GNU Guix Cookbook

Tutorials and examples for using the GNU Guix Functional Package Manager

The GNU Guix Developers

Table of Contents

GNU Guix Cookbook

This document presents tutorials and detailed examples for GNU Guix, a functional package management tool written for the GNU system. Please see *GNU Guix reference manual* for details about the system, its API, and related concepts.

This manual is also available in French (see *Livre de recettes de GNU Guix*) and German (see *GNU-Guix-Kochbuch*). If you would like to translate this document in your native language, consider joining Weblate (`https://translate.fedoraproject.org/projects/guix/documentation-cookbook`) (see Section "Translating Guix" in *GNU Guix reference manual*).

1 Scheme tutorials

GNU Guix is written in the general purpose programming language Scheme, and many of its features can be accessed and manipulated programmatically. You can use Scheme to generate package definitions, to modify them, to build them, to deploy whole operating systems, etc.

Knowing the basics of how to program in Scheme will unlock many of the advanced features Guix provides — and you don't even need to be an experienced programmer to use them!

Let's get started!

1.1 A Scheme Crash Course

Guix uses the Guile implementation of Scheme. To start playing with the language, install it with `guix install guile` and start a *REPL*—short for *read-eval-print loop* (`https://en.wikipedia.org/wiki/Read%E2%80%93eval%E2%80%93print_loop`)—by running `guile` from the command line.

Alternatively you can also run `guix environment --ad-hoc guile -- guile` if you'd rather not have Guile installed in your user profile.

In the following examples, lines show what you would type at the REPL; lines starting with "⇒" show evaluation results, while lines starting with "⊣" show things that get printed. See Section "Using Guile Interactively" in *GNU Guile Reference Manual*, for more details on the REPL.

- Scheme syntax boils down to a tree of expressions (or *s-expression* in Lisp lingo). An expression can be a literal such as numbers and strings, or a compound which is a parenthesized list of compounds and literals. `#true` and `#false` (abbreviated `#t` and `#f`) stand for the Booleans "true" and "false", respectively.

  Examples of valid expressions:

  ```
  "Hello World!"
  ⇒ "Hello World!"

  17
  ⇒ 17

  (display (string-append "Hello " "Guix" "\n"))
  ⊣ Hello Guix!
  ⇒ #<unspecified>
  ```

- This last example is a function call nested in another function call. When a parenthesized expression is evaluated, the first term is the function and the rest are the arguments passed to the function. Every function returns the last evaluated expression as its return value.
- Anonymous functions are declared with the `lambda` term:

  ```
  (lambda (x) (* x x))
  ⇒ #<procedure 120e348 at <unknown port>:24:0 (x)>
  ```

The above procedure returns the square of its argument. Since everything is an expression, the `lambda` expression returns an anonymous procedure, which can in turn be applied to an argument:

```
((lambda (x) (* x x)) 3)
⇒ 9
```

- Anything can be assigned a global name with `define`:

```
(define a 3)
(define square (lambda (x) (* x x)))
(square a)
⇒ 9
```

- Procedures can be defined more concisely with the following syntax:

```
(define (square x) (* x x))
```

- A list structure can be created with the `list` procedure:

```
(list 2 a 5 7)
⇒ (2 3 5 7)
```

- The *quote* disables evaluation of a parenthesized expression: the first term is not called over the other terms (see Section "Expression Syntax" in *GNU Guile Reference Manual*). Thus it effectively returns a list of terms.

```
'(display (string-append "Hello " "Guix" "\n"))
⇒ (display (string-append "Hello " "Guix" "\n"))

'(2 a 5 7)
⇒ (2 a 5 7)
```

- The *quasiquote* disables evaluation of a parenthesized expression until *unquote* (a comma) re-enables it. Thus it provides us with fine-grained control over what is evaluated and what is not.

```
`(2 a 5 7 (2 ,a 5 ,(+ a 4)))
⇒ (2 a 5 7 (2 3 5 7))
```

Note that the above result is a list of mixed elements: numbers, symbols (here `a`) and the last element is a list itself.

- Multiple variables can be named locally with `let` (see Section "Local Bindings" in *GNU Guile Reference Manual*):

```
(define x 10)
(let ((x 2)
      (y 3))
  (list x y))
⇒ (2 3)

x
⇒ 10

y
error In procedure module-lookup: Unbound variable: y
```

Use `let*` to allow later variable declarations to refer to earlier definitions.

```
(let* ((x 2)
       (y (* x 3)))
  (list x y))
⇒ (2 6)
```

- *Keywords* are typically used to identify the named parameters of a procedure. They are prefixed by `#:` (hash, colon) followed by alphanumeric characters: `#:like-this`. See Section “Keywords” in *GNU Guile Reference Manual*.
- The percentage `%` is typically used for read-only global variables in the build stage. Note that it is merely a convention, like `_` in C. Scheme treats `%` exactly the same as any other letter.
- Modules are created with `define-module` (see Section “Creating Guile Modules” in *GNU Guile Reference Manual*). For instance

  ```
  (define-module (guix build-system ruby)
    #:use-module (guix store)
    #:export (ruby-build
              ruby-build-system))
  ```

  defines the module `guix build-system ruby` which must be located in `guix/build-system/ruby.scm` somewhere in the Guile load path. It depends on the `(guix store)` module and it exports two variables, `ruby-build` and `ruby-build-system`.

For a more detailed introduction, check out Scheme at a Glance (`http://www.troubleshooters.com/codecorn/scheme_guile/hello.htm`), by Steve Litt.

One of the reference Scheme books is the seminal “Structure and Interpretation of Computer Programs”, by Harold Abelson and Gerald Jay Sussman, with Julie Sussman. You’ll find a free copy online (`https://mitpress.mit.edu/sites/default/files/sicp/index.html`), together with videos of the lectures by the authors (`https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-001-structure-and-interpretation-of-computer-programs-spring-2005/video-lectures/`). The book is available in Texinfo format as the `sicp` Guix package. Go ahead, run `guix install sicp` and start reading with `info sicp` (see *Structure and Interpretation of Computer Programs*). An unofficial ebook is also available (`https://sarabander.github.io/sicp/`).

You’ll find more books, tutorials and other resources at `https://schemers.org/`.

2 Packaging

This chapter is dedicated to teaching you how to add packages to the collection of packages that come with GNU Guix. This involves writing package definitions in Guile Scheme, organizing them in package modules, and building them.

2.1 Packaging Tutorial

GNU Guix stands out as the *hackable* package manager, mostly because it uses GNU Guile (`https://www.gnu.org/software/guile/`), a powerful high-level programming language, one of the Scheme (`https://en.wikipedia.org/wiki/Scheme_%28programming_language%29`) dialects from the Lisp family (`https://en.wikipedia.org/wiki/Lisp_%28programming_language%29`).

Package definitions are also written in Scheme, which empowers Guix in some very unique ways, unlike most other package managers that use shell scripts or simple languages.

- Use functions, structures, macros and all of Scheme expressiveness for your package definitions.
- Inheritance makes it easy to customize a package by inheriting from it and modifying only what is needed.
- Batch processing: the whole package collection can be parsed, filtered and processed. Building a headless server with all graphical interfaces stripped out? It's possible. Want to rebuild everything from source using specific compiler optimization flags? Pass the `#:make-flags "..."` argument to the list of packages. It wouldn't be a stretch to think Gentoo USE flags (`https://wiki.gentoo.org/wiki/USE_flag`) here, but this goes even further: the changes don't have to be thought out beforehand by the packager, they can be *programmed* by the user!

The following tutorial covers all the basics around package creation with Guix. It does not assume much knowledge of the Guix system nor of the Lisp language. The reader is only expected to be familiar with the command line and to have some basic programming knowledge.

2.1.1 A "Hello World" package

The "Defining Packages" section of the manual introduces the basics of Guix packaging (see Section "Defining Packages" in *GNU Guix Reference Manual*). In the following section, we will partly go over those basics again.

GNU Hello is a dummy project that serves as an idiomatic example for packaging. It uses the GNU build system (`./configure && make && make install`). Guix already provides a package definition which is a perfect example to start with. You can look up its declaration with `guix edit hello` from the command line. Let's see how it looks:

```
(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)
         (synopsis "Hello, GNU world: An example GNU package")
         (description
          "GNU Hello prints the message \"Hello, world!\" and then exits.  It
     serves as an example of standard GNU coding practices.  As such, it supports
     command-line arguments, multiple languages, and so on.")
         (home-page "https://www.gnu.org/software/hello/")
         (license gpl3+)))
```

As you can see, most of it is rather straightforward. But let's review the fields together:

'`name`' The project name. Using Scheme conventions, we prefer to keep it lower case, without underscore and using dash-separated words.

'`source`' This field contains a description of the source code origin. The `origin` record contains these fields:

1. The method, here `url-fetch` to download via HTTP/FTP, but other methods exist, such as `git-fetch` for Git repositories.
2. The URI, which is typically some `https://` location for `url-fetch`. Here the special 'mirror://gnu' refers to a set of well known locations, all of which can be used by Guix to fetch the source, should some of them fail.
3. The `sha256` checksum of the requested file. This is essential to ensure the source is not corrupted. Note that Guix works with base32 strings, hence the call to the `base32` function.

'`build-system`'
This is where the power of abstraction provided by the Scheme language really shines: in this case, the `gnu-build-system` abstracts away the famous `./configure && make && make install` shell invocations. Other build systems include the `trivial-build-system` which does not do anything and requires from the packager to program all the build steps, the `python-build-system`, the `emacs-build-system`, and many more (see Section "Build Systems" in *GNU Guix Reference Manual*).

'`synopsis`'
It should be a concise summary of what the package does. For many packages a tagline from the project's home page can be used as the synopsis.

'`description`'
Same as for the synopsis, it's fine to re-use the project description from the homepage. Note that Guix uses Texinfo syntax.

'`home-page`'
Use HTTPS if available.

'`license`' See `guix/licenses.scm` in the project source for a full list of available licenses.

Time to build our first package! Nothing fancy here for now: we will stick to a dummy `my-hello`, a copy of the above declaration.

As with the ritualistic "Hello World" taught with most programming languages, this will possibly be the most "manual" approach. We will work out an ideal setup later; for now we will go the simplest route.

Save the following to a file `my-hello.scm`.

