 [Base Services], page 120). It evaluates to a list of services. Of course, you could always use standard list combinators such as `map` and `fold` to do that (see Section "SRFI-1" in *GNU Guile Reference Manual*); `modify-services` simply provides a more concise form for this common pattern.

`modify-services services (type variable => body) ...` [Scheme Syntax]
Modify the services listed in *services* according to the given clauses. Each clause has the form:

```
(type variable => body)
```

where *type* is a service type—e.g., `guix-service-type`—and *variable* is an identifier that is bound within the *body* to the service parameters—e.g., a `guix-configuration` instance—of the original service of that *type*.

The *body* should evaluate to the new service parameters, which will be used to configure the new service. This new service will replace the original in the resulting list. Because a service's service parameters are created using `define-record-type*`, you can write a succinct *body* that evaluates to the new service parameters by using the `inherit` feature that `define-record-type*` provides.

See Section 6.2.1 [Using the Configuration System], page 103, for example usage.

Next comes the programming interface for service types. This is something you want to know when writing new service definitions, but not necessarily when simply looking for ways to customize your `operating-system` declaration.

`service-type` [Data Type]

This is the representation of a *service type* (see Section 6.2.15.2 [Service Types and Services], page 221).

`name` This is a symbol, used only to simplify inspection and debugging.

`extensions`
A non-empty list of `<service-extension>` objects (see below).

`compose` (default: `#f`)
If this is `#f`, then the service type denotes services that cannot be extended—i.e., services that do not receive "values" from other services.

Otherwise, it must be a one-argument procedure. The procedure is called by `fold-services` and is passed a list of values collected from extensions. It must return a value that is a valid parameter value for the service instance.

`extend` (default: `#f`)
If this is `#f`, services of this type cannot be extended.

Otherwise, it must be a two-argument procedure: `fold-services` calls it, passing it the initial value of the service as the first argument and the result of applying `compose` to the extension values as the second argument.

See Section 6.2.15.2 [Service Types and Services], page 221, for examples.

`service-extension` *target-type compute* [Scheme Procedure]

Return a new extension for services of type *target-type*. *compute* must be a one-argument procedure: `fold-services` calls it, passing it the value associated with the service that provides the extension; it must return a valid value for the target service.

`service-extension?` *obj* [Scheme Procedure]

Return true if *obj* is a service extension.

Occasionally, you might want to simply extend an existing service. This involves creating a new service type and specifying the extension of interest, which can be verbose; the `simple-service` procedure provides a shorthand for this.

`simple-service` *name target value* [Scheme Procedure]

Return a service that extends *target* with *value*. This works by creating a singleton service type *name*, of which the returned service is an instance.

For example, this extends mcron (see Section 6.2.7.2 [Scheduled Job Execution], page 129) with an additional job:

```
(simple-service 'my-mcron-job mcron-service-type
                #~(job '(next-hour (3)) "guix gc -F 2G"))
```

At the core of the service abstraction lies the `fold-services` procedure, which is responsible for "compiling" a list of services down to a single directory that contains everything needed to boot and run the system—the directory shown by the `guix system build` command (see Section 6.2.13 [Invoking guix system], page 214). In essence, it propagates service extensions down the service graph, updating each node parameters on the way, until it reaches the root node.

`fold-services` *services* [*#:target-type system-service-type*] [Scheme Procedure]

Fold *services* by propagating their extensions down to the root of type *target-type*; return the root service adjusted accordingly.

Lastly, the `(gnu services)` module also defines several essential service types, some of which are listed below.

`system-service-type` [Scheme Variable]

This is the root of the service graph. It produces the system directory as returned by the `guix system build` command.

`boot-service-type` [Scheme Variable]

The type of the "boot service", which produces the *boot script*. The boot script is what the initial RAM disk runs when booting.

`etc-service-type` [Scheme Variable]

The type of the `/etc` service. This service can be extended by passing it name/file tuples such as:

```
(list `("issue" ,(plain-file "issue" "Welcome!\n")))
```

In this example, the effect would be to add an `/etc/issue` file pointing to the given file.

`setuid-program-service-type` [Scheme Variable]

Type for the "setuid-program service". This service collects lists of executable file names, passed as gexps, and adds them to the set of setuid-root programs on the system (see Section 6.2.8 [Setuid Programs], page 206).

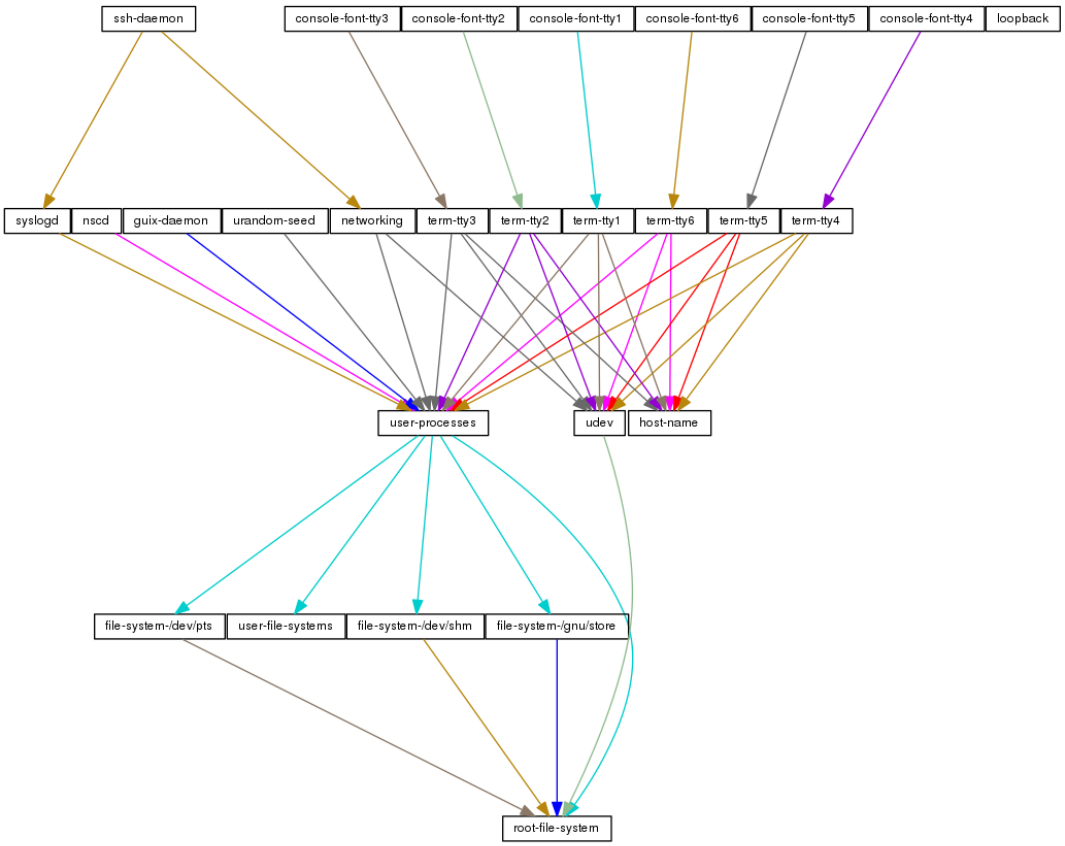
`profile-service-type` [Scheme Variable]

Type of the service that populates the *system profile*—i.e., the programs under `/run/current-system/profile`. Other services can extend it by passing it lists of packages to add to the system profile.

6.2.15.4 Shepherd Services

The `(gnu services shepherd)` module provides a way to define services managed by the GNU Shepherd, which is the GuixSD initialization system—the first process that is started when the system boots, also known as PID 1 (see Section "Introduction" in *The GNU Shepherd Manual*).

