Livre de recettes de GNU Guix

Didacticiels et exemples d'utilisation du gestionnaire de paquets GNU Guix

Les développeurs de GNU Guix

Table des matières

1 Didacticiels pour Scheme

GNU Guix est écrit dans le langage de programmation Scheme. Nombre de ses fonctionnalités peuvent être consultées et manipulées par programmation. Vous pouvez utiliser Scheme entre autres pour générer des définitions de paquets, pour les modifier, pour les compiler ou pour déployer des systèmes d'exploitation entiers.

Connaître les bases de la programmation en Scheme vous permettra d'utiliser de nombreuses fonctionnalités avancées de Guix — et vous n'avez pas besoin d'être un·e programmeur·euse chevronné·e !

C'est parti !

1.1 Cours accéléré du langage Scheme

Guix utilise l'implémentation Guile du langage Scheme. Pour commencer à jouer avec le langage, installez-le avec `guix install guile` et démarrer une *BLÉA* (*REPL* en anglais), une *boucle de lecture, évaluation, affichage* (`https://en.wikipedia.org/wiki/Read%E2%80%93eval%E2%80%93print_loop`), en lançant `guile` sur la ligne de commande.

Vous pouvez également lancer la commande `guix shell guile -- guile` si vous préférez ne pas installer Guile dans votre profil utilisateur.

Dans les exemples suivants, les lignes montrent ce que vous devez taper sur la REPL ; les lignes commençant par « ⇒ » montrent le résultat de l'évaluation, tandis que les lignes commençant par « ⊣ » montrent ce qui est affiché. Voir Section "Using Guile Interactively" dans *GNU Guile Reference Manual*, pour plus d'information sur la REPL.

- La syntaxe de Scheme se résume à un arbre d'expressions (ou une *s-expression* dans le jargon Lisp). Une expression peut être un litéral comme un nombre ou une chaîne de caractères, ou composée d'une liste d'autres éléments composés et litéraux, entourée de parenthèses. `#true` et `#false` (abrégés `#t` et `#f`) correspondent aux booléens « vrai » et « faux ».

  Voici des exemples d'expressions valides :

  ```
  "Bonjour le monde !"
  ⇒ "Bonjour le monde !"

  17
  ⇒ 17

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

- Ce dernier exemple est un appel de fonction imbriqué dans un autre appel de fonction. Lorsqu'une expression parenthésée est évaluée, le premier élément est la fonction et le reste sont les arguments passés à la fonction. Chaque fonction renvoie la dernière expression évaluée.
- On peut déclarer des fonctions anonymes — des *procédures* en Scheme — avec le terme `lambda` :

  ```
  (lambda (x) (* x x))
  ```

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

La procédure ci-dessus renvoie le carré de son argument. Comme tout est une expression, l'expression `lambda` renvoie une procédure anonyme, qui peut ensuite être appliquée à un argument :

```
((lambda (x) (* x x)) 3)
⇒ 9
```

Les procédures sont des valeurs normales comme les nombres, les chaines de caractère, les booléens, etc.

- On peut assigner un nom global à tout ce qu'on veut avec `define` :

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

- On peut définir des procédure de manière plus concise avec la syntaxe suivante :

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

- On peut créer une structure de liste avec la procédure `list` :

```
(list 2 a 5 7)
⇒ (2 3 5 7)
```

- Des procédures standards sont fournies par le module `(srfi srfi-1)` pour créer et traiter des listes (voir Section "SRFI-1" dans *le manuel de référence de GNU Guile*). Voici certaines des plus utiles en action :

```
(use-modules (srfi srfi-1))  ; importe les procédures de traitement des listes

(append (list 1 2) (list 3 4))
⇒ (1 2 3 4)

(map (lambda (x) (* x x)) (list 1 2 3 4))
⇒ (1 4 9 16)

(delete 3 (list 1 2 3 4))        ⇒ (1 2 4)
(filter odd? (list 1 2 3 4))     ⇒ (1 3)
(remove even? (list 1 2 3 4))    ⇒ (1 3)
(find number? (list "a" 42 "b")) ⇒ 42
```

Remarquez que le premier argument de `map`, `filter`, `remove` et `find` est une procédure !

- La *quote* (l'apostrophe) désactive l'évaluation d'une expression parenthésée, aussi appelée S-expression ou « s-exp » : le premier élément n'est pas appelé avec les autres éléments en argument (voir Section "Expression Syntax" dans *GNU Guile Reference Manual*). Donc, il renvoie une liste de termes.

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

'(2 a 5 7)
⇒ (2 a 5 7)
```