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

(package
  (name "my-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, Guix world: An example custom Guix package")
  (description
   "GNU Hello prints the message \"Hello, world!\" and then exits.  It
serves as an example of standard GNU coding practices.  As such, it supports
command-line arguments, multiple languages, and so on.")
  (home-page "https://www.gnu.org/software/hello/")
  (license gpl3+))
```

We will explain the extra code in a moment.

Feel free to play with the different values of the various fields. If you change the source, you'll need to update the checksum. Indeed, Guix refuses to build anything if the given checksum does not match the computed checksum of the source code. To obtain the correct checksum of the package declaration, we need to download the source, compute the sha256 checksum and convert it to base32.

Thankfully, Guix can automate this task for us; all we need is to provide the URI:

```
sh
$ guix download mirror://gnu/hello/hello-2.10.tar.gz

Starting download of /tmp/guix-file.JLYgL7
From https://ftpmirror.gnu.org/gnu/hello/hello-2.10.tar.gz...
following redirection to ‘https://mirror.ibcp.fr/pub/gnu/hello/hello-2.10.tar.gz’...
 ...10.tar.gz  709KiB                                 2.5MiB/s 00:00 [################
/gnu/store/hbdalsf5lpf01x4dcknwx6xbn6n5km6k-hello-2.10.tar.gz
0ssi1wpaf7plaswqqjwigppsg5fyh99vdlb9kzl7c9lng89ndq1i
```

In this specific case the output tells us which mirror was chosen. If the result of the above command is not the same as in the above snippet, update your `my-hello` declaration accordingly.

Note that GNU package tarballs come with an OpenPGP signature, so you should definitely check the signature of this tarball with ‘gpg‘ to authenticate it before going further:

```
sh
$ guix download mirror://gnu/hello/hello-2.10.tar.gz.sig

Starting download of /tmp/guix-file.03tFfb
From https://ftpmirror.gnu.org/gnu/hello/hello-2.10.tar.gz.sig...
following redirection to ‘https://ftp.igh.cnrs.fr/pub/gnu/hello/hello-2.10.tar.gz.sig’
 ....tar.gz.sig  819B
/gnu/store/rzs8wba9ka7grrmgcpfyxvs58mly0sx6-hello-2.10.tar.gz.sig
0q0v86n3y38z17rl146gdakw9xc4mcscpk8dscs412j22glrv9jf
$ gpg --verify /gnu/store/rzs8wba9ka7grrmgcpfyxvs58mly0sx6-hello-2.10.tar.gz.sig /gnu/
gpg: Signature made Sun 16 Nov 2014 01:08:37 PM CET
gpg:                using RSA key A9553245FDE9B739
gpg: Good signature from "Sami Kerola <kerolasa@iki.fi>" [unknown]
gpg:                 aka "Sami Kerola (http://www.iki.fi/kerolasa/) <kerolasa@iki.fi>"
gpg: WARNING: This key is not certified with a trusted signature!
gpg:          There is no indication that the signature belongs to the owner.
Primary key fingerprint: 8ED3 96E3 7E38 D471 A005  30D3 A955 3245 FDE9 B739
```

You can then happily run

```
sh
$ guix package --install-from-file=my-hello.scm
```

You should now have `my-hello` in your profile!

```
sh
$ guix package --list-installed=my-hello
my-hello 2.10 out
/gnu/store/f1db2mfm8syb8qvc357c53slbvf1g9m9-my-hello-2.10
```

We’ve gone as far as we could without any knowledge of Scheme. Before moving on to more complex packages, now is the right time to brush up on your Scheme knowledge. see Section 1.1 [A Scheme Crash Course], page 2, to get up to speed.

2.1.2 Setup

In the rest of this chapter we will rely on some basic Scheme programming knowledge. Now let’s detail the different possible setups for working on Guix packages.

There are several ways to set up a Guix packaging environment.

We recommend you work directly on the Guix source checkout since it makes it easier for everyone to contribute to the project.

But first, let’s look at other possibilities.

2.1.2.1 Local file

This is what we previously did with ‘`my-hello`’. With the Scheme basics we’ve covered, we are now able to explain the leading chunks. As stated in `guix package --help`:

```
-f, --install-from-file=FILE
                        install the package that the code within FILE
                        evaluates to
```

Thus the last expression *must* return a package, which is the case in our earlier example.

The `use-modules` expression tells which of the modules we need in the file. Modules are a collection of values and procedures. They are commonly called “libraries” or “packages” in other programming languages.

2.1.2.2 ‘GUIX_PACKAGE_PATH’

Note: Starting from Guix 0.16, the more flexible Guix channels are the preferred way and supersede ‘`GUIX_PACKAGE_PATH`*’. See next section.*

It can be tedious to specify the file from the command line instead of simply calling `guix package --install my-hello` as you would do with the official packages.

Guix makes it possible to streamline the process by adding as many “package declaration directories” as you want.

Create a directory, say `~/guix-packages` and add it to the ‘`GUIX_PACKAGE_PATH`’ environment variable:

```
$ mkdir ~/guix-packages
$ export GUIX_PACKAGE_PATH=~/guix-packages
```

To add several directories, separate them with a colon (`:`).

Our previous ‘`my-hello`’ needs some adjustments though:

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

(define-public my-hello
  (package
    (name "my-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, Guix world: An example custom Guix package")
    (description
     "GNU Hello prints the message \"Hello, world!\" and then exits.  It
serves as an example of standard GNU coding practices.  As such, it supports
command-line arguments, multiple languages, and so on.")
    (home-page "https://www.gnu.org/software/hello/")
```

```
    (license gpl3+)))
```

Note that we have assigned the package value to an exported variable name with `define-public`. This is effectively assigning the package to the `my-hello` variable so that it can be referenced, among other as dependency of other packages.

If you use `guix package --install-from-file=my-hello.scm` on the above file, it will fail because the last expression, `define-public`, does not return a package. If you want to use `define-public` in this use-case nonetheless, make sure the file ends with an evaluation of `my-hello`:

```
; ...
(define-public my-hello
  ; ...
  )

my-hello
```

This last example is not very typical.

Now '`my-hello`' should be part of the package collection like all other official packages. You can verify this with:

```
$ guix package --show=my-hello
```

2.1.2.3 Guix channels

Guix 0.16 features channels, which is very similar to '`GUIX_PACKAGE_PATH`' but provides better integration and provenance tracking. Channels are not necessarily local, they can be maintained as a public Git repository for instance. Of course, several channels can be used at the same time.

See Section "Channels" in *GNU Guix Reference Manual* for setup details.

2.1.2.4 Direct checkout hacking

Working directly on the Guix project is recommended: it reduces the friction when the time comes to submit your changes upstream to let the community benefit from your hard work!

Unlike most software distributions, the Guix repository holds in one place both the tooling (including the package manager) and the package definitions. This choice was made so that it would give developers the flexibility to modify the API without breakage by updating all packages at the same time. This reduces development inertia.

Check out the official Git (`https://git-scm.com/`) repository:

```
$ git clone https://git.savannah.gnu.org/git/guix.git
```

In the rest of this article, we use '`$GUIX_CHECKOUT`' to refer to the location of the checkout.

Follow the instructions in the manual (see Section "Contributing" in *GNU Guix Reference Manual*) to set up the repository environment.

Once ready, you should be able to use the package definitions from the repository environment.

Feel free to edit package definitions found in '`$GUIX_CHECKOUT/gnu/packages`'.

The '$GUIX_CHECKOUT/pre-inst-env' script lets you use 'guix' over the package collection of the repository (see Section "Running Guix Before It Is Installed" in *GNU Guix Reference Manual*).

- Search packages, such as Ruby:

```
$ cd $GUIX_CHECKOUT
$ ./pre-inst-env guix package --list-available=ruby
    ruby    1.8.7-p374      out     gnu/packages/ruby.scm:119:2
    ruby    2.1.6   out     gnu/packages/ruby.scm:91:2
    ruby    2.2.2   out     gnu/packages/ruby.scm:39:2
```

- Build a package, here Ruby version 2.1:

```
$ ./pre-inst-env guix build --keep-failed ruby@2.1
/gnu/store/c13v73jxmj2nir2xjqaz5259zywsa9zi-ruby-2.1.6
```

- Install it to your user profile:

```
$ ./pre-inst-env guix package --install ruby@2.1
```

- Check for common mistakes:

```
$ ./pre-inst-env guix lint ruby@2.1
```

Guix strives at maintaining a high packaging standard; when contributing to the Guix project, remember to

- follow the coding style (see Section "Coding Style" in *GNU Guix Reference Manual*),
- and review the check list from the manual (see Section "Submitting Patches" in *GNU Guix Reference Manual*).

Once you are happy with the result, you are welcome to send your contribution to make it part of Guix. This process is also detailed in the manual. (see Section "Contributing" in *GNU Guix Reference Manual*)

It's a community effort so the more join in, the better Guix becomes!

2.1.3 Extended example

The above "Hello World" example is as simple as it goes. Packages can be more complex than that and Guix can handle more advanced scenarios. Let's look at another, more sophisticated package (slightly modified from the source):

```
(define-module (gnu packages version-control)
  #:use-module ((guix licenses) #:prefix license:)
  #:use-module (guix utils)
  #:use-module (guix packages)
  #:use-module (guix git-download)
  #:use-module (guix build-system cmake)
  #:use-module (gnu packages ssh)
  #:use-module (gnu packages web)
  #:use-module (gnu packages pkg-config)
  #:use-module (gnu packages python)
  #:use-module (gnu packages compression)
  #:use-module (gnu packages tls))
```

```
(define-public my-libgit2
  (let ((commit "e98d0a37c93574d2c6107bf7f31140b548c6a7bf")
        (revision "1"))
    (package
      (name "my-libgit2")
      (version (git-version "0.26.6" revision commit))
      (source (origin
                (method git-fetch)
                (uri (git-reference
                      (url "https://github.com/libgit2/libgit2/")
                      (commit commit)))
                (file-name (git-file-name name version))
                (sha256
                 (base32
                  "17pjvprmdrx4h6bb1hhc98w9qi6ki7yl57f090n9kbhswxqfs7s3"))
                (patches (search-patches "libgit2-mtime-0.patch"))
                (modules '((guix build utils)))
                (snippet '(begin
                            ;; Remove bundled software.
                            (delete-file-recursively "deps")
                            #true))))
      (build-system cmake-build-system)
      (outputs '("out" "debug"))
      (arguments
       `(#:tests? #true                         ; Run the test suite (this is the defa
         #:configure-flags '("-DUSE_SHA1DC=ON") ; SHA-1 collision detection
         #:phases
         (modify-phases %standard-phases
           (add-after 'unpack 'fix-hardcoded-paths
             (lambda _
               (substitute* "tests/repo/init.c"
                 (("#!/bin/sh") (string-append "#!" (which "sh"))))
               (substitute* "tests/clar/fs.h"
                 (("/bin/cp") (which "cp"))
                 (("/bin/rm") (which "rm")))
               #true))
           ;; Run checks more verbosely.
           (replace 'check
             (lambda _ (invoke "./libgit2_clar" "-v" "-Q")))
           (add-after 'unpack 'make-files-writable-for-tests
             (lambda _ (for-each make-file-writable (find-files "." ".*")))))))
      (inputs
       `(("libssh2" ,libssh2)
         ("http-parser" ,http-parser)
         ("python" ,python-wrapper)))
      (native-inputs
       `(("pkg-config" ,pkg-config)))
```

```
    (propagated-inputs
     ;; These two libraries are in 'Requires.private' in libgit2.pc.
     `(("openssl" ,openssl)
       ("zlib" ,zlib)))
    (home-page "https://libgit2.github.com/")
    (synopsis "Library providing Git core methods")
    (description
     "Libgit2 is a portable, pure C implementation of the Git core methods
provided as a re-entrant linkable library with a solid API, allowing you to
write native speed custom Git applications in any language with bindings.")
    ;; GPLv2 with linking exception
    (license license:gpl2))))
```

(In those cases were you only want to tweak a few fields from a package definition, you should rely on inheritance instead of copy-pasting everything. See below.)