Services in the Shepherd can depend on each other. For instance, the SSH daemon may need to be started after the syslog daemon has been started, which in turn can only happen once all the file systems have been mounted. The simple operating system defined earlier (see Section 6.2.1 [Using the Configuration System], page 103) results in a service graph like this:



You can actually generate such a graph for any operating system definition using the `guix system shepherd-graph` command (see [system-shepherd-graph], page 217).

The *%shepherd-root-service* is a service object representing PID 1, of type *shepherd-root-service-type*; it can be extended by passing it lists of `<shepherd-service>` objects.

`shepherd-service` [Data Type]

The data type representing a service managed by the Shepherd.

`provision`
: This is a list of symbols denoting what the service provides.

  These are the names that may be passed to `herd start`, `herd status`, and similar commands (see Section "Invoking herd" in *The GNU Shepherd Manual*). See Section "Slots of services" in *The GNU Shepherd Manual*, for details.

`requirements` (default: `'()`)
: List of symbols denoting the Shepherd services this one depends on.

`respawn?` (default: `#t`)
: Whether to restart the service when it stops, for instance when the underlying process dies.

`start`
`stop` (default: `#~(const #f)`)
: The `start` and `stop` fields refer to the Shepherd's facilities to start and stop processes (see Section "Service De- and Constructors" in *The GNU Shepherd Manual*). They are given as G-expressions that get expanded in the Shepherd configuration file (see Section 4.6 [G-Expressions], page 56).

`documentation`
: A documentation string, as shown when running:

  ```
  herd doc service-name
  ```

  where *service-name* is one of the symbols in *provision* (see Section "Invoking herd" in *The GNU Shepherd Manual*).

`modules` (default: *%default-modules*)
: This is the list of modules that must be in scope when `start` and `stop` are evaluated.

`shepherd-root-service-type` [Scheme Variable]
: The service type for the Shepherd "root service"—i.e., PID 1.

  This is the service type that extensions target when they want to create shepherd services (see Section 6.2.15.2 [Service Types and Services], page 221, for an example). Each extension must pass a list of `<shepherd-service>`.

`%shepherd-root-service` [Scheme Variable]
: This service represents PID 1.

6.3 Documentation

In most cases packages installed with Guix come with documentation. There are two main documentation formats: "Info", a browseable hypertext format used for GNU software, and "manual pages" (or "man pages"), the linear documentation format traditionally found on Unix. Info manuals are accessed with the `info` command or with Emacs, and man pages are accessed using `man`.

You can look for documentation of software installed on your system by keyword. For example, the following command searches for information about "TLS" in Info manuals:

```
$ info -k TLS
```

```
"(emacs)Network Security" -- STARTTLS
"(emacs)Network Security" -- TLS
"(gnutls)Core TLS API" -- gnutls_certificate_set_verify_flags
"(gnutls)Core TLS API" -- gnutls_certificate_set_verify_function
...
```

The command below searches for the same keyword in man pages:

```
$ man -k TLS
SSL (7)              - OpenSSL SSL/TLS library
certtool (1)         - GnuTLS certificate tool
...
```

These searches are purely local to your computer so you have the guarantee that documentation you find corresponds to what you have actually installed, you can access it off-line, and your privacy is respected.

Once you have these results, you can view the relevant documentation by running, say:

```
$ info "(gnutls)Core TLS API"
```

or:

```
$ man certtool
```

Info manuals contain sections and indices as well as hyperlinks like those found in Web pages. The `info` reader (see *Stand-alone GNU Info*) and its Emacs counterpart (see Section "Misc Help" in *The GNU Emacs Manual*) provide intuitive key bindings to navigate manuals. See Section "Getting Started" in *Info: An Introduction*, for an introduction to Info navigation.

6.4 Installing Debugging Files

Program binaries, as produced by the GCC compilers for instance, are typically written in the ELF format, with a section containing *debugging information*. Debugging information is what allows the debugger, GDB, to map binary code to source code; it is required to debug a compiled program in good conditions.

The problem with debugging information is that is takes up a fair amount of disk space. For example, debugging information for the GNU C Library weighs in at more than 60 MiB. Thus, as a user, keeping all the debugging info of all the installed programs is usually not an option. Yet, space savings should not come at the cost of an impediment to debugging—especially in the GNU system, which should make it easier for users to exert their computing freedom (see Chapter 6 [GNU Distribution], page 96).

Thankfully, the GNU Binary Utilities (Binutils) and GDB provide a mechanism that allows users to get the best of both worlds: debugging information can be stripped from the binaries and stored in separate files. GDB is then able to load debugging information from those files, when they are available (see Section "Separate Debug Files" in *Debugging with GDB*).

The GNU distribution takes advantage of this by storing debugging information in the `lib/debug` sub-directory of a separate package output unimaginatively called `debug` (see Section 3.4 [Packages with Multiple Outputs], page 27). Users can choose to install the

`debug` output of a package when they need it. For instance, the following command installs the debugging information for the GNU C Library and for GNU Guile:

```
guix package -i glibc:debug guile:debug
```

GDB must then be told to look for debug files in the user's profile, by setting the `debug-file-directory` variable (consider setting it from the `~/.gdbinit` file, see Section "Startup" in *Debugging with GDB*):

```
(gdb) set debug-file-directory ~/.guix-profile/lib/debug
```

From there on, GDB will pick up debugging information from the `.debug` files under `~/.guix-profile/lib/debug`.

In addition, you will most likely want GDB to be able to show the source code being debugged. To do that, you will have to unpack the source code of the package of interest (obtained with `guix build --source`, see Section 5.1 [Invoking guix build], page 64), and to point GDB to that source directory using the `directory` command (see Section "Source Path" in *Debugging with GDB*).

The `debug` output mechanism in Guix is implemented by the `gnu-build-system` (see Section 4.2 [Build Systems], page 41). Currently, it is opt-in—debugging information is available only for the packages with definitions explicitly declaring a `debug` output. This may be changed to opt-out in the future if our build farm servers can handle the load. To check whether a package has a `debug` output, use `guix package --list-available` (see Section 3.2 [Invoking guix package], page 18).

6.5 Security Updates

Occasionally, important security vulnerabilities are discovered in software packages and must be patched. Guix developers try hard to keep track of known vulnerabilities and to apply fixes as soon as possible in the `master` branch of Guix (we do not yet provide a "stable" branch containing only security updates.) The `guix lint` tool helps developers find out about vulnerable versions of software packages in the distribution:

```
$ guix lint -c cve
gnu/packages/base.scm:652:2: glibc@2.21: probably vulnerable to CVE-2015-1781, CVE-2015-7547
gnu/packages/gcc.scm:334:2: gcc@4.9.3: probably vulnerable to CVE-2015-5276
gnu/packages/image.scm:312:2: openjpeg@2.1.0: probably vulnerable to CVE-2016-1923, CVE-2016-1924
...
```

See Section 5.7 [Invoking guix lint], page 80, for more information.

Note: As of version 0.13.0, the feature described below is considered "beta".

Guix follows a functional package management discipline (see Chapter 1 [Introduction], page 2), which implies that, when a package is changed, *every package that depends on it* must be rebuilt. This can significantly slow down the deployment of fixes in core packages such as libc or Bash, since basically the whole distribution would need to be rebuilt. Using pre-built binaries helps (see Section 3.3 [Substitutes], page 25), but deployment may still take more time than desired.

To address this, Guix implements *grafts*, a mechanism that allows for fast deployment of critical updates without the costs associated with a whole-distribution rebuild. The idea is to rebuild only the package that needs to be patched, and then to "graft" it onto packages explicitly installed by the user and that were previously referring to the original package.

The cost of grafting is typically very low, and order of magnitudes lower than a full rebuild of the dependency chain.