- La `quasiquote` (`` ` ``, l'apostrophe à l'envers) désactive l'évaluation d'une expression parenthésée jusqu'à ce qu'un `unquote` (`,`, une virgule) la réactive. De cette manière, on garde un contrôle fin sur ce qui est évalué et sur ce qui ne l'est pas.

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

  Remarquez que le résultat ci-dessus est une liste d'éléments mixtes : des nombres, des symboles (ici `a`) et le dernier élément est aussi une liste.

- Guix définit une variante des S-expressions gonflées aux stéroïdes appelées *G-expressions* ou « gexps », qui fournit une variante de `quasiquote` et `unquote` : `#~` (ou `gexp`) et `#$` (ou `ungexp`). Elles vous permettent *d'échelonner du code pour une future exécution*.

  Par exemple, vous rencontrerez des gexps dans certaines définitions de paquets qui fournissent du code à exécuter pendant la construction du paquet. Elles ressemblent à ceci :

```
(use-modules (guix gexp)           ;pour qu'on puisse écrire des gexps
             (gnu packages base))  ;pour « coreutils »

;; Ci-dessous, une G-expression représentant du code échelonné.
#~(begin
    ;; Invoque « ls » du paquet défini par la variable
    ;; « coreutils ».
    (system* #$(file-append coreutils "/bin/ls") "-l")

    ;; Crée le répertoire de sortie de ce paquet.
    (mkdir #$output))
```

  Voir Section "G-Expressions" dans *le manuel de référence de GNU Guix*, pour plus de détails sur les gexps.

- On peut nommer localement plusieurs variables avec `let` (voir Section "Local Bindings" dans *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
```

  On peut utiliser `let*` pour permettre au déclarations de variables ultérieures d'utiliser les définitions précédentes.

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

⇒ `(2 6)`

- On utilise typiquement des *mot-clés* pour identifier les paramètres nommés d'une procédure. Ils sont précédés de `#:` (dièse, deux-points) suivi par des caractères alphanumériques : `#:comme-ça`. Voir Section "Keywords" dans *GNU Guile Reference Manual*.
- On utilise souvent le signe pourcent `%` pour les variables globales non modifiables à l'étape de construction. Remarquez que ce n'est qu'une convention, comme `_` en C. Scheme traite `%` de la même manière que les autres lettres.
- On peut créer des modules avec `define-module` (voir Section "Creating Guile Modules" dans *GNU Guile Reference Manual*). Par exemple :

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

  défini le module `guix build-system ruby` qui doit se situer dans `guix/build-system/ruby.scm` quelque part dans le chemin de recherche de Guile. Il dépend du module `(guix store)` et exporte deux variables, `ruby-build` et `ruby-build-system`.

  Voir Section "Modules de paquets" dans *le manuel de référence de GNU Guix*, pour plus d'information sur les modules qui définissent des paquets.

  Pour aller plus loin: Scheme est un langage qui a été beaucoup utilisé pour enseigner la programmation et vous trouverez plein de supports qui l'utilisent. Voici une liste de documents qui vous en apprendront plus sur le Scheme :

  - *A Scheme Primer* (`https://spritely.institute/static/papers/scheme-primer.html`), par Christine Lemmer-Webber et le Spritely Institute.
  - Scheme at a Glance (`http://www.troubleshooters.com/codecorn/scheme_guile/hello.htm`), par Steve Litt.
  - *Structure and Interpretation of Computer Programs* (`https://sarabander.github.io/sicp/`), par Harold Abelson et Gerald Jay Sussman, avec Julie Sussman. Souvent appelé "SICP", ce livre est une référence.

    Vous pouvez aussi l'installer et le lire sur votre ordinateur :