Let's discuss those fields in depth.

2.1.3.1 `git-fetch` method

Unlike the `url-fetch` method, `git-fetch` expects a `git-reference` which takes a Git repository and a commit. The commit can be any Git reference such as tags, so if the `version` is tagged, then it can be used directly. Sometimes the tag is prefixed with a `v`, in which case you'd use `(commit (string-append "v" version))`.

To ensure that the source code from the Git repository is stored in a directory with a descriptive name, we use `(file-name (git-file-name name version))`.

The `git-version` procedure can be used to derive the version when packaging programs for a specific commit, following the Guix contributor guidelines (see Section "Version Numbers" in *GNU Guix Reference Manual*).

How does one obtain the `sha256` hash that's in there, you ask? By invoking `guix hash` on a checkout of the desired commit, along these lines:

```
git clone https://github.com/libgit2/libgit2/
cd libgit2
git checkout v0.26.6
guix hash -rx .
```

`guix hash -rx` computes a SHA256 hash over the whole directory, excluding the `.git` sub-directory (see Section "Invoking guix hash" in *GNU Guix Reference Manual*).

In the future, `guix download` will hopefully be able to do these steps for you, just like it does for regular downloads.

2.1.3.2 Snippets

Snippets are quoted (i.e. non-evaluated) Scheme code that are a means of patching the source. They are a Guix-y alternative to the traditional `.patch` files. Because of the quote, the code in only evaluated when passed to the Guix daemon for building. There can be as many snippets as needed.

Snippets might need additional Guile modules which can be imported from the `modules` field.

2.1.3.3 Inputs

First, a syntactic comment: See the quasi-quote / comma syntax?

```
(native-inputs
 `(("pkg-config" ,pkg-config)))
```

is equivalent to

```
(native-inputs
 (list (list "pkg-config" pkg-config)))
```

You'll mostly see the former because it's shorter.

There are 3 different input types. In short:

native-inputs
: Required for building but not runtime – installing a package through a substitute won't install these inputs.

inputs
: Installed in the store but not in the profile, as well as being present at build time.

propagated-inputs
: Installed in the store and in the profile, as well as being present at build time.

See Section "Package Reference" in *GNU Guix Reference Manual* for more details.

The distinction between the various inputs is important: if a dependency can be handled as an *input* instead of a *propagated input*, it should be done so, or else it "pollutes" the user profile for no good reason.

For instance, a user installing a graphical program that depends on a command line tool might only be interested in the graphical part, so there is no need to force the command line tool into the user profile. The dependency is a concern to the package, not to the user. *Inputs* make it possible to handle dependencies without bugging the user by adding undesired executable files (or libraries) to their profile.

Same goes for *native-inputs*: once the program is installed, build-time dependencies can be safely garbage-collected. It also matters when a substitute is available, in which case only the *inputs* and *propagated inputs* will be fetched: the *native inputs* are not required to install a package from a substitute.

2.1.3.4 Outputs

Just like how a package can have multiple inputs, it can also produce multiple outputs.

Each output corresponds to a separate directory in the store.

The user can choose which output to install; this is useful to save space or to avoid polluting the user profile with unwanted executables or libraries.

Output separation is optional. When the `outputs` field is left out, the default and only output (the complete package) is referred to as `"out"`.

Typical separate output names include `debug` and `doc`.

It's advised to separate outputs only when you've shown it's worth it: if the output size is significant (compare with `guix size`) or in case the package is modular.

2.1.3.5 Build system arguments

The `arguments` is a keyword-value list used to configure the build process.

The simplest argument `#:tests?` can be used to disable the test suite when building the package. This is mostly useful when the package does not feature any test suite. It's strongly recommended to keep the test suite on if there is one.

Another common argument is `:make-flags`, which specifies a list of flags to append when running make, as you would from the command line. For instance, the following flags

```
#:make-flags (list (string-append "prefix=" (assoc-ref %outputs "out"))
                   "CC=gcc")
```

translate into

```
$ make CC=gcc prefix=/gnu/store/...-<out>
```

This sets the C compiler to `gcc` and the `prefix` variable (the installation directory in Make parlance) to `(assoc-ref %outputs "out")`, which is a build-stage global variable pointing to the destination directory in the store (something like `/gnu/store/...-my-libgit2-20180408`).

Similarly, it's possible to set the configure flags:

```
#:configure-flags '("-DUSE_SHA1DC=ON")
```

The `%build-inputs` variable is also generated in scope. It's an association table that maps the input names to their store directories.

The `phases` keyword lists the sequential steps of the build system. Typically phases include `unpack`, `configure`, `build`, `install` and `check`. To know more about those phases, you need to work out the appropriate build system definition in '`$GUIX_CHECKOUT/guix/build/gnu-build-system.scm`':

```
(define %standard-phases
  ;; Standard build phases, as a list of symbol/procedure pairs.
  (let-syntax ((phases (syntax-rules ()
                         ((_ p ...) `((p . ,p) ...)))))
    (phases set-SOURCE-DATE-EPOCH set-paths install-locale unpack
            bootstrap
            patch-usr-bin-file
            patch-source-shebangs configure patch-generated-file-shebangs
            build check install
            patch-shebangs strip
            validate-runpath
            validate-documentation-location
            delete-info-dir-file
            patch-dot-desktop-files
            install-license-files
            reset-gzip-timestamps
            compress-documentation)))
```

Or from the REPL:

```
(add-to-load-path "/path/to/guix/checkout")
,use (guix build gnu-build-system)
(map first %standard-phases)
```

```
⇒ (set-SOURCE-DATE-EPOCH set-paths install-locale unpack bootstrap patch-usr-bin-file
```

If you want to know more about what happens during those phases, consult the associated procedures.

For instance, as of this writing the definition of `unpack` for the GNU build system is:

```
(define* (unpack #:key source #:allow-other-keys)
  "Unpack SOURCE in the working directory, and change directory within the
source.  When SOURCE is a directory, copy it in a sub-directory of the current
working directory."
  (if (file-is-directory? source)
      (begin
        (mkdir "source")
        (chdir "source")

        ;; Preserve timestamps (set to the Epoch) on the copied tree so that
        ;; things work deterministically.
        (copy-recursively source "."
                          #:keep-mtime? #true))
      (begin
        (if (string-suffix? ".zip" source)
            (invoke "unzip" source)
            (invoke "tar" "xvf" source))
        (chdir (first-subdirectory "."))))
  #true)
```

Note the `chdir` call: it changes the working directory to where the source was unpacked. Thus every phase following the `unpack` will use the source as a working directory, which is why we can directly work on the source files. That is to say, unless a later phase changes the working directory to something else.

We modify the list of `%standard-phases` of the build system with the `modify-phases` macro as per the list of specified modifications, which may have the following forms:

- `(add-before` *phase new-phase procedure*`)`: Run *procedure* named *new-phase* before *phase*.
- `(add-after` *phase new-phase procedure*`)`: Same, but afterwards.
- `(replace` *phase procedure*`)`.
- `(delete` *phase*`)`.

The *procedure* supports the keyword arguments `inputs` and `outputs`. Each input (whether *native*, *propagated* or not) and output directory is referenced by their name in those variables. Thus `(assoc-ref outputs "out")` is the store directory of the main output of the package. A phase procedure may look like this:

```
(lambda* (#:key inputs outputs #:allow-other-keys)
  (let ((bash-directory (assoc-ref inputs "bash"))
        (output-directory (assoc-ref outputs "out"))
        (doc-directory (assoc-ref outputs "doc")))
    ;; ...
    #true))
```

The procedure must return `#true` on success. It's brittle to rely on the return value of the last expression used to tweak the phase because there is no guarantee it would be a `#true`. Hence the trailing `#true` to ensure the right value is returned on success.

2.1.3.6 Code staging

The astute reader may have noticed the quasi-quote and comma syntax in the argument field. Indeed, the build code in the package declaration should not be evaluated on the client side, but only when passed to the Guix daemon. This mechanism of passing code around two running processes is called code staging (`https://arxiv.org/abs/1709.00833`).

2.1.3.7 Utility functions

When customizing `phases`, we often need to write code that mimics the equivalent system invocations (`make`, `mkdir`, `cp`, etc.) commonly used during regular "Unix-style" installations.

Some like `chmod` are native to Guile. See *Guile reference manual* for a complete list.

Guix provides additional helper functions which prove especially handy in the context of package management.

Some of those functions can be found in '`$GUIX_CHECKOUT/guix/guix/build/utils.scm`'. Most of them mirror the behaviour of the traditional Unix system commands:

`which`
: Like the '`which`' system command.

`find-files`
: Akin to the '`find`' system command.

`mkdir-p`
: Like '`mkdir -p`', which creates all parents as needed.

`install-file`
: Similar to '`install`' when installing a file to a (possibly non-existing) directory. Guile has `copy-file` which works like '`cp`'.

`copy-recursively`
: Like '`cp -r`'.

`delete-file-recursively`
: Like '`rm -rf`'.

`invoke`
: Run an executable. This should be used instead of `system*`.