For instance, suppose a security update needs to be applied to Bash. Guix developers will provide a package definition for the "fixed" Bash, say *bash-fixed*, in the usual way (see Section 4.1 [Defining Packages], page 35). Then, the original package definition is augmented with a `replacement` field pointing to the package containing the bug fix:

```
(define bash
  (package
    (name "bash")
    ;; ...
    (replacement bash-fixed)))
```

From there on, any package depending directly or indirectly on Bash—as reported by `guix gc --requisites` (see Section 3.5 [Invoking guix gc], page 27)—that is installed is automatically "rewritten" to refer to *bash-fixed* instead of *bash*. This grafting process takes time proportional to the size of the package, usually less than a minute for an "average" package on a recent machine. Grafting is recursive: when an indirect dependency requires grafting, then grafting "propagates" up to the package that the user is installing.

Currently, the length of the name and version of the graft and that of the package it replaces (*bash-fixed* and *bash* in the example above) must be equal. This restriction mostly comes from the fact that grafting works by patching files, including binary files, directly. Other restrictions may apply: for instance, when adding a graft to a package providing a shared library, the original shared library and its replacement must have the same `SONAME` and be binary-compatible.

The `--no-grafts` command-line option allows you to forcefully avoid grafting (see Section 5.1.1 [Common Build Options], page 64). Thus, the command:

```
guix build bash --no-grafts
```

returns the store file name of the original Bash, whereas:

```
guix build bash
```

returns the store file name of the "fixed", replacement Bash. This allows you to distinguish between the two variants of Bash.

To verify which Bash your whole profile refers to, you can run (see Section 3.5 [Invoking guix gc], page 27):

```
guix gc -R `readlink -f ~/.guix-profile` | grep bash
```

... and compare the store file names that you get with those above. Likewise for a complete GuixSD system generation:

```
guix gc -R `guix system build my-config.scm` | grep bash
```

Lastly, to check which Bash running processes are using, you can use the `lsof` command:

```
lsof | grep /gnu/store/.*bash
```

6.6 Package Modules

From a programming viewpoint, the package definitions of the GNU distribution are provided by Guile modules in the `(gnu packages ...)` name space[11] (see Section "Modules" in *GNU Guile Reference Manual*). For instance, the `(gnu packages emacs)` module exports a variable named `emacs`, which is bound to a `<package>` object (see Section 4.1 [Defining Packages], page 35).

The `(gnu packages ...)` module name space is automatically scanned for packages by the command-line tools. For instance, when running `guix package -i emacs`, all the `(gnu packages ...)` modules are scanned until one that exports a package object whose name is `emacs` is found. This package search facility is implemented in the `(gnu packages)` module.

Users can store package definitions in modules with different names—e.g., `(my-packages emacs)`[12]. These package definitions will not be visible by default. Users can invoke commands such as `guix package` and `guix build` with the `-e` option so that they know where to find the package. Better yet, they can use the `-L` option of these commands to make those modules visible (see Section 5.1 [Invoking guix build], page 64), or define the `GUIX_PACKAGE_PATH` environment variable. This environment variable makes it easy to extend or customize the distribution and is honored by all the user interfaces.

`GUIX_PACKAGE_PATH` [Environment Variable]

This is a colon-separated list of directories to search for additional package modules. Directories listed in this variable take precedence over the own modules of the distribution.

The distribution is fully *bootstrapped* and *self-contained*: each package is built based solely on other packages in the distribution. The root of this dependency graph is a small set of *bootstrap binaries*, provided by the `(gnu packages bootstrap)` module. For more information on bootstrapping, see Section 6.8 [Bootstrapping], page 237.

6.7 Packaging Guidelines

The GNU distribution is nascent and may well lack some of your favorite packages. This section describes how you can help make the distribution grow. See Chapter 7 [Contributing], page 242, for additional information on how you can help.

Free software packages are usually distributed in the form of *source code tarballs*—typically `tar.gz` files that contain all the source files. Adding a package to the distribution means essentially two things: adding a *recipe* that describes how to build the package, including a list of other packages required to build it, and adding *package metadata* along with that recipe, such as a description and licensing information.

In Guix all this information is embodied in *package definitions*. Package definitions provide a high-level view of the package. They are written using the syntax of the Scheme

[11] Note that packages under the `(gnu packages ...)` module name space are not necessarily "GNU packages". This module naming scheme follows the usual Guile module naming convention: `gnu` means that these modules are distributed as part of the GNU system, and `packages` identifies modules that define packages.

[12] Note that the file name and module name must match. For instance, the `(my-packages emacs)` module must be stored in a `my-packages/emacs.scm` file relative to the load path specified with `--load-path` or `GUIX_PACKAGE_PATH`. See Section "Modules and the File System" in *GNU Guile Reference Manual*, for details.

programming language; in fact, for each package we define a variable bound to the package definition, and export that variable from a module (see Section 6.6 [Package Modules], page 231). However, in-depth Scheme knowledge is *not* a prerequisite for creating packages. For more information on package definitions, see Section 4.1 [Defining Packages], page 35.

Once a package definition is in place, stored in a file in the Guix source tree, it can be tested using the `guix build` command (see Section 5.1 [Invoking guix build], page 64). For example, assuming the new package is called `gnew`, you may run this command from the Guix build tree (see Section 7.2 [Running Guix Before It Is Installed], page 243):

```
./pre-inst-env guix build gnew --keep-failed
```

Using `--keep-failed` makes it easier to debug build failures since it provides access to the failed build tree. Another useful command-line option when debugging is `--log-file`, to access the build log.

If the package is unknown to the `guix` command, it may be that the source file contains a syntax error, or lacks a `define-public` clause to export the package variable. To figure it out, you may load the module from Guile to get more information about the actual error:

```
./pre-inst-env guile -c '(use-modules (gnu packages gnew))'
```

Once your package builds correctly, please send us a patch (see Chapter 7 [Contributing], page 242). Well, if you need help, we will be happy to help you too. Once the patch is committed in the Guix repository, the new package automatically gets built on the supported platforms by our continuous integration system (`http://hydra.gnu.org/jobset/gnu/master`).

Users can obtain the new package definition simply by running `guix pull` (see Section 3.6 [Invoking guix pull], page 29). When `hydra.gnu.org` is done building the package, installing the package automatically downloads binaries from there (see Section 3.3 [Substitutes], page 25). The only place where human intervention is needed is to review and apply the patch.

6.7.1 Software Freedom

The GNU operating system has been developed so that users can have freedom in their computing. GNU is *free software*, meaning that users have the four essential freedoms (`http://www.gnu.org/philosophy/free-sw.html`): to run the program, to study and change the program in source code form, to redistribute exact copies, and to distribute modified versions. Packages found in the GNU distribution provide only software that conveys these four freedoms.

In addition, the GNU distribution follow the free software distribution guidelines (`http://www.gnu.org/distros/free-system-distribution-guidelines.html`). Among other things, these guidelines reject non-free firmware, recommendations of non-free software, and discuss ways to deal with trademarks and patents.

Some otherwise free upstream package sources contain a small and optional subset that violates the above guidelines, for instance because this subset is itself non-free code. When that happens, the offending items are removed with appropriate patches or code snippets in the `origin` form of the package (see Section 4.1 [Defining Packages], page 35). This way, `guix build --source` returns the "freed" source rather than the unmodified upstream source.

6.7.2 Package Naming

A package has actually two names associated with it: First, there is the name of the *Scheme variable*, the one following `define-public`. By this name, the package can be made known in the Scheme code, for instance as input to another package. Second, there is the string in the `name` field of a package definition. This name is used by package management commands such as `guix package` and `guix build`.