```
guix install sicp info-reader
info sicp
```

  Vous trouverez plus de livres, de didacticiels et d'autres ressources sur `https://schemers.org/`.

2 Empaquetage

Ce chapitre est conçu pour vous enseigner comment ajouter des paquets à la collection de paquets de GNU Guix. Pour cela, vous devrez écrire des définitions de paquets en Guile Scheme, les organiser en modules et les construire.

2.1 Didacticiel d'empaquetage

GNU Guix se démarque des autres gestionnaire de paquets en étant *bidouillable*, surtout parce qu'il utilise GNU Guile (`https://www.gnu.org/software/guile/`), un langage de programmation de haut-niveau puissant, l'un des dialectes Scheme (`https://fr.wikipedia.org/wiki/Scheme`) de la famille Lisp (`https://fr.wikipedia.org/wiki/Lisp`).

Les définitions de paquets sont aussi écrites en Scheme, ce qui le rend plus puissant de manière assez unique par rapport aux autres gestionnaires de paquets qui utilisent des scripts shell ou des langages simples.

- Vous pouvez utiliser des fonctions, des structures, des macros et toute l'expressivité de Scheme dans vos définitions de paquets.
- L'héritage facilite la personnalisation d'un paquet en héritant d'un autre et en modifiant uniquement les points nécessaires.
- Traitement par lot : la collection des paquets entière peut être analysée, filtrée et traitée. Vous voulez construire un serveur sans interface graphique ? C'est possible. Vous voulez tout reconstruire à partir des sources avec des drapeaux d'optimisation spécifiques ? Passez l'argument `#:make-flags "..."` à la liste des paquets. Ce ne serait pas aberrant de penser au drapeau USE de Gentoo (`https://wiki.gentoo.org/wiki/USE_flag`), mais cela va plus loin : la personne qui crée les paquets n'a pas besoin de penser à l'avance à ces changements, ils peuvent être *programmés* par l'utilisateur ou l'utilisatrice !

Le didacticiel suivant traite des bases de la création de paquets avec Guix. Il ne présuppose aucune connaissance du système Guix ni du langage Lisp. On ne s'attend qu'à ce que vous aillez une certaine familiarité avec la ligne de commande et des connaissances de base en programmation.

2.1.1 Un paquet « hello world »

La section « Définir des paquets » du manuel explique les bases de l'empaquetage avec Guix (voir Section "Définir des paquets" dans *le manuel de référence de GNU Guix*). Dans la section suivante, nous reparlerons en partie de ces bases.

GNU Hello est un exemple de projet qui sert d'exemple idiomatique pour l'empaquetage. Il utilise le système de construction de GNU (`./configure && make && make install`). Guix fournit déjà une définition de paquet qui est un parfait exemple pour commencer. Vous pouvez voir sa déclaration avec `guix edit hello` depuis la ligne de commande. Voyons à quoi elle ressemble :

```
(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+)))
```

Comme vous pouvez le voir, la plus grosse partie est assez simple. Mais examinons les champs ensemble :

'`name`' Le nom du projet. Avec les conventions de Scheme, on préfère le laisser en minuscule, sans tiret du bas, en séparant les mots par des tirets.

'`source`' Ce champ contient une description de l'origine du code source. L'enregistrement `origin` contient ces champs :

1. La méthode, ici `url-fetch` pour télécharger via HTTP/FTP, mais d'autres méthodes existent, comme `git-fetch` pour les dépôts Git.
2. L'URI, qui est typiquement un emplacement `https://` pour `url-fetch`. Ici le code spécial « mirror://gnu » fait référence à une ensemble d'emplacements bien connus, qui peuvent tous être utilisés par Guix pour récupérer la source, si l'un d'entre eux échoue.
3. La somme de contrôle `sha256` du fichier demandé. C'est essentiel pour s'assurer que la source n'est pas corrompue. Remarquez que Guix fonctionne avec des chaînes en base32, d'où l'appel à la fonction `base32`.

'`build-system`'
C'est ici que la puissance d'abstraction du langage Scheme brille de toute sa splendeur : dans ce cas, le `gnu-build-system` permet d'abstraire les fameuses invocations shell `./configure && make && make install`. Parmi les autres systèmes de construction on trouve le `trivial-build-system` qui ne fait rien et demande de programmer toutes les étapes de construction, le `python-build-system`, `emacs-build-system` et bien d'autres (voir Section "Systèmes de construction" dans *le manuel de référence de GNU Guix*).

'`synopsis`'
Le synopsis devrait être un résumé court de ce que fait le paquet. Pour beaucoup de paquets, le slogan de la page d'accueil du projet est approprié pour le synopsis.

‘`description`’
: Comme le synopsis, vous pouvez réutiliser la description de la page d'accueil du projet. Remarquez que Guix utilise la syntaxe Texinfo.

‘`home-page`’
: Utilisez l'adresse en HTTPS si elle est disponible.

‘`license`’
: Voir `guix/licenses.scm` dans les sources du projet pour une liste complète des licences disponibles.

Il est temps de construire notre premier paquet ! Rien de bien compliqué pour l'instant : nous allons garder notre exemple avec `my-hello`, une copie de la déclaration montrée plus haut.

Comme avec le rituel « Hello World » enseigné avec la plupart des langages de programmation, ce sera sans doute l'approche la plus « manuelle » d'empaquetage que vous utiliserez. Nous vous montrerons une configuration idéale plus tard, pour l'instant nous allons suivre la voie la plus simple.

Enregistrez ce qui suit dans un fichier nommé `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+))
```