`with-directory-excursion`
: Run the body in a different working directory, then restore the previous working directory.

`substitute*`
: A "`sed`-like" function.

See Section "Build Utilities" in *GNU Guix Reference Manual*, for more information on these utilities.

2.1.3.8 Module prefix

The license in our last example needs a prefix: this is because of how the `license` module was imported in the package, as `#:use-module ((guix licenses) #:prefix license:)`. The Guile module import mechanism (see Section "Using Guile Modules" in *Guile reference manual*) gives the user full control over namespacing: this is needed to avoid clashes between, say, the '`zlib`' variable from '`licenses.scm`' (a *license* value) and the '`zlib`' variable from '`compression.scm`' (a *package* value).

2.1.4 Other build systems

What we've seen so far covers the majority of packages using a build system other than the `trivial-build-system`. The latter does not automate anything and leaves you to build everything manually. This can be more demanding and we won't cover it here for now, but thankfully it is rarely necessary to fall back on this system.

For the other build systems, such as ASDF, Emacs, Perl, Ruby and many more, the process is very similar to the GNU build system except for a few specialized arguments.

See Section "Build Systems" in *GNU Guix Reference Manual*, for more information on build systems, or check the source code in the '`$GUIX_CHECKOUT/guix/build`' and '`$GUIX_CHECKOUT/guix/build-system`' directories.

2.1.5 Programmable and automated package definition

We can't repeat it enough: having a full-fledged programming language at hand empowers us in ways that reach far beyond traditional package management.

Let's illustrate this with some awesome features of Guix!

2.1.5.1 Recursive importers

You might find some build systems good enough that there is little to do at all to write a package, to the point that it becomes repetitive and tedious after a while. A *raison d'être* of computers is to replace human beings at those boring tasks. So let's tell Guix to do this for us and create the package definition of an R package from CRAN (the output is trimmed for conciseness):

```
$ guix import cran --recursive walrus

(define-public r-mc2d
    ; ...
    (license gpl2+)))

(define-public r-jmvcore
    ; ...
    (license gpl2+)))

(define-public r-wrs2
    ; ...
    (license gpl3)))

(define-public r-walrus
```

```
  (package
    (name "r-walrus")
    (version "1.0.3")
    (source
      (origin
        (method url-fetch)
        (uri (cran-uri "walrus" version))
        (sha256
          (base32
            "1nk2glcvy4hyksl5ipq2mz8jy4fss90hx6cq98m3w96kzjni6jjj"))))
    (build-system r-build-system)
    (propagated-inputs
      `(("r-ggplot2" ,r-ggplot2)
        ("r-jmvcore" ,r-jmvcore)
        ("r-r6" ,r-r6)
        ("r-wrs2" ,r-wrs2)))
    (home-page "https://github.com/jamovi/walrus")
    (synopsis "Robust Statistical Methods")
    (description
      "This package provides a toolbox of common robust statistical
tests, including robust descriptives, robust t-tests, and robust ANOVA.
It is also available as a module for 'jamovi' (see
<https://www.jamovi.org> for more information).  Walrus is based on the
WRS2 package by Patrick Mair, which is in turn based on the scripts and
work of Rand Wilcox.  These analyses are described in depth in the book
'Introduction to Robust Estimation & Hypothesis Testing'.")
    (license gpl3)))
```

The recursive importer won't import packages for which Guix already has package definitions, except for the very first.

Not all applications can be packaged this way, only those relying on a select number of supported systems. Read about the full list of importers in the guix import section of the manual (see Section "Invoking guix import" in *GNU Guix Reference Manual*).

2.1.5.2 Automatic update

Guix can be smart enough to check for updates on systems it knows. It can report outdated package definitions with

```
$ guix refresh hello
```

In most cases, updating a package to a newer version requires little more than changing the version number and the checksum. Guix can do that automatically as well:

```
$ guix refresh hello --update
```

2.1.5.3 Inheritance

If you've started browsing the existing package definitions, you might have noticed that a significant number of them have a `inherit` field:

```
(define-public adwaita-icon-theme
```

```
(package (inherit gnome-icon-theme)
  (name "adwaita-icon-theme")
  (version "3.26.1")
  (source (origin
            (method url-fetch)
            (uri (string-append "mirror://gnome/sources/" name "/"
                                (version-major+minor version) "/"
                                name "-" version ".tar.xz"))
            (sha256
             (base32
              "17fpahgh5dyckgz7rwqvzgnhx53cx9kr2xw0szprc6bnqy977fi8"))))
  (native-inputs
   `(("gtk-encode-symbolic-svg" ,gtk+ "bin"))))
```

All unspecified fields are inherited from the parent package. This is very convenient to create alternative packages, for instance with different source, version or compilation options.

2.1.6 Getting help

Sadly, some applications can be tough to package. Sometimes they need a patch to work with the non-standard file system hierarchy enforced by the store. Sometimes the tests won't run properly. (They can be skipped but this is not recommended.) Other times the resulting package won't be reproducible.

Should you be stuck, unable to figure out how to fix any sort of packaging issue, don't hesitate to ask the community for help.

See the Guix homepage (`https://www.gnu.org/software/guix/contact/`) for information on the mailing lists, IRC, etc.

2.1.7 Conclusion

This tutorial was a showcase of the sophisticated package management that Guix boasts. At this point we have mostly restricted this introduction to the `gnu-build-system` which is a core abstraction layer on which more advanced abstractions are based.

Where do we go from here? Next we ought to dissect the innards of the build system by removing all abstractions, using the `trivial-build-system`: this should give us a thorough understanding of the process before investigating some more advanced packaging techniques and edge cases.

Other features worth exploring are the interactive editing and debugging capabilities of Guix provided by the Guile REPL.

Those fancy features are completely optional and can wait; now is a good time to take a well-deserved break. With what we've introduced here you should be well armed to package lots of programs. You can get started right away and hopefully we will see your contributions soon!

2.1.8 References

- The package reference in the manual (`https://www.gnu.org/software/guix/manual/en/html_node/Defining-Packages.html`)

- Pjotr's hacking guide to GNU Guix (`https://gitlab.com/pjotrp/guix-notes/blob/master/HACKING.org`)
- "GNU Guix: Package without a scheme!" (`https://www.gnu.org/software/guix/guix-ghm-andreas-20130823.pdf`), by Andreas Enge

3 System Configuration

Guix offers a flexible language for declaratively configuring your Guix System. This flexibility can at times be overwhelming. The purpose of this chapter is to demonstrate some advanced configuration concepts.

see Section "System Configuration" in *GNU Guix Reference Manual* for a complete reference.

3.1 Auto-Login to a Specific TTY

While the Guix manual explains auto-login one user to *all* TTYs (see Section "auto-login to TTY" in *GNU Guix Reference Manual*), some might prefer a situation, in which one user is logged into one TTY with the other TTYs either configured to login different users or no one at all. Note that one can auto-login one user to any TTY, but it is usually advisable to avoid `tty1`, which, by default, is used to log warnings and errors.

Here is how one might set up auto login for one user to one tty:

```
(define (auto-login-to-tty config tty user)
  (if (string=? tty (mingetty-configuration-tty config))
        (mingetty-configuration
         (inherit config)
         (auto-login user))
        config))

(define %my-services
  (modify-services %base-services
    ;; ...
    (mingetty-service-type config =>
                           (auto-login-to-tty
                            config "tty3" "alice"))))

(operating-system
  ;; ...
  (services %my-services))
```

One could also `compose` (see Section "Higher-Order Functions" in *The Guile Reference Manual*) `auto-login-to-tty` to login multiple users to multiple ttys.

Finally, here is a note of caution. Setting up auto login to a TTY, means that anyone can turn on your computer and run commands as your regular user. However, if you have an encrypted root partition, and thus already need to enter a passphrase when the system boots, auto-login might be a convenient option.

3.2 Customizing the Kernel

Guix is, at its core, a source based distribution with substitutes (see Section "Substitutes" in *GNU Guix Reference Manual*), and as such building packages from their source code is an expected part of regular package installations and upgrades. Given this starting point, it makes sense that efforts are made to reduce the amount of time spent compiling packages, and recent changes and upgrades to the building and distribution of substitutes continues to be a topic of discussion within Guix.

The kernel, while not requiring an overabundance of RAM to build, does take a rather long time on an average machine. The official kernel configuration, as is the case with many GNU/Linux distributions, errs on the side of inclusiveness, and this is really what causes the build to take such a long time when the kernel is built from source.

The Linux kernel, however, can also just be described as a regular old package, and as such can be customized just like any other package. The procedure is a little bit different, although this is primarily due to the nature of how the package definition is written.

The `linux-libre` kernel package definition is actually a procedure which creates a package.

```
(define* (make-linux-libre version hash supported-systems
                           #:key
                           ;; A function that takes an arch and a variant.
                           ;; See kernel-config for an example.
                           (extra-version #false)
                           (configuration-file #false)
                           (defconfig "defconfig")
                           (extra-options %default-extra-linux-options)
                           (patches (list %boot-logo-patch)))
  ...)
```

The current `linux-libre` package is for the 5.1.x series, and is declared like this:

```
(define-public linux-libre
  (make-linux-libre %linux-libre-version
                    %linux-libre-hash
                    '("x86_64-linux" "i686-linux" "armhf-linux" "aarch64-linux")
                    #:patches %linux-libre-5.1-patches
                    #:configuration-file kernel-config))
```

Any keys which are not assigned values inherit their default value from the `make-linux-libre` definition. When comparing the two snippets above, you may notice that the code comment in the first doesn't actually refer to the `#:extra-version` keyword; it is actually for `#:configuration-file`. Because of this, it is not actually easy to include a custom kernel configuration from the definition, but don't worry, there are other ways to work with what we do have.

There are two ways to create a kernel with a custom kernel configuration. The first is to provide a standard `.config` file during the build process by including an actual `.config` file as a native input to our custom kernel. The following is a snippet from the custom `'configure` phase of the `make-linux-libre` package definition:

```
(let ((build  (assoc-ref %standard-phases 'build))
      (config (assoc-ref (or native-inputs inputs) "kconfig")))

  ;; Use a custom kernel configuration file or a default
  ;; configuration file.
  (if config
      (begin
```

```
            (copy-file config ".config")
            (chmod ".config" #o666))
          (invoke "make" ,defconfig)))
```

Below is a sample kernel package. The `linux-libre` package is nothing special and can be inherited from and have its fields overridden like any other package:

```
(define-public linux-libre/E2140
  (package
    (inherit linux-libre)
    (native-inputs
     `(("kconfig" ,(local-file "E2140.config"))
      ,@(alist-delete "kconfig"
                      (package-native-inputs linux-libre))))))
```

In the same directory as the file defining `linux-libre-E2140` is a file named `E2140.config`, which is an actual kernel configuration file. The `defconfig` keyword of `make-linux-libre` is left blank here, so the only kernel configuration in the package is the one which was included in the `native-inputs` field.