Both are usually the same and correspond to the lowercase conversion of the project name chosen upstream, with underscores replaced with hyphens. For instance, GNUnet is available as `gnunet`, and SDL_net as `sdl-net`.

We do not add `lib` prefixes for library packages, unless these are already part of the official project name. But see Section 6.7.5 [Python Modules], page 235, and Section 6.7.6 [Perl Modules], page 236, for special rules concerning modules for the Python and Perl languages.

Font package names are handled differently, see Section 6.7.8 [Fonts], page 237.

6.7.3 Version Numbers

We usually package only the latest version of a given free software project. But sometimes, for instance for incompatible library versions, two (or more) versions of the same package are needed. These require different Scheme variable names. We use the name as defined in Section 6.7.2 [Package Naming], page 233, for the most recent version; previous versions use the same name, suffixed by `-` and the smallest prefix of the version number that may distinguish the two versions.

The name inside the package definition is the same for all versions of a package and does not contain any version number.

For instance, the versions 2.24.20 and 3.9.12 of GTK+ may be packaged as follows:

```
(define-public gtk+
  (package
    (name "gtk+")
    (version "3.9.12")
    ...))
(define-public gtk+-2
  (package
    (name "gtk+")
    (version "2.24.20")
    ...))
```

If we also wanted GTK+ 3.8.2, this would be packaged as

```
(define-public gtk+-3.8
  (package
    (name "gtk+")
    (version "3.8.2")
    ...))
```

Occasionally, we package snapshots of upstream's version control system (VCS) instead of formal releases. This should remain exceptional, because it is up to upstream developers to clarify what the stable release is. Yet, it is sometimes necessary. So, what should we put in the `version` field?

Clearly, we need to make the commit identifier of the VCS snapshot visible in the version string, but we also need to make sure that the version string is monotonically increasing so that `guix package --upgrade` can determine which version is newer. Since commit identifiers, notably with Git, are not monotonically increasing, we add a revision number that we increase each time we upgrade to a newer snapshot. The resulting version string looks like this:

```
2.0.11-3.cabba9e
  ^    ^    ^
  |    |    `-- upstream commit ID
  |    |
  |    `--- Guix package revision
  |
latest upstream version
```

It is a good idea to strip commit identifiers in the `version` field to, say, 7 digits. It avoids an aesthetic annoyance (assuming aesthetics have a role to play here) as well as problems related to OS limits such as the maximum shebang length (127 bytes for the Linux kernel.) It is best to use the full commit identifiers in `origin`s, though, to avoid ambiguities. A typical package definition may look like this:

```
(define my-package
  (let ((commit "c3f29bc928d5900971f65965feaae59e1272a3f7")
        (revision "1"))          ;Guix package revision
    (package
      (version (string-append "0.9-" revision "."
                              (string-take commit 7)))
      (source (origin
                (method git-fetch)
                (uri (git-reference
                      (url "git://example.org/my-package.git")
                      (commit commit)))
                (sha256 (base32 "1mbikn..."))
                (file-name (string-append "my-package-" version
                                          "-checkout"))))
      ;; ...
      )))
```

6.7.4 Synopses and Descriptions

As we have seen before, each package in GNU Guix includes a synopsis and a description (see Section 4.1 [Defining Packages], page 35). Synopses and descriptions are important: They are what `guix package --search` searches, and a crucial piece of information to help users determine whether a given package suits their needs. Consequently, packagers should pay attention to what goes into them.

Synopses must start with a capital letter and must not end with a period. They must not start with “a” or “the”, which usually does not bring anything; for instance, prefer “File-frobbing tool” over “A tool that frobs files”. The synopsis should say what the package is—e.g., “Core GNU utilities (file, text, shell)”—or what it is used for—e.g., the synopsis for GNU grep is “Print lines matching a pattern”.

Keep in mind that the synopsis must be meaningful for a very wide audience. For example, "Manipulate alignments in the SAM format" might make sense for a seasoned bioinformatics researcher, but might be fairly unhelpful or even misleading to a non-specialized audience. It is a good idea to come up with a synopsis that gives an idea of the application domain of the package. In this example, this might give something like "Manipulate nucleotide sequence alignments", which hopefully gives the user a better idea of whether this is what they are looking for.

Descriptions should take between five and ten lines. Use full sentences, and avoid using acronyms without first introducing them. Please avoid marketing phrases such as "world-leading", "industrial-strength", and "next-generation", and avoid superlatives like "the most advanced"—they are not helpful to users looking for a package and may even sound suspicious. Instead, try to be factual, mentioning use cases and features.

Descriptions can include Texinfo markup, which is useful to introduce ornaments such as `@code` or `@dfn`, bullet lists, or hyperlinks (see Section "Overview" in *GNU Texinfo*). However you should be careful when using some characters for example '`@`' and curly braces which are the basic special characters in Texinfo (see Section "Special Characters" in *GNU Texinfo*). User interfaces such as `guix package --show` take care of rendering it appropriately.

Synopses and descriptions are translated by volunteers at the Translation Project (`http://translationproject.org/domain/guix-packages.html`) so that as many users as possible can read them in their native language. User interfaces search them and display them in the language specified by the current locale.

Translation is a lot of work so, as a packager, please pay even more attention to your synopses and descriptions as every change may entail additional work for translators. In order to help them, it is possible to make recommendations or instructions visible to them by inserting special comments like this (see Section "xgettext Invocation" in *GNU Gettext*):

```
;; TRANSLATORS: "X11 resize-and-rotate" should not be translated.
(description "ARandR is designed to provide a simple visual front end
for the X11 resize-and-rotate (RandR) extension. ...")
```

6.7.5 Python Modules

We currently package Python 2 and Python 3, under the Scheme variable names `python-2` and `python` as explained in Section 6.7.3 [Version Numbers], page 233. To avoid confusion and naming clashes with other programming languages, it seems desirable that the name of a package for a Python module contains the word `python`.

Some modules are compatible with only one version of Python, others with both. If the package Foo compiles only with Python 3, we name it `python-foo`; if it compiles only with Python 2, we name it `python2-foo`. If it is compatible with both versions, we create two packages with the corresponding names.

If a project already contains the word `python`, we drop this; for instance, the module python-dateutil is packaged under the names `python-dateutil` and `python2-dateutil`. If the project name starts with `py` (e.g. `pytz`), we keep it and prefix it as described above.

6.7.5.1 Specifying Dependencies

Dependency information for Python packages is usually available in the package source tree, with varying degrees of accuracy: in the `setup.py` file, in `requirements.txt`, or in `tox.ini`.

Your mission, when writing a recipe for a Python package, is to map these dependencies to the appropriate type of "input" (see Section 4.1.1 [package Reference], page 38). Although the `pypi` importer normally does a good job (see Section 5.5 [Invoking guix import], page 73), you may want to check the following check list to determine which dependency goes where.

- We currently package Python 2 with `setuptools` and `pip` installed like Python 3.4 has per default. Thus you don't need to specify either of these as an input. `guix lint` will warn you if you do.
- Python dependencies required at run time go into `propagated-inputs`. They are typically defined with the `install_requires` keyword in `setup.py`, or in the `requirements.txt` file.
- Python packages required only at build time—e.g., those listed with the `setup_requires` keyword in `setup.py`—or only for testing—e.g., those in `tests_require`—go into `native-inputs`. The rationale is that (1) they do not need to be propagated because they are not needed at run time, and (2) in a cross-compilation context, it's the "native" input that we'd want.

  Examples are the `pytest`, `mock`, and `nose` test frameworks. Of course if any of these packages is also required at run-time, it needs to go to `propagated-inputs`.
- Anything that does not fall in the previous categories goes to `inputs`, for example programs or C libraries required for building Python packages containing C extensions.
- If a Python package has optional dependencies (`extras_require`), it is up to you to decide whether to add them or not, based on their usefulness/overhead ratio (see Section 7.5 [Submitting Patches], page 245).