Nous allons expliquer le code supplémentaire dans un moment.

Essayez de jouer avec les différentes valeurs des différents champs. Si vous changez la source, vous devrez mettre à jour la somme de contrôle. En fait, Guix refusera de construire quoi que ce soit si la somme de contrôle donnée ne correspond pas à la somme de contrôle calculée de la source téléchargée. Pour obtenir la bonne somme de contrôle pour une déclaration de paquet, vous devrez télécharger la source, calculer la somme de contrôle sha256 et la convertir en base32.

Heureusement, Guix peut automatiser cette tache pour nous ; tout ce qu'on doit faire est de lui fournir l'URI :

```
$ 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
```

Dans ce cas particulier, la sortie nous dit quel miroir a été choisi. Si le résultat de la commande au-dessus n'est pas le même que ce qui est montré, mettez à jour votre déclaration `my-hello` en fonction.

Remarquez que les archives des paquets GNU sont accompagnées de leur signature OpenPGP, donc vous devriez vérifier la signature de cette archive avec « gpg » pour l'authentifier avant d'aller plus loin :

```
$ 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 faite le dim. 16 nov. 2014 13:08:37 CET
gpg:                avec la clef RSA A9553245FDE9B739
gpg: Bonne signature de « Sami Kerola (https://www.iki.fi/kerolasa/) <kerolasa@iki.fi>
gpg: Attention : cette clef n'est pas certifiée avec une signature de confiance.
gpg:             Rien n'indique que la signature appartient à son propriétaire.
Empreinte de clef principale : 8ED3 96E3 7E38 D471 A005  30D3 A955 3245 FDE9 B739
```

Vous pouvez ensuite lancer

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

Vous devriez maintenant avoir `my-hello` dans votre profil !

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

Nous sommes allés aussi loin que possible sans aucune connaissance de Scheme. Avant de continuer sur des paquets plus complexes, il est maintenant temps de vous renforcer sur votre connaissance du langage Scheme. Voir Section 1.1 [Cours accéléré du langage Scheme], page 1, pour démarrer.

2.1.2 Configuration

Dans le reste de ce chapitre, nous nous appuierons sur vos connaissances de base du langage Scheme. Maintenant voyons les différentes configurations possibles pour travailler sur des paquets Guix.

Il y a plusieurs moyens de mettre en place un environnement d'empaquetage pour Guix.

Nous vous recommandons de travailler directement dans le dépôt des sources de Guix car ça facilitera la contribution au projet.

Mais d'abord, voyons les autres possibilités.

2.1.2.1 Fichier local

C'est ce que nous venons de faire avec '`my-hello`'. Avec les bases de Scheme que nous vous avons présentées, nous pouvons maintenant éclairer le sens du début du fichier. Comme le dit `guix package --help` :

```
-f, --install-from-file=FICHIER
                        installer le paquet évalué par le code dans
                        FICHIER
```

Ainsi, la dernière expression *doit* renvoyer un paquet, ce qui est le cas dans notre exemple précédent.