The second way to create a custom kernel is to pass a new value to the `extra-options` keyword of the `make-linux-libre` procedure. The `extra-options` keyword works with another function defined right below it:

```
(define %default-extra-linux-options
  `(;; https://lists.gnu.org/archive/html/guix-devel/2014-04/msg00039.html
   ("CONFIG_DEVPTS_MULTIPLE_INSTANCES" . #true)
   ;; Modules required for initrd:
   ("CONFIG_NET_9P" . m)
   ("CONFIG_NET_9P_VIRTIO" . m)
   ("CONFIG_VIRTIO_BLK" . m)
   ("CONFIG_VIRTIO_NET" . m)
   ("CONFIG_VIRTIO_PCI" . m)
   ("CONFIG_VIRTIO_BALLOON" . m)
   ("CONFIG_VIRTIO_MMIO" . m)
   ("CONFIG_FUSE_FS" . m)
   ("CONFIG_CIFS" . m)
   ("CONFIG_9P_FS" . m)))

(define (config->string options)
  (string-join (map (match-lambda
                      ((option . 'm)
                       (string-append option "=m"))
                      ((option . #true)
                       (string-append option "=y"))
                      ((option . #false)
                       (string-append option "=n")))
                    options)
               "\n"))
```

And in the custom configure script from the ‘make-linux-libre‘ package:

```
;; Appending works even when the option wasn't in the
;; file.  The last one prevails if duplicated.
(let ((port (open-file ".config" "a"))
      (extra-configuration ,(config->string extra-options)))
  (display extra-configuration port)
  (close-port port))

(invoke "make" "oldconfig")
```

So by not providing a configuration-file the `.config` starts blank, and then we write into it the collection of flags that we want. Here's another custom kernel:

```
(define %macbook41-full-config
  (append %macbook41-config-options
          %file-systems
          %efi-support
          %emulation
          (@@ (gnu packages linux) %default-extra-linux-options)))

(define-public linux-libre-macbook41
  ;; XXX: Access the internal 'make-linux-libre' procedure, which is
  ;; private and unexported, and is liable to change in the future.
  ((@@ (gnu packages linux) make-linux-libre) (@@ (gnu packages linux) %linux-libre-ve
                      (@@ (gnu packages linux) %linux-libre-hash)
                      '("x86_64-linux")
                      #:extra-version "macbook41"
                      #:patches (@@ (gnu packages linux) %linux-libre-5.1-patches)
                      #:extra-options %macbook41-config-options))
```

In the above example `%file-systems` is a collection of flags enabling different file system support, `%efi-support` enables EFI support and `%emulation` enables a x86_64-linux machine to act in 32-bit mode also. `%default-extra-linux-options` are the ones quoted above, which had to be added in since they were replaced in the `extra-options` keyword.

This all sounds like it should be doable, but how does one even know which modules are required for a particular system? Two places that can be helpful in trying to answer this question is the Gentoo Handbook (`https://wiki.gentoo.org/wiki/Handbook:AMD64/Installation/Kernel`) and the documentation from the kernel itself (`https://www.kernel.org/doc/html/latest/admin-guide/README.html?highlight=localmodconfig`). From the kernel documentation, it seems that `make localmodconfig` is the command we want.

In order to actually run `make localmodconfig` we first need to get and unpack the kernel source code:

```
shell
tar xf $(guix build linux-libre --source)
```

Once inside the directory containing the source code run `touch .config` to create an initial, empty `.config` to start with. `make localmodconfig` works by seeing what you already have in `.config` and letting you know what you're missing. If the file is blank then you're missing everything. The next step is to run:

```
shell
guix environment linux-libre -- make localmodconfig
```

and note the output. Do note that the `.config` file is still empty. The output generally contains two types of warnings. The first start with "WARNING" and can actually be ignored in our case. The second read:

```
shell
module pcspkr did not have configs CONFIG_INPUT_PCSPKR
```

For each of these lines, copy the `CONFIG_XXXX_XXXX` portion into the `.config` in the directory, and append `=m`, so in the end it looks like this:

```
shell
CONFIG_INPUT_PCSPKR=m
CONFIG_VIRTIO=m
```

After copying all the configuration options, run `make localmodconfig` again to make sure that you don't have any output starting with "module". After all of these machine specific modules there are a couple more left that are also needed. `CONFIG_MODULES` is necessary so that you can build and load modules separately and not have everything built into the kernel. `CONFIG_BLK_DEV_SD` is required for reading from hard drives. It is possible that there are other modules which you will need.

This post does not aim to be a guide to configuring your own kernel however, so if you do decide to build a custom kernel you'll have to seek out other guides to create a kernel which is just right for your needs.

The second way to setup the kernel configuration makes more use of Guix's features and allows you to share configuration segments between different kernels. For example, all machines using EFI to boot have a number of EFI configuration flags that they need. It is likely that all the kernels will share a list of file systems to support. By using variables it is easier to see at a glance what features are enabled and to make sure you don't have features in one kernel but missing in another.

Left undiscussed however, is Guix's initrd and its customization. It is likely that you'll need to modify the initrd on a machine using a custom kernel, since certain modules which are expected to be built may not be available for inclusion into the initrd.

3.3 Guix System Image API

Historically, Guix System is centered around an `operating-system` structure. This structure contains various fields ranging from the bootloader and kernel declaration to the services to install.

Depending on the target machine, that can go from a standard `x86_64` machine to a small ARM single board computer such as the Pine64, the image constraints can vary a lot. The hardware manufacturers will impose different image formats with various partition sizes and offsets.

To create images suitable for all those machines, a new abstraction is necessary: that's the goal of the `image` record. This record contains all the required information to be transformed into a standalone image, that can be directly booted on any target machine.

```
(define-record-type* <image>
  image make-image
```

```
image?
(name               image-name ;symbol
                    (default #f))
(format             image-format) ;symbol
(target             image-target
                    (default #f))
(size               image-size  ;size in bytes as integer
                    (default 'guess))
(operating-system   image-operating-system  ;<operating-system>
                    (default #f))
(partitions         image-partitions ;list of <partition>
                    (default '()))
(compression?       image-compression? ;boolean
                    (default #t))
(volatile-root?     image-volatile-root? ;boolean
                    (default #t))
(substitutable?     image-substitutable? ;boolean
                    (default #t)))
```

This record contains the operating-system to instantiate. The `format` field defines the image type and can be `efi-raw`, `qcow2` or `iso9660` for instance. In the future, it could be extended to `docker` or other image types.

A new directory in the Guix sources is dedicated to images definition. For now there are four files:

- `gnu/system/images/hurd.scm`
- `gnu/system/images/pine64.scm`
- `gnu/system/images/novena.scm`
- `gnu/system/images/pinebook-pro.scm`

Let's have a look to `pine64.scm`. It contains the `pine64-barebones-os` variable which is a minimal definition of an operating-system dedicated to the **Pine A64 LTS** board.

```
(define pine64-barebones-os
  (operating-system
   (host-name "vignemale")
   (timezone "Europe/Paris")
   (locale "en_US.utf8")
   (bootloader (bootloader-configuration
                (bootloader u-boot-pine64-lts-bootloader)
                (targets '("/dev/vda"))))
   (initrd-modules '())
   (kernel linux-libre-arm64-generic)
   (file-systems (cons (file-system
                         (device (file-system-label "my-root"))
                         (mount-point "/")
                         (type "ext4"))
                        %base-file-systems))
   (services (cons (service agetty-service-type
```

```
                                  (agetty-configuration
                                   (extra-options '("-L")) ; no carrier detect
                                   (baud-rate "115200")
                                   (term "vt100")
                                   (tty "ttyS0")))
                         %base-services))))
```

The `kernel` and `bootloader` fields are pointing to packages dedicated to this board.

Right below, the `pine64-image-type` variable is also defined.

```
(define pine64-image-type
  (image-type
   (name 'pine64-raw)
   (constructor (cut image-with-os arm64-disk-image <>))))
```

It's using a record we haven't talked about yet, the `image-type` record, defined this way:

```
(define-record-type* <image-type>
  image-type make-image-type
  image-type?
  (name           image-type-name) ;symbol
  (constructor    image-type-constructor)) ;<operating-system> -> <image>
```

The main purpose of this record is to associate a name to a procedure transforming an `operating-system` to an image. To understand why it is necessary, let's have a look to the command producing an image from an `operating-system` configuration file:

```
guix system image my-os.scm
```

This command expects an `operating-system` configuration but how should we indicate that we want an image targeting a Pine64 board? We need to provide an extra information, the `image-type`, by passing the `--image-type` or `-t` flag, this way:

```
guix system image --image-type=pine64-raw my-os.scm
```

This `image-type` parameter points to the `pine64-image-type` defined above. Hence, the `operating-system` declared in `my-os.scm` will be applied the `(cut image-with-os arm64-disk-image <>)` procedure to turn it into an image.

The resulting image looks like:

```
(image
 (format 'disk-image)
 (target "aarch64-linux-gnu")
 (operating-system my-os)
 (partitions
  (list (partition
         (inherit root-partition)
         (offset root-offset)))))
```

which is the aggregation of the `operating-system` defined in `my-os.scm` to the `arm64-disk-image` record.

But enough Scheme madness. What does this image API bring to the Guix user?

One can run:

```
mathieu@cervin:~$ guix system --list-image-types
```

```
The available image types are:

   - pinebook-pro-raw
   - pine64-raw
   - novena-raw
   - hurd-raw
   - hurd-qcow2
   - qcow2
   - uncompressed-iso9660
   - efi-raw
   - arm64-raw
   - arm32-raw
   - iso9660
```

and by writing an `operating-system` file based on `pine64-barebones-os`, you can customize your image to your preferences in a file (`my-pine-os.scm`) like this:

```
(use-modules (gnu services linux)
             (gnu system images pine64))

(let ((base-os pine64-barebones-os))
  (operating-system
    (inherit base-os)
    (timezone "America/Indiana/Indianapolis")
    (services
     (cons
      (service earlyoom-service-type
               (earlyoom-configuration
                (prefer-regexp "icecat|chromium")))
      (operating-system-user-services base-os)))))
```

run:

```
guix system image --image-type=pine64-raw my-pine-os.scm
```

or,

```
guix system image --image-type=hurd-raw my-hurd-os.scm
```

to get an image that can be written directly to a hard drive and booted from.