6.7.6 Perl Modules

Perl programs standing for themselves are named as any other package, using the lowercase upstream name. For Perl packages containing a single class, we use the lowercase class name, replace all occurrences of `::` by dashes and prepend the prefix `perl-`. So the class `XML::Parser` becomes `perl-xml-parser`. Modules containing several classes keep their lowercase upstream name and are also prepended by `perl-`. Such modules tend to have the word `perl` somewhere in their name, which gets dropped in favor of the prefix. For instance, `libwww-perl` becomes `perl-libwww`.

6.7.7 Java Packages

Java programs standing for themselves are named as any other package, using the lowercase upstream name.

To avoid confusion and naming clashes with other programming languages, it is desirable that the name of a package for a Java package is prefixed with `java-`. If a project already contains the word `java`, we drop this; for instance, the package `ngsjava` is packaged under the name `java-ngs`.

For Java packages containing a single class or a small class hierarchy, we use the lowercase class name, replace all occurrences of `.` by dashes and prepend the prefix `java-`. So the class `apache.commons.cli` becomes package `java-apache-commons-cli`.

6.7.8 Fonts

For fonts that are in general not installed by a user for typesetting purposes, or that are distributed as part of a larger software package, we rely on the general packaging rules for software; for instance, this applies to the fonts delivered as part of the X.Org system or fonts that are part of TeX Live.

To make it easier for a user to search for fonts, names for other packages containing only fonts are constructed as follows, independently of the upstream package name.

The name of a package containing only one font family starts with `font-`; it is followed by the foundry name and a dash `-` if the foundry is known, and the font family name, in which spaces are replaced by dashes (and as usual, all upper case letters are transformed to lower case). For example, the Gentium font family by SIL is packaged under the name `font-sil-gentium`.

For a package containing several font families, the name of the collection is used in the place of the font family name. For instance, the Liberation fonts consist of three families, Liberation Sans, Liberation Serif and Liberation Mono. These could be packaged separately under the names `font-liberation-sans` and so on; but as they are distributed together under a common name, we prefer to package them together as `font-liberation`.

In the case where several formats of the same font family or font collection are packaged separately, a short form of the format, prepended by a dash, is added to the package name. We use `-ttf` for TrueType fonts, `-otf` for OpenType fonts and `-type1` for PostScript Type 1 fonts.

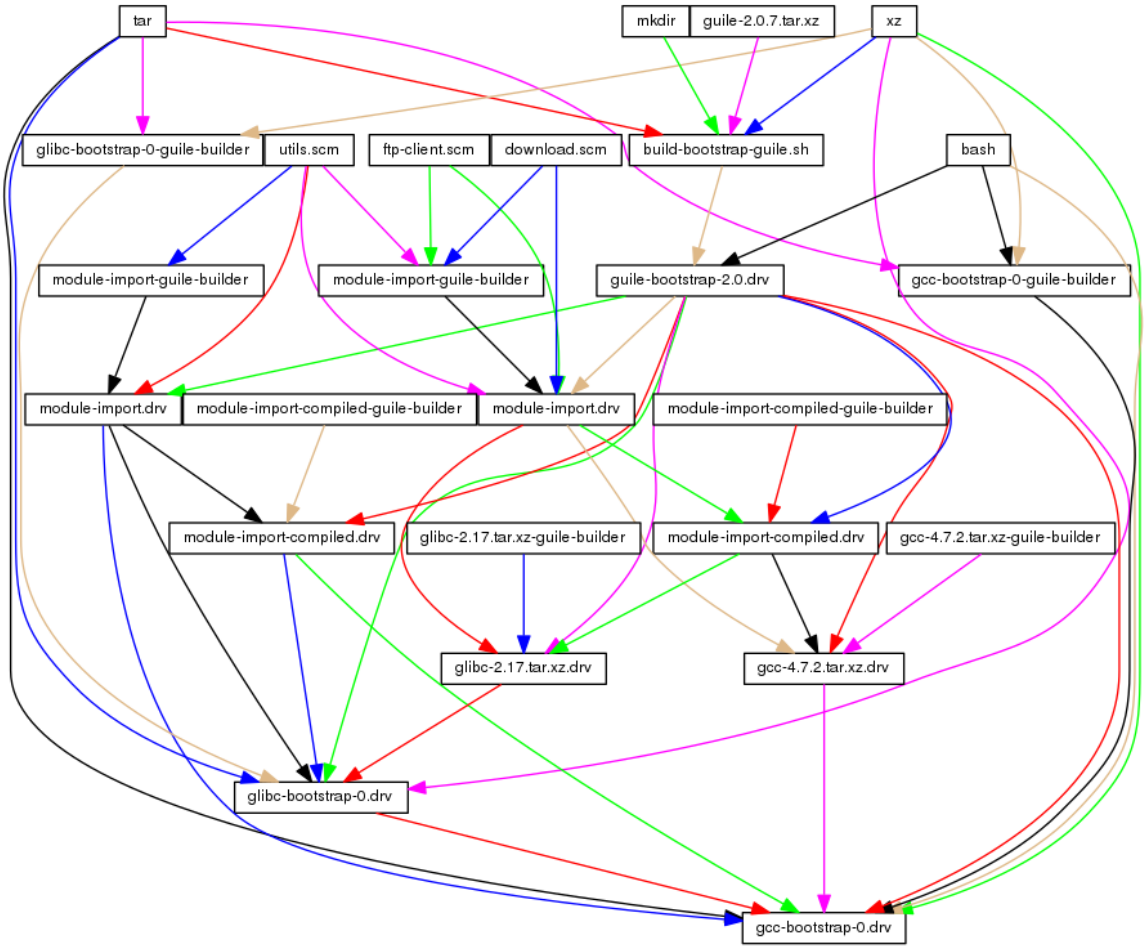
6.8 Bootstrapping

Bootstrapping in our context refers to how the distribution gets built “from nothing”. Remember that the build environment of a derivation contains nothing but its declared inputs (see Chapter 1 [Introduction], page 2). So there’s an obvious chicken-and-egg problem: how does the first package get built? How does the first compiler get compiled? Note that this is a question of interest only to the curious hacker, not to the regular user, so you can shamelessly skip this section if you consider yourself a “regular user”.

The GNU system is primarily made of C code, with libc at its core. The GNU build system itself assumes the availability of a Bourne shell and command-line tools provided by GNU Coreutils, Awk, Findutils, ‘sed’, and ‘grep’. Furthermore, build programs—programs that run `./configure`, `make`, etc.—are written in Guile Scheme (see Section 4.4 [Derivations], page 50). Consequently, to be able to build anything at all, from scratch, Guix relies on pre-built binaries of Guile, GCC, Binutils, libc, and the other packages mentioned above—the *bootstrap binaries*.

These bootstrap binaries are “taken for granted”, though we can also re-create them if needed (more on that later).

Preparing to Use the Bootstrap Binaries



The figure above shows the very beginning of the dependency graph of the distribution, corresponding to the package definitions of the `(gnu packages bootstrap)` module. A similar figure can be generated with `guix graph` (see Section 5.9 [Invoking guix graph], page 82), along the lines of:

```
guix graph -t derivation \
  -e '(@@ (gnu packages bootstrap) %bootstrap-gcc)' \
  | dot -Tps > t.ps
```

At this level of detail, things are slightly complex. First, Guile itself consists of an ELF executable, along with many source and compiled Scheme files that are dynamically loaded when it runs. This gets stored in the `guile-2.0.7.tar.xz` tarball shown in this graph. This tarball is part of Guix's "source" distribution, and gets inserted into the store with `add-to-store` (see Section 4.3 [The Store], page 48).