Without changing anything to `my-hurd-os.scm`, calling:

```
guix system image --image-type=hurd-qcow2 my-hurd-os.scm
```

will instead produce a Hurd QEMU image.

3.4 Connecting to Wireguard VPN

To connect to a Wireguard VPN server you need the kernel module to be loaded in memory and a package providing networking tools that support it (e.g. `wireguard-tools` or `network-manager`).

Here is a configuration example for Linux-Libre < 5.6, where the module is out of tree and need to be loaded manually—following revisions of the kernel have it built-in and so don't need such configuration:

```
(use-modules (gnu))
(use-service-modules desktop)
(use-package-modules vpn)

(operating-system
  ;; ...
  (services (cons (simple-service 'wireguard-module
                                  kernel-module-loader-service-type
                                  '("wireguard"))
                  %desktop-services))
  (packages (cons wireguard-tools %base-packages))
  (kernel-loadable-modules (list wireguard-linux-compat)))
```

After reconfiguring and restarting your system you can either use Wireguard tools or NetworkManager to connect to a VPN server.

3.4.1 Using Wireguard tools

To test your Wireguard setup it is convenient to use `wg-quick`. Just give it a configuration file `wg-quick up ./wg0.conf`; or put that file in `/etc/wireguard` and run `wg-quick up wg0` instead.

> **Note:** Be warned that the author described this command as a: "[. . .] very quick and dirty bash script [. . .]".

3.4.2 Using NetworkManager

Thanks to NetworkManager support for Wireguard we can connect to our VPN using `nmcli` command. Up to this point this guide assumes that you're using Network Manager service provided by `%desktop-services`. Ortherwise you need to adjust your services list to load `network-manager-service-type` and reconfigure your Guix system.

To import your VPN configuration execute nmcli import command:

```
shell
# nmcli connection import type wireguard file wg0.conf
Connection 'wg0' (edbee261-aa5a-42db-b032-6c7757c60fde) successfully added
```

This will create a configuration file in `/etc/NetworkManager/wg0.nmconnection`. Next connect to the Wireguard server:

```
shell
$ nmcli connection up wg0
Connection successfully activated (D-Bus active path: /org/freedesktop/NetworkManager/
```

By default NetworkManager will connect automatically on system boot. To change that behaviour you need to edit your config:

```
shell
# nmcli connection modify wg0 connection.autoconnect no
```

For more specific information about NetworkManager and wireguard see this post by thaller (`https://blogs.gnome.org/thaller/2019/03/15/wireguard-in-networkmanager/`).

3.5 Customizing a Window Manager

3.5.1 StumpWM

You could install StumpWM with a Guix system by adding `stumpwm` and optionally `‘(,stumpwm "lib")` packages to a system configuration file, e.g. `/etc/config.scm`.

An example configuration can look like this:

```
(use-modules (gnu))
(use-package-modules wm)

(operating-system
  ;; ...
  (packages (append (list sbcl stumpwm ‘(,stumpwm "lib"))
                    %base-packages)))
```

By default StumpWM uses X11 fonts, which could be small or pixelated on your system. You could fix this by installing StumpWM contrib Lisp module `sbcl-ttf-fonts`, adding it to Guix system packages:

```
(use-modules (gnu))
(use-package-modules fonts wm)

(operating-system
  ;; ...
  (packages (append (list sbcl stumpwm ‘(,stumpwm "lib"))
                    sbcl-ttf-fonts font-dejavu %base-packages)))
```

Then you need to add the following code to a StumpWM configuration file `~/.stumpwm.d/init.lisp`:

```
(require :ttf-fonts)
(setf xft:*font-dirs* ’("/run/current-system/profile/share/fonts/"))
(setf clx-truetype:+font-cache-filename+ (concat (getenv "HOME") "/.fonts/font-cache.s
(xft:cache-fonts)
(set-font (make-instance ’xft:font :family "DejaVu Sans Mono" :subfamily "Book" :size
```

3.5.2 Session lock

Depending on your environment, locking the screen of your session might come built in or it might be something you have to set up yourself. If you use a desktop environment like GNOME or KDE, it’s usually built in. If you use a plain window manager like StumpWM or EXWM, you might have to set it up yourself.

3.5.2.1 Xorg

If you use Xorg, you can use the utility xss-lock (`https://www.mankier.com/1/xss-lock`) to lock the screen of your session. xss-lock is triggered by DPMS which since Xorg 1.8 is auto-detected and enabled if ACPI is also enabled at kernel runtime.

To use xss-lock, you can simple execute it and put it into the background before you start your window manager from e.g. your `~/.xsession`:

```
xss-lock -- slock &
```

```
exec stumpwm
```

In this example, xss-lock uses `slock` to do the actual locking of the screen when it determines it's appropriate, like when you suspend your device.

For slock to be allowed to be a screen locker for the graphical session, it needs to be made setuid-root so it can authenticate users, and it needs a PAM service. This can be achieved by adding the following service to your `config.scm`:

```
(screen-locker-service slock)
```

If you manually lock your screen, e.g. by directly calling slock when you want to lock your screen but not suspend it, it's a good idea to notify xss-lock about this so no confusion occurs. This can be done by executing `xset s activate` immediately before you execute slock.

3.6 Running Guix on a Linode Server

To run Guix on a server hosted by Linode (`https://www.linode.com`), start with a recommended Debian server. We recommend using the default distro as a way to bootstrap Guix. Create your SSH keys.

```
ssh-keygen
```

Be sure to add your SSH key for easy login to the remote server. This is trivially done via Linode's graphical interface for adding SSH keys. Go to your profile and click add SSH Key. Copy into it the output of:

```
cat ~/.ssh/<username>_rsa.pub
```

Power the Linode down.

In the Linode's Storage tab, resize the Debian disk to be smaller. 30 GB free space is recommended. Then click "Add a disk", and fill out the form with the following:

- Label: "Guix"
- Filesystem: ext4
- Set it to the remaining size

In the Configurations tab, press "Edit" on the default Debian profile. Under "Block Device Assignment" click "Add a Device". It should be `/dev/sdc` and you can select the "Guix" disk. Save Changes.

Now "Add a Configuration", with the following:

- Label: Guix
- Kernel:GRUB 2 (it's at the bottom! This step is **IMPORTANT!**)
- Block device assignment:
- `/dev/sda`: Guix
- `/dev/sdb`: swap
- Root device: `/dev/sda`
- Turn off all the filesystem/boot helpers

Now power it back up, booting with the Debian configuration. Once it's running, ssh to your server via `ssh root@`*`<your-server-IP-here>`*. (You can find your server IP address in your Linode Summary section.) Now you can run the "install guix from see Section "Binary Installation" in *GNU Guix*" steps:

```
sudo apt-get install gpg
wget https://sv.gnu.org/people/viewgpg.php?user_id=15145 -qO - | gpg --import -
wget https://git.savannah.gnu.org/cgit/guix.git/plain/etc/guix-install.sh
chmod +x guix-install.sh
./guix-install.sh
guix pull
```

Now it's time to write out a config for the server. The key information is below. Save the resulting file as `guix-config.scm`.

```
(use-modules (gnu)
             (guix modules))
(use-service-modules networking
                     ssh)
(use-package-modules admin
                     certs
                     package-management
                     ssh
                     tls)

(operating-system
  (host-name "my-server")
  (timezone "America/New_York")
  (locale "en_US.UTF-8")
  ;; This goofy code will generate the grub.cfg
  ;; without installing the grub bootloader on disk.
  (bootloader (bootloader-configuration
               (bootloader
                (bootloader
                 (inherit grub-bootloader)
                 (installer #~(const #true))))))
  (file-systems (cons (file-system
                        (device "/dev/sda")
                        (mount-point "/")
                        (type "ext4"))
                      %base-file-systems))


  (swap-devices (list "/dev/sdb"))


  (initrd-modules (cons "virtio_scsi"    ; Needed to find the disk
                        %base-initrd-modules))

  (users (cons (user-account
                (name "janedoe")
```

```
                 (group "users")
                 ;; Adding the account to the "wheel" group
                 ;; makes it a sudoer.
                 (supplementary-groups '("wheel"))
                 (home-directory "/home/janedoe"))
                %base-user-accounts))

  (packages (cons* nss-certs            ;for HTTPS access
                   openssh-sans-x
                   %base-packages))

  (services (cons*
              (service dhcp-client-service-type)
              (service openssh-service-type
                       (openssh-configuration
                        (openssh openssh-sans-x)
                        (password-authentication? #false)
                        (authorized-keys
                         `(("janedoe" ,(local-file "janedoe_rsa.pub"))
                           ("root" ,(local-file "janedoe_rsa.pub"))))))
              %base-services)))
```

Replace the following fields in the above configuration:

```
(host-name "my-server")       ; replace with your server name
; if you chose a linode server outside the U.S., then
; use tzselect to find a correct timezone string
(timezone "America/New_York") ; if needed replace timezone
(name "janedoe")              ; replace with your username
("janedoe" ,(local-file "janedoe_rsa.pub")) ; replace with your ssh key
("root" ,(local-file "janedoe_rsa.pub")) ; replace with your ssh key
```

The last line in the above example lets you log into the server as root and set the initial root password (see the note at the end of this recipe about root login). After you have done this, you may delete that line from your configuration and reconfigure to prevent root login.

Copy your ssh public key (eg: `~/.ssh/id_rsa.pub`) as `<your-username-here>_rsa.pub` and put `guix-config.scm` in the same directory. In a new terminal run these commands.

```
sftp root@<remote server ip address>
put /path/to/files/<username>_rsa.pub .
put /path/to/files/guix-config.scm .
```

In your first terminal, mount the guix drive:

```
mkdir /mnt/guix
mount /dev/sdc /mnt/guix
```

Due to the way we set up the bootloader section of the guix-config.scm, only the grub configuration file will be installed. So, we need to copy over some of the other GRUB stuff already installed on the Debian system:

```
mkdir -p /mnt/guix/boot/grub
cp -r /boot/grub/* /mnt/guix/boot/grub/
```

Now initialize the Guix installation:

```
guix system init guix-config.scm /mnt/guix
```

Ok, power it down! Now from the Linode console, select boot and select "Guix".