But how do we write a derivation that unpacks this tarball and adds it to the store? To solve this problem, the `guile-bootstrap-2.0.drv` derivation—the first one that gets

built—uses `bash` as its builder, which runs `build-bootstrap-guile.sh`, which in turn calls `tar` to unpack the tarball. Thus, `bash`, `tar`, `xz`, and `mkdir` are statically-linked binaries, also part of the Guix source distribution, whose sole purpose is to allow the Guile tarball to be unpacked.

Once `guile-bootstrap-2.0.drv` is built, we have a functioning Guile that can be used to run subsequent build programs. Its first task is to download tarballs containing the other pre-built binaries—this is what the `.tar.xz.drv` derivations do. Guix modules such as `ftp-client.scm` are used for this purpose. The `module-import.drv` derivations import those modules in a directory in the store, using the original layout. The `module-import-compiled.drv` derivations compile those modules, and write them in an output directory with the right layout. This corresponds to the `#:modules` argument of `build-expression->derivation` (see Section 4.4 [Derivations], page 50).

Finally, the various tarballs are unpacked by the derivations `gcc-bootstrap-0.drv`, `glibc-bootstrap-0.drv`, etc., at which point we have a working C tool chain.

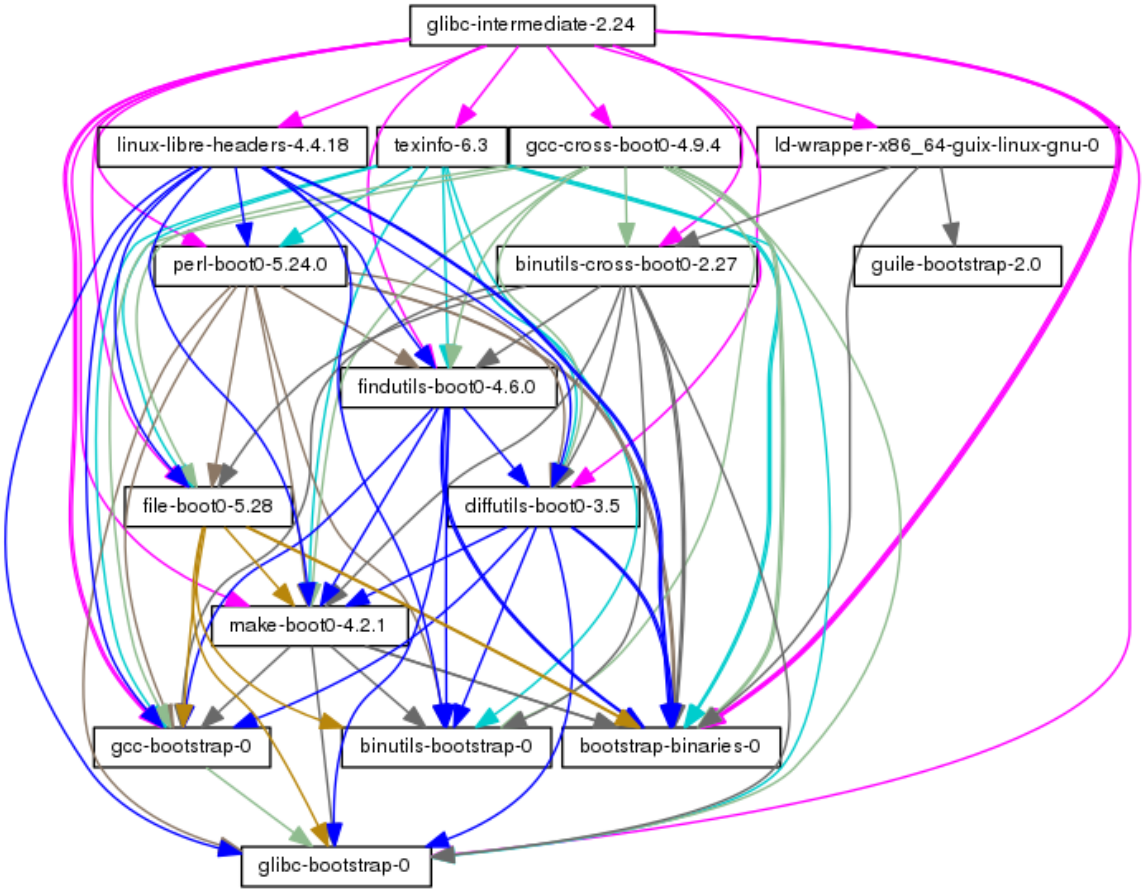
Building the Build Tools

Bootstrapping is complete when we have a full tool chain that does not depend on the pre-built bootstrap tools discussed above. This no-dependency requirement is verified by checking whether the files of the final tool chain contain references to the `/gnu/store` directories of the bootstrap inputs. The process that leads to this "final" tool chain is described by the package definitions found in the `(gnu packages commencement)` module.

The `guix graph` command allows us to "zoom out" compared to the graph above, by looking at the level of package objects instead of individual derivations—remember that a package may translate to several derivations, typically one derivation to download its source, one to build the Guile modules it needs, and one to actually build the package from source. The command:

```
guix graph -t bag \
  -e '(@@ (gnu packages commencement)
          glibc-final-with-bootstrap-bash)' | dot -Tps > t.ps
```

produces the dependency graph leading to the "final" C library[13], depicted below.



The first tool that gets built with the bootstrap binaries is GNU Make—noted `make-boot0` above—which is a prerequisite for all the following packages. From there Findutils and Diffutils get built.

Then come the first-stage Binutils and GCC, built as pseudo cross tools—i.e., with `--target` equal to `--host`. They are used to build libc. Thanks to this cross-build trick, this libc is guaranteed not to hold any reference to the initial tool chain.

From there the final Binutils and GCC (not shown above) are built. GCC uses `ld` from the final Binutils, and links programs against the just-built libc. This tool chain is used to build the other packages used by Guix and by the GNU Build System: Guile, Bash, Coreutils, etc.

And voilà! At this point we have the complete set of build tools that the GNU Build System expects. These are in the `%final-inputs` variable of the `(gnu packages commencement)` module, and are implicitly used by any package that uses `gnu-build-system` (see Section 4.2 [Build Systems], page 41).

[13] You may notice the `glibc-intermediate` label, suggesting that it is not *quite* final, but as a good approximation, we will consider it final.

Building the Bootstrap Binaries

Because the final tool chain does not depend on the bootstrap binaries, those rarely need to be updated. Nevertheless, it is useful to have an automated way to produce them, should an update occur, and this is what the `(gnu packages make-bootstrap)` module provides.

The following command builds the tarballs containing the bootstrap binaries (Guile, Binutils, GCC, libc, and a tarball containing a mixture of Coreutils and other basic command-line tools):

```
guix build bootstrap-tarballs
```

The generated tarballs are those that should be referred to in the `(gnu packages bootstrap)` module mentioned at the beginning of this section.

Still here? Then perhaps by now you've started to wonder: when do we reach a fixed point? That is an interesting question! The answer is unknown, but if you would like to investigate further (and have significant computational and storage resources to do so), then let us know.

6.9 Porting to a New Platform

As discussed above, the GNU distribution is self-contained, and self-containment is achieved by relying on pre-built "bootstrap binaries" (see Section 6.8 [Bootstrapping], page 237). These binaries are specific to an operating system kernel, CPU architecture, and application binary interface (ABI). Thus, to port the distribution to a platform that is not yet supported, one must build those bootstrap binaries, and update the `(gnu packages bootstrap)` module to use them on that platform.