Once it boots, you should be able to log in via SSH! (The server config will have changed though.) You may encounter an error like:

```
$ ssh root@<server ip address>
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@     WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
Someone could be eavesdropping on you right now (man-in-the-middle attack)!
It is also possible that a host key has just been changed.
The fingerprint for the ECDSA key sent by the remote host is
SHA256:0B+wp33w57AnKQuHCvQP0+ZdKaqYrI/kyU7CfVbS7R4.
Please contact your system administrator.
Add correct host key in /home/joshua/.ssh/known_hosts to get rid of this message.
Offending ECDSA key in /home/joshua/.ssh/known_hosts:3
ECDSA host key for 198.58.98.76 has changed and you have requested strict checking.
Host key verification failed.
```

Either delete `~/.ssh/known_hosts` file, or delete the offending line starting with your server IP address.

Be sure to set your password and root's password.

```
ssh root@<remote ip address>
passwd  ; for the root password
passwd <username> ; for the user password
```

You may not be able to run the above commands at this point. If you have issues remotely logging into your linode box via SSH, then you may still need to set your root and user password initially by clicking on the "Launch Console" option in your linode. Choose the "Glish" instead of "Weblish". Now you should be able to ssh into the machine.

Hooray! At this point you can shut down the server, delete the Debian disk, and resize the Guix to the rest of the size. Congratulations!

By the way, if you save it as a disk image right at this point, you'll have an easy time spinning up new Guix images! You may need to down-size the Guix image to 6144MB, to save it as an image. Then you can resize it again to the max size.

3.7 Setting up a bind mount

To bind mount a file system, one must first set up some definitions before the `operating-system` section of the system definition. In this example we will bind mount a folder from a spinning disk drive to `/tmp`, to save wear and tear on the primary SSD, without dedicating an entire partition to be mounted as `/tmp`.

First, the source drive that hosts the folder we wish to bind mount should be defined, so that the bind mount can depend on it.

```
(define source-drive ;; "source-drive" can be named anything you want.
   (file-system
    (device (uuid "UUID goes here"))
```

```
        (mount-point "/path-to-spinning-disk-goes-here")
        (type "ext4"))) ;; Make sure to set this to the appropriate type for your drive.
```

The source folder must also be defined, so that guix will know it's not a regular block device, but a folder.

```
(define (%source-directory) "/path-to-spinning-disk-goes-here/tmp") ;; "source-directo
```

Finally, inside the `file-systems` definition, we must add the mount itself.

```
(file-systems (cons*

                ...<other drives omitted for clarity>...

                source-drive ;; Must match the name you gave the source drive in the e

                (file-system
                 (device (%source-directory)) ;; Make sure "source-directory" matches
                 (mount-point "/tmp")
                 (type "none") ;; We are mounting a folder, not a partition, so this t
                 (flags '(bind-mount))
                 (dependencies (list source-drive)) ;; Ensure "source-drive" matches w
                 )

                 ...<other drives omitted for clarity>...

                ))
```

3.8 Getting substitutes from Tor

Guix daemon can use a HTTP proxy to get substitutes, here we are configuring it to get them via Tor.

> **Warning:** *Not all* Guix daemon's traffic will go through Tor! Only HTTP/HTTPS will get proxied; FTP, Git protocol, SSH, etc connections will still go through the clearnet. Again, this configuration isn't foolproof some of your traffic won't get routed by Tor at all. Use it at your own risk.
>
> Also note that the procedure described here applies only to package substitution. When you update your guix distribution with `guix pull`, you still need to use `torsocks` if you want to route the connection to guix's git repository servers through Tor.

Guix's substitute server is available as a Onion service, if you want to use it to get your substitutes through Tor configure your system as follow:

```
(use-modules (gnu))
(use-service-module base networking)

(operating-system
  ...
  (services
    (cons
```

```
(service tor-service-type
        (tor-configuration
          (config-file (plain-file "tor-config"
                                   "HTTPTunnelPort 127.0.0.1:9250"))))
(modify-services %base-services
  (guix-service-type
    config => (guix-configuration
                (inherit config)
                ;; ci.guix.gnu.org's Onion service
                (substitute-urls "https://bp7o7ckwlewr4slm.onion")
                (http-proxy "http://localhost:9250")))))))
```

This will keep a tor process running that provides a HTTP CONNECT tunnel which will be used by `guix-daemon`. The daemon can use other protocols than HTTP(S) to get remote resources, request using those protocols won't go through Tor since we are only setting a HTTP tunnel here. Note that `substitutes-urls` is using HTTPS and not HTTP or it won't work, that's a limitation of Tor's tunnel; you may want to use `privoxy` instead to avoid such limitations.

If you don't want to always get substitutes through Tor but using it just some of the times, then skip the `guix-configuration`. When you want to get a substitute from the Tor tunnel run:

```
sudo herd set-http-proxy guix-daemon http://localhost:9250
guix build --substitute-urls=https://bp7o7ckwlewr4slm.onion ...
```

3.9 Setting up NGINX with Lua

NGINX could be extended with Lua scripts.

Guix provides NGINX service with ability to load Lua module and specific Lua packages, and reply to requests by evaluating Lua scripts.

The following example demonstrates system definition with configuration to evaluate `index.lua` Lua script on HTTP request to `http://localhost/hello` endpoint:

```
local shell = require "resty.shell"

local stdin = ""
local timeout = 1000  -- ms
local max_size = 4096  -- byte

local ok, stdout, stderr, reason, status =
   shell.run([[/run/current-system/profile/bin/ls /tmp]], stdin, timeout, max_size)

ngx.say(stdout)
```

```
(use-modules (gnu))
(use-service-modules #;... web)
(use-package-modules #;... lua)
(operating-system
  ;; ...
  (services
```

```
;; ...
(service nginx-service-type
         (nginx-configuration
          (modules
           (list
            (file-append nginx-lua-module "/etc/nginx/modules/ngx_http_lua_module.s
          (lua-package-path (list lua-resty-core
                                  lua-resty-lrucache
                                  lua-resty-signal
                                  lua-tablepool
                                  lua-resty-shell))
          (lua-package-cpath (list lua-resty-signal))
          (server-blocks
           (list (nginx-server-configuration
                  (server-name '("localhost"))
                  (listen '("80"))
                  (root "/etc")
                  (locations (list
                              (nginx-location-configuration
                               (uri "/hello")
                               (body (list #~(format #f "content_by_lua_file ~s;"
                                                     #$(local-file "index.lua")))))
```

4 Advanced package management

Guix is a functional package manager that offers many features beyond what more traditional package managers can do. To the uninitiated, those features might not have obvious use cases at first. The purpose of this chapter is to demonstrate some advanced package management concepts.

see Section "Package Management" in *GNU Guix Reference Manual* for a complete reference.

4.1 Guix Profiles in Practice

Guix provides a very useful feature that may be quite foreign to newcomers: *profiles*. They are a way to group package installations together and all users on the same system are free to use as many profiles as they want.

Whether you're a developer or not, you may find that multiple profiles bring you great power and flexibility. While they shift the paradigm somewhat compared to *traditional package managers*, they are very convenient to use once you've understood how to set them up.

If you are familiar with Python's '`virtualenv`', you can think of a profile as a kind of universal '`virtualenv`' that can hold any kind of software whatsoever, not just Python software. Furthermore, profiles are self-sufficient: they capture all the runtime dependencies which guarantees that all programs within a profile will always work at any point in time.

Multiple profiles have many benefits:

- Clean semantic separation of the various packages a user needs for different contexts.
- Multiple profiles can be made available into the environment either on login or within a dedicated shell.
- Profiles can be loaded on demand. For instance, the user can use multiple shells, each of them running different profiles.
- Isolation: Programs from one profile will not use programs from the other, and the user can even install different versions of the same programs to the two profiles without conflict.
- Deduplication: Profiles share dependencies that happens to be the exact same. This makes multiple profiles storage-efficient.
- Reproducible: when used with declarative manifests, a profile can be fully specified by the Guix commit that was active when it was set up. This means that the exact same profile can be set up anywhere and anytime (`https://guix.gnu.org/blog/2018/multi-dimensional-transactions-and-rollbacks-oh-my/`), with just the commit information. See the section on Section 4.1.5 [Reproducible profiles], page 43.
- Easier upgrades and maintenance: Multiple profiles make it easy to keep package listings at hand and make upgrades completely frictionless.

Concretely, here follows some typical profiles:

- The dependencies of a project you are working on.
- Your favourite programming language libraries.

- Laptop-specific programs (like '`powertop`') that you don't need on a desktop.
- TEXlive (this one can be really useful when you need to install just one package for this one document you've just received over email).
- Games.

Let's dive in the set up!

4.1.1 Basic setup with manifests

A Guix profile can be set up *via* a so-called *manifest specification* that looks like this:

```
(specifications->manifest
  '("package-1"
    ;; Version 1.3 of package-2.
    "package-2@1.3"
    ;; The "lib" output of package-3.
    "package-3:lib"
    ; ...
    "package-N"))
```

see Section "Invoking guix package" in *GNU Guix Reference Manual*, for the syntax details.

We can create a manifest specification per profile and install them this way:

```
GUIX_EXTRA_PROFILES=$HOME/.guix-extra-profiles
mkdir -p "$GUIX_EXTRA_PROFILES"/my-project # if it does not exist yet
guix package --manifest=/path/to/guix-my-project-manifest.scm --profile="$GUIX_EXTRA_P
```

Here we set an arbitrary variable '`GUIX_EXTRA_PROFILES`' to point to the directory where we will store our profiles in the rest of this article.

Placing all your profiles in a single directory, with each profile getting its own sub-directory, is somewhat cleaner. This way, each sub-directory will contain all the symlinks for precisely one profile. Besides, "looping over profiles" becomes obvious from any programming language (e.g. a shell script) by simply looping over the sub-directories of '`$GUIX_EXTRA_PROFILES`'.

Note that it's also possible to loop over the output of

```
guix package --list-profiles
```

although you'll probably have to filter out `~/.config/guix/current`.

To enable all profiles on login, add this to your `~/.bash_profile` (or similar):

```
for i in $GUIX_EXTRA_PROFILES/*; do
  profile=$i/$(basename "$i")
  if [ -f "$profile"/etc/profile ]; then
    GUIX_PROFILE="$profile"
    . "$GUIX_PROFILE"/etc/profile
  fi
  unset profile
done
```

Note to Guix System users: the above reflects how your default profile `~/.guix-profile` is activated from `/etc/profile`, that latter being loaded by `~/.bashrc` by default.

You can obviously choose to only enable a subset of them:

```
for i in "$GUIX_EXTRA_PROFILES"/my-project-1 "$GUIX_EXTRA_PROFILES"/my-project-2; do
  profile=$i/$(basename "$i")
  if [ -f "$profile"/etc/profile ]; then
    GUIX_PROFILE="$profile"
    . "$GUIX_PROFILE"/etc/profile
  fi
  unset profile
done
```

When a profile is off, it's straightforward to enable it for an individual shell without "polluting" the rest of the user session:

```
GUIX_PROFILE="path/to/my-project" ; . "$GUIX_PROFILE"/etc/profile
```

The key to enabling a profile is to *source* its '`etc/profile`' file. This file contains shell code that exports the right environment variables necessary to activate the software contained in the profile. It is built automatically by Guix and meant to be sourced. It contains the same variables you would get if you ran:

```
guix package --search-paths=prefix --profile=$my_profile"
```

Once again, see (see Section "Invoking guix package" in *GNU Guix Reference Manual*) for the command line options.

To upgrade a profile, simply install the manifest again:

```
guix package -m /path/to/guix-my-project-manifest.scm -p "$GUIX_EXTRA_PROFILES"/my-pro
```

To upgrade all profiles, it's easy enough to loop over them. For instance, assuming your manifest specifications are stored in `~/.guix-manifests/guix-$profile-manifest.scm`, with '`$profile`' being the name of the profile (e.g. "project1"), you could do the following in Bourne shell:

```
for profile in "$GUIX_EXTRA_PROFILES"/*; do
  guix package --profile="$profile" --manifest="$HOME/.guix-manifests/guix-$profile-ma
done
```

Each profile has its own generations:

```
guix package -p "$GUIX_EXTRA_PROFILES"/my-project/my-project --list-generations
```

You can roll-back to any generation of a given profile:

```
guix package -p "$GUIX_EXTRA_PROFILES"/my-project/my-project --switch-generations=17
```

Finally, if you want to switch to a profile without inheriting from the current environment, you can activate it from an empty shell:

```
env -i $(which bash) --login --noprofile --norc
. my-project/etc/profile
```

4.1.2 Required packages

Activating a profile essentially boils down to exporting a bunch of environmental variables. This is the role of the '`etc/profile`' within the profile.

Note: Only the environmental variables of the packages that consume them will be set.

For instance, 'MANPATH' won't be set if there is no consumer application for man pages within the profile. So if you need to transparently access man pages once the profile is loaded, you've got two options:

- Either export the variable manually, e.g.

```
export MANPATH=/path/to/profile${MANPATH:+:}$MANPATH
```

- Or include 'man-db' to the profile manifest.

The same is true for 'INFOPATH' (you can install 'info-reader'), 'PKG_CONFIG_PATH' (install 'pkg-config'), etc.

4.1.3 Default profile

What about the default profile that Guix keeps in `~/.guix-profile`?

You can assign it the role you want. Typically you would install the manifest of the packages you want to use all the time.

Alternatively, you could keep it "manifest-less" for throw-away packages that you would just use for a couple of days. This way makes it convenient to run

```
guix install package-foo
guix upgrade package-bar
```

without having to specify the path to a profile.

4.1.4 The benefits of manifests

Manifests are a convenient way to keep your package lists around and, say, to synchronize them across multiple machines using a version control system.

A common complaint about manifests is that they can be slow to install when they contain large number of packages. This is especially cumbersome when you just want get an upgrade for one package within a big manifest.

This is one more reason to use multiple profiles, which happen to be just perfect to break down manifests into multiple sets of semantically connected packages. Using multiple, small profiles provides more flexibility and usability.

Manifests come with multiple benefits. In particular, they ease maintenance:

- When a profile is set up from a manifest, the manifest itself is self-sufficient to keep a "package listing" around and reinstall the profile later or on a different system. For ad-hoc profiles, we would need to generate a manifest specification manually and maintain the package versions for the packages that don't use the default version.
- `guix package --upgrade` always tries to update the packages that have propagated inputs, even if there is nothing to do. Guix manifests remove this problem.
- When partially upgrading a profile, conflicts may arise (due to diverging dependencies between the updated and the non-updated packages) and they can be annoying to resolve manually. Manifests remove this problem altogether since all packages are always upgraded at once.
- As mentioned above, manifests allow for reproducible profiles, while the imperative `guix install`, `guix upgrade`, etc. do not, since they produce different profiles every time even when they hold the same packages. See the related discussion on the matter (https://issues.guix.gnu.org/issue/33285).

- Manifest specifications are usable by other '`guix`' commands. For example, you can run `guix weather -m manifest.scm` to see how many substitutes are available, which can help you decide whether you want to try upgrading today or wait a while. Another example: you can run `guix pack -m manifest.scm` to create a pack containing all the packages in the manifest (and their transitive references).
- Finally, manifests have a Scheme representation, the '`<manifest>`' record type. They can be manipulated in Scheme and passed to the various Guix APIs (`https://en.wikipedia.org/wiki/Api`).

It's important to understand that while manifests can be used to declare profiles, they are not strictly equivalent: profiles have the side effect that they "pin" packages in the store, which prevents them from being garbage-collected (see Section "Invoking guix gc" in *GNU Guix Reference Manual*) and ensures that they will still be available at any point in the future.

Let's take an example:

1. We have an environment for hacking on a project for which there isn't a Guix package yet. We build the environment using a manifest, and then run `guix environment -m manifest.scm`. So far so good.
2. Many weeks pass and we have run a couple of `guix pull` in the mean time. Maybe a dependency from our manifest has been updated; or we may have run `guix gc` and some packages needed by our manifest have been garbage-collected.
3. Eventually, we set to work on that project again, so we run `guix environment -m manifest.scm`. But now we have to wait for Guix to build and install stuff!

Ideally, we could spare the rebuild time. And indeed we can, all we need is to install the manifest to a profile and use `GUIX_PROFILE=/the/profile; . "$GUIX_PROFILE"/etc/profile` as explained above: this guarantees that our hacking environment will be available at all times.

Security warning: While keeping old profiles around can be convenient, keep in mind that outdated packages may not have received the latest security fixes.

4.1.5 Reproducible profiles

To reproduce a profile bit-for-bit, we need two pieces of information:

- a manifest,
- a Guix channel specification.

Indeed, manifests alone might not be enough: different Guix versions (or different channels) can produce different outputs for a given manifest.

You can output the Guix channel specification with '`guix describe --format=channels`'. Save this to a file, say '`channel-specs.scm`'.

On another computer, you can use the channel specification file and the manifest to reproduce the exact same profile:

```
GUIX_EXTRA_PROFILES=$HOME/.guix-extra-profiles
GUIX_EXTRA=$HOME/.guix-extra

mkdir "$GUIX_EXTRA"/my-project
```

```
guix pull --channels=channel-specs.scm --profile "$GUIX_EXTRA/my-project/guix"

mkdir -p "$GUIX_EXTRA_PROFILES/my-project"
"$GUIX_EXTRA"/my-project/guix/bin/guix package --manifest=/path/to/guix-my-project-man
```

It's safe to delete the Guix channel profile you've just installed with the channel specification, the project profile does not depend on it.

5 Environment management

Guix provides multiple tools to manage environment. This chapter demonstrate such utilities.

5.1 Guix environment via direnv

Guix provides a ‘`direnv`’ package, which could extend shell after directory change. This tool could be used to prepare a pure Guix environment.

The following example provides a shell function for `~/.direnvrc` file, which could be used from Guix Git repository in `~/src/guix/.envrc` file to setup a build environment similar to described in see Section “Building from Git” in *GNU Guix Reference Manual*.

Create a `~/.direnvrc` with a Bash code:

```
# Thanks <https://github.com/direnv/direnv/issues/73#issuecomment-152284914>
export_function()
{
  local name=$1
  local alias_dir=$PWD/.direnv/aliases
  mkdir -p "$alias_dir"
  PATH_add "$alias_dir"
  local target="$alias_dir/$name"
  if declare -f "$name" >/dev/null; then
    echo "#!$SHELL" > "$target"
    declare -f "$name" >> "$target" 2>/dev/null
    # Notice that we add shell variables to the function trigger.
    echo "$name \$*" >> "$target"
    chmod +x "$target"
  fi
}

use_guix()
{
    # Set GitHub token.
    export GUIX_GITHUB_TOKEN="xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"

    # Unset ’GUIX_PACKAGE_PATH’.
    export GUIX_PACKAGE_PATH=""

    # Recreate a garbage collector root.
    gcroots="$HOME/.config/guix/gcroots"
    mkdir -p "$gcroots"
    gcroot="$gcroots/guix"
    if [ -L "$gcroot" ]
    then
        rm -v "$gcroot"
    fi
```

```
# Miscellaneous packages.
PACKAGES_MAINTENANCE=(
    direnv
    git
    git:send-email
    git-cal
    gnupg
    guile-colorized
    guile-readline
    less
    ncurses
    openssh
    xdot
)

# Environment packages.
PACKAGES=(help2man guile-sqlite3 guile-gcrypt)

# Thanks <https://lists.gnu.org/archive/html/guix-devel/2016-09/msg00859.html>
eval "$(guix environment --search-paths --root="$gcroot" --pure guix --ad-hoc ${PA

# Predefine configure flags.
configure()
{
    ./configure --localstatedir=/var --prefix=
}
export_function configure

# Run make and optionally build something.
build()
{
    make -j 2
    if [ $# -gt 0 ]
    then
        ./pre-inst-env guix build "$@"
    fi
}
export_function build

# Predefine push Git command.
push()
{
    git push --set-upstream origin
}
export_function push
```

```
    clear                       # Clean up the screen.
    git-cal --author='Your Name' # Show contributions calendar.

    # Show commands help.
    echo "
build          build a package or just a project if no argument provided
configure      run ./configure with predefined parameters
push           push to upstream Git repository
"
}
```

Every project containing `.envrc` with a string `use guix` will have predefined environment variables and procedures.

Run `direnv allow` to setup the environment for the first time.

6 Acknowledgments

Guix is based on the Nix package manager (`https://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!

This document includes adapted sections from articles that have previously been published on the Guix blog at `https://guix.gnu.org/blog`.

Appendix A GNU Free Documentation License

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 noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

   This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

   We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

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 `https://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