Fortunately, Guix can *cross compile* those bootstrap binaries. When everything goes well, and assuming the GNU tool chain supports the target platform, this can be as simple as running a command like this one:

```
guix build --target=armv5tel-linux-gnueabi bootstrap-tarballs
```

For this to work, the `glibc-dynamic-linker` procedure in `(gnu packages bootstrap)` must be augmented to return the right file name for libc's dynamic linker on that platform; likewise, `system->linux-architecture` in `(gnu packages linux)` must be taught about the new platform.

Once these are built, the `(gnu packages bootstrap)` module needs to be updated to refer to these binaries on the target platform. That is, the hashes and URLs of the bootstrap tarballs for the new platform must be added alongside those of the currently supported platforms. The bootstrap Guile tarball is treated specially: it is expected to be available locally, and `gnu/local.mk` has rules do download it for the supported architectures; a rule for the new platform must be added as well.

In practice, there may be some complications. First, it may be that the extended GNU triplet that specifies an ABI (like the `eabi` suffix above) is not recognized by all the GNU tools. Typically, glibc recognizes some of these, whereas GCC uses an extra `--with-abi` configure flag (see `gcc.scm` for examples of how to handle this). Second, some of the required packages could fail to build for that platform. Lastly, the generated binaries could be broken for some reason.

7 Contributing

This project is a cooperative effort, and we need your help to make it grow! Please get in touch with us on `guix-devel@gnu.org` and `#guix` on the Freenode IRC network. We welcome ideas, bug reports, patches, and anything that may be helpful to the project. We particularly welcome help on packaging (see Section 6.7 [Packaging Guidelines], page 231).

We want to provide a warm, friendly, and harassment-free environment, so that anyone can contribute to the best of their abilities. To this end our project uses a "Contributor Covenant", which was adapted from `http://contributor-covenant.org/`. You can find a local version in the `CODE-OF-CONDUCT` file in the source tree.

Contributors are not required to use their legal name in patches and on-line communication; they can use any name or pseudonym of their choice.

7.1 Building from Git

If you want to hack Guix itself, it is recommended to use the latest version from the Git repository. When building Guix from a checkout, the following packages are required in addition to those mentioned in the installation instructions (see Section 2.2 [Requirements], page 5).

- GNU Autoconf (`http://gnu.org/software/autoconf/`);
- GNU Automake (`http://gnu.org/software/automake/`);
- GNU Gettext (`http://gnu.org/software/gettext/`);
- GNU Texinfo (`http://gnu.org/software/texinfo/`);
- Graphviz (`http://www.graphviz.org/`);
- GNU Help2man (optional) (`http://www.gnu.org/software/help2man/`).

The easiest way to set up a development environment for Guix is, of course, by using Guix! The following command starts a new shell where all the dependencies and appropriate environment variables are set up to hack on Guix:

```
guix environment guix
```

See Section 5.10 [Invoking guix environment], page 86, for more information on that command. Extra dependencies can be added with `--ad-hoc`:

```
guix environment guix --ad-hoc help2man git strace
```

Run `./bootstrap` to generate the build system infrastructure using Autoconf and Automake. If you get an error like this one:

```
configure.ac:46: error: possibly undefined macro: PKG_CHECK_MODULES
```

it probably means that Autoconf couldn't find `pkg.m4`, which is provided by pkg-config. Make sure that `pkg.m4` is available. The same holds for the `guile.m4` set of macros provided by Guile. For instance, if you installed Automake in `/usr/local`, it wouldn't look for `.m4` files in `/usr/share`. In that case, you have to invoke the following command:

```
export ACLOCAL_PATH=/usr/share/aclocal
```

See Section "Macro Search Path" in *The GNU Automake Manual*, for more information.

Then, run `./configure` as usual. Make sure to pass `--localstatedir=`*directory* where *directory* is the `localstatedir` value used by your current installation (see Section 4.3 [The Store], page 48, for information about this).

Finally, you have to invoke `make check` to run tests (see Section 2.3 [Running the Test Suite], page 6). If anything fails, take a look at installation instructions (see Chapter 2 [Installation], page 3) or send a message to the mailing list.

7.2 Running Guix Before It Is Installed

In order to keep a sane working environment, you will find it useful to test the changes made in your local source tree checkout without actually installing them. So that you can distinguish between your "end-user" hat and your "motley" costume.

To that end, all the command-line tools can be used even if you have not run `make install`. To do that, prefix each command with `./pre-inst-env` (the `pre-inst-env` script lives in the top build tree of Guix), as in:

```
$ sudo ./pre-inst-env guix-daemon --build-users-group=guixbuild
$ ./pre-inst-env guix build hello
```

Similarly, for a Guile session using the Guix modules:

```
$ ./pre-inst-env guile -c '(use-modules (guix utils)) (pk (%current-system))'

;;; ("x86_64-linux")
```

... and for a REPL (see Section "Using Guile Interactively" in *Guile Reference Manual*):

```
$ ./pre-inst-env guile
scheme@(guile-user)> ,use(guix)
scheme@(guile-user)> ,use(gnu)
scheme@(guile-user)> (define snakes
                       (fold-packages
                         (lambda (package lst)
                           (if (string-prefix? "python"
                                               (package-name package))
                               (cons package lst)
                               lst))
                         '()))
scheme@(guile-user)> (length snakes)
$1 = 361
```

The `pre-inst-env` script sets up all the environment variables necessary to support this, including `PATH` and `GUILE_LOAD_PATH`.

Note that `./pre-inst-env guix pull` does *not* upgrade the local source tree; it simply updates the `~/.config/guix/latest` symlink (see Section 3.6 [Invoking guix pull], page 29). Run `git pull` instead if you want to upgrade your local source tree.[1]

7.3 The Perfect Setup

The Perfect Setup to hack on Guix is basically the perfect setup used for Guile hacking (see Section "Using Guile in Emacs" in *Guile Reference Manual*). First, you need more than

[1] If you would like to set up `guix` to use your Git checkout, you can point the `~/.config/guix/latest` symlink to your Git checkout directory. If you are the sole user of your system, you may also consider pointing the `/root/.config/guix/latest` symlink to point to `~/.config/guix/latest`; this way it will always use the same `guix` as your user does.

an editor, you need Emacs (`http://www.gnu.org/software/emacs`), empowered by the wonderful Geiser (`http://nongnu.org/geiser/`).

Geiser allows for interactive and incremental development from within Emacs: code compilation and evaluation from within buffers, access to on-line documentation (docstrings), context-sensitive completion, `M-.` to jump to an object definition, a REPL to try out your code, and more (see Section "Introduction" in *Geiser User Manual*). For convenient Guix development, make sure to augment Guile's load path so that it finds source files from your checkout:

```
;; Assuming the Guix checkout is in ~/src/guix.
(with-eval-after-load 'geiser-guile
  (add-to-list 'geiser-guile-load-path "~/src/guix"))
```

To actually edit the code, Emacs already has a neat Scheme mode. But in addition to that, you must not miss Paredit (`http://www.emacswiki.org/emacs/ParEdit`). It provides facilities to directly operate on the syntax tree, such as raising an s-expression or wrapping it, swallowing or rejecting the following s-expression, etc.

7.4 Coding Style

In general our code follows the GNU Coding Standards (see *GNU Coding Standards*). However, they do not say much about Scheme, so here are some additional rules.

7.4.1 Programming Paradigm

Scheme code in Guix is written in a purely functional style. One exception is code that involves input/output, and procedures that implement low-level concepts, such as the `memoize` procedure.

7.4.2 Modules

Guile modules that are meant to be used on the builder side must live in the `(guix build ...)` name space. They must not refer to other Guix or GNU modules. However, it is OK for a "host-side" module to use a build-side module.

Modules that deal with the broader GNU system should be in the `(gnu ...)` name space rather than `(guix ...)`.

7.4.3 Data Types and Pattern Matching

The tendency in classical Lisp is to use lists to represent everything, and then to browse them "by hand" using `car`, `cdr`, `cadr`, and co. There are several problems with that style, notably the fact that it is hard to read, error-prone, and a hindrance to proper type error reports.

Guix code should define appropriate data types (for instance, using `define-record-type*`) rather than abuse lists. In addition, it should use pattern matching, via Guile's `(ice-9 match)` module, especially when matching lists.

7.4.4 Formatting Code

When writing Scheme code, we follow common wisdom among Scheme programmers. In general, we follow the Riastradh's Lisp Style Rules (`http://mumble.net/~campbell/`

`scheme/style.txt`). This document happens to describe the conventions mostly used in Guile's code too. It is very thoughtful and well written, so please do read it.

Some special forms introduced in Guix, such as the `substitute*` macro, have special indentation rules. These are defined in the `.dir-locals.el` file, which Emacs automatically uses. Also note that Emacs-Guix provides `guix-devel-mode` mode that indents and highlights Guix code properly (see Section "Development" in *The Emacs-Guix Reference Manual*).

If you do not use Emacs, please make sure to let your editor knows these rules. To automatically indent a package definition, you can also run:

```
./etc/indent-code.el gnu/packages/file.scm package
```

This automatically indents the definition of *package* in `gnu/packages/file.scm` by running Emacs in batch mode. To indent a whole file, omit the second argument:

```
./etc/indent-code.el gnu/services/file.scm
```

We require all top-level procedures to carry a docstring. This requirement can be relaxed for simple private procedures in the `(guix build ...)` name space, though.

Procedures should not have more than four positional parameters. Use keyword parameters for procedures that take more than four parameters.

7.5 Submitting Patches

Development is done using the Git distributed version control system. Thus, access to the repository is not strictly necessary. We welcome contributions in the form of patches as produced by `git format-patch` sent to the `guix-patches@gnu.org` mailing list.

This mailing list is backed by a Debbugs instance accessible at `https://bugs.gnu.org/guix-patches`, which allows us to keep track of submissions. Each message sent to that mailing list gets a new tracking number assigned; people can then follow up on the submission by sending email to *NNN*`@debbugs.gnu.org`, where *NNN* is the tracking number. When sending a patch series, please first send one message to `guix-patches@gnu.org`, and then send subsequent patches to *NNN*`@debbugs.gnu.org` to make sure they are kept together. See the Debbugs documentation (`https://debbugs.gnu.org/Advanced.html`), for more information.

Please write commit logs in the ChangeLog format (see Section "Change Logs" in *GNU Coding Standards*); you can check the commit history for examples.

Before submitting a patch that adds or modifies a package definition, please run through this check list:

1. Take some time to provide an adequate synopsis and description for the package. See Section 6.7.4 [Synopses and Descriptions], page 234, for some guidelines.
2. Run `guix lint` *package*, where *package* is the name of the new or modified package, and fix any errors it reports (see Section 5.7 [Invoking guix lint], page 80).
3. Make sure the package builds on your platform, using `guix build` *package*.
4. Make sure the package does not use bundled copies of software already available as separate packages.

   Sometimes, packages include copies of the source code of their dependencies as a convenience for users. However, as a distribution, we want to make sure that such packages

end up using the copy we already have in the distribution, if there is one. This improves resource usage (the dependency is built and stored only once), and allows the distribution to make transverse changes such as applying security updates for a given software package in a single place and have them affect the whole system—something that bundled copies prevent.

5. Take a look at the profile reported by `guix size` (see Section 5.8 [Invoking guix size], page 81). This will allow you to notice references to other packages unwillingly retained. It may also help determine whether to split the package (see Section 3.4 [Packages with Multiple Outputs], page 27), and which optional dependencies should be used.

6. For important changes, check that dependent package (if applicable) are not affected by the change; `guix refresh --list-dependent` *package* will help you do that (see Section 5.6 [Invoking guix refresh], page 77).

   Depending on the number of dependent packages and thus the amount of rebuilding induced, commits go to different branches, along these lines:

   300 dependent packages or less
   : `master` branch (non-disruptive changes).

   between 300 and 1,200 dependent packages
   : `staging` branch (non-disruptive changes). This branch is intended to be merged in `master` every 3 weeks or so. Topical changes (e.g., an update of the GNOME stack) can instead go to a specific branch (say, `gnome-updates`).

   more than 1,200 dependent packages
   : `core-updates` branch (may include major and potentially disruptive changes). This branch is intended to be merged in `master` every 2.5 months or so.

   All these branches are tracked by our build farm and merged into `master` once everything has been successfully built. This allows us to fix issues before they hit users, and to reduce the window during which pre-built binaries are not available.

7. Check whether the package's build process is deterministic. This typically means checking whether an independent build of the package yields the exact same result that you obtained, bit for bit.

   A simple way to do that is by building the same package several times in a row on your machine (see Section 5.1 [Invoking guix build], page 64):

   ```
   guix build --rounds=2 my-package
   ```

   This is enough to catch a class of common non-determinism issues, such as timestamps or randomly-generated output in the build result.

   Another option is to use `guix challenge` (see Section 5.12 [Invoking guix challenge], page 92). You may run it once the package has been committed and built by `hydra.gnu.org` to check whether it obtains the same result as you did. Better yet: Find another machine that can build it and run `guix publish`. Since the remote build machine is likely different from yours, this can catch non-determinism issues related to the hardware—e.g., use of different instruction set extensions—or to the operating system kernel—e.g., reliance on `uname` or `/proc` files.

8. When writing documentation, please use gender-neutral wording when referring to people, such as singular "they", "their", "them" (`https://en.wikipedia.org/wiki/Singular_they`), and so forth.
9. Verify that your patch contains only one set of related changes. Bundling unrelated changes together makes reviewing harder and slower.

   Examples of unrelated changes include the addition of several packages, or a package update along with fixes to that package.
10. Please follow our code formatting rules, possibly running the `etc/indent-code.el` script to do that automatically for you (see Section 7.4.4 [Formatting Code], page 244).

When posting a patch to the mailing list, use '`[PATCH] ...`' as a subject. You may use your email client or the `git send-email` command. We prefer to get patches in plain text messages, either inline or as MIME attachments. You are advised to pay attention if your email client changes anything like line breaks or indentation which could potentially break the patches.

8 Acknowledgments

Guix is based on the Nix package manager (`http://nixos.org/nix/`), which was designed and implemented by Eelco Dolstra, with contributions from other people (see the `nix/AUTHORS` file in Guix.) Nix pioneered functional package management, and promoted unprecedented features, such as transactional package upgrades and rollbacks, per-user profiles, and referentially transparent build processes. Without this work, Guix would not exist.

The Nix-based software distributions, Nixpkgs and NixOS, have also been an inspiration for Guix.

GNU Guix itself is a collective work with contributions from a number of people. See the `AUTHORS` file in Guix for more information on these fine people. The `THANKS` file lists people who have helped by reporting bugs, taking care of the infrastructure, providing artwork and themes, making suggestions, and more—thank you!

Appendix A GNU Free Documentation License

Version 1.3, 3 November 2008

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or non-commercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released

under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

   If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

   If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

   If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

   It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

   You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

   A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any,

be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their

titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

   You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

   The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

   In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

   If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

   However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

   Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

   Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `http://www.gnu.org/copyleft/`.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

    "Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

    "CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

    "Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

    An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

    The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.3
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
Texts.  A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with. . . Texts." line with this:

```
with the Invariant Sections being list their titles, with
the Front-Cover Texts being list, and with the Back-Cover Texts
being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Concept Index

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Programming Index

L

M

N

O

P

Q

R

S

T

U

V

W

X