L'expression `use-modules` indique quels modules sont nécessaires dans le fichier. Les modules sont des collections de valeurs et de procédures. Ils sont souvent appelés « bibliothèques » ou « paquets » dans d'autres langages de programmation.

2.1.2.2 Canaux

Guix et sa collection de paquet peut être étendu par des *canaux*. Un canal est un dépôt Git public ou non, qui contient des fichiers `.scm` qui fournissent des paquets (voir Section "Définition des paquets" dans *le manuel de référence de GNU Guix*) ou des services (voir Section "Définir des services" dans *le manuel de référence de GNU Guix*).

Comment créer un canal ? Tout d'abord, créez un répertoire qui contiendra vos fichiers `.scm`, disons `~/mon-canal` :

```
mkdir ~/mon-canal
```

Imaginons que vous souhaitiez ajouter le paquet '`my-hello`' que nous avons vu plus tôt. Il a d'abord besoin de quelques ajustement :

```
(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+)))
```

Remarquez que nous avons assigné la valeur du paquet à un nom de variable exportée avec `define-public`. Cela assigne en fait le paquet à la variable `my-hello` pour qu'elle puisse être utilisée, par exemple en dépendance d'un autre paquet.

Si vous utilisez `guix package --install-from-file=my-hello.scm` avec le fichier précédent, la commande échouera car la dernière expression, `define-public`, ne renvoie pas un paquet. Si vous voulez utiliser `define-public` dans ce cas tout de même, assurez-vous que le fichier termine par une évaluation de `my-hello` :

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

my-hello
```

Ce dernier exemple n'est pas très typique.

Maintenant, comme rendre ce paquet visible pour les commandes `guix` afin de tester vos paquets ? Vous devez ajouter le répertoire au chemin de recherche avec l'option en ligne de commande `-L`, comme dans ces exemples :

```
guix show -L ~/mon-canal my-hello
guix build -L ~/mon-canal my-hello
```

L'étape finale consiste à transformer `~/mon-canal` en un vrai canal, pour rendre disponible votre collection de paquets via n'importe quelle commande `guix`. Pour cela, vous devez d'abord en faire un dépôt Git :

```
cd ~/mon-canal
git init
git add my-hello.scm
git commit -m "Premier commit sur mon canal."
```

Et voilà, vous avez un canal ! À partir de maintenant, vous pouvez ajouter ce canal à votre configuration des canaux dans `~/.config/guix/channels.scm` (voir Section “Spécifier des canaux supplémentaires” dans *le manuel de référence de GNU Guix*). En supposant que vous gardez votre canal localement pour l'instant, le fichier `channels.scm` ressemblerait à quelque chose comme ça :

```
(append (list (channel
```

```
                (name 'mon-canal)
                (url (string-append "file://" (getenv "HOME")
                                    "/mon-canal"))))
          %default-channels)
```

La prochaine fois que vous exécuterez `guix pull`, votre canal sera récupéré et les paquets qu'il définit seront directement disponibles pour toutes les commandes `guix`, même si vous n'utilisez pas `-L`. La commande `guix describe` vous montrera que Guix utilise effectivement à la fois `mon-canal` et les canaux `guix`.

Voir Section "Écrire de nouveaux de canaux" dans *le manuel de référence de GNU Guix* pour des détails.

2.1.2.3 Bidouillage direct dans le dépôt git

Nous vous recommandons de travailler directement sur le projet Guix : cela réduit le travail nécessaire quand vous voudrez soumettre vos changements en amont pour que la communauté puisse bénéficier de votre dur labeur !

Contrairement à la plupart des distributions logiciels, le dépôt Guix contient à la fois les outils (dont le gestionnaire de paquets) et les définitions des paquets. Nous avons fait ce choix pour permettre aux développeurs et développeuses de profiter de plus de flexibilité pour changer l'API sans rien casser, en mettant à jour tous les paquets en même temps. Cela réduit l'inertie dans le développement.

Clonez le dépôt Git (`https://git-scm.com/`) officiel :

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

Dans le reste de cet article, nous utiliserons '`$GUIX_CHECKOUT`' pour faire référence à l'emplacement de ce clone.

Suivez les instructions du manuel (voir Section "Contribuer" dans *le manuel de référence de GNU Guix*) pour mettre en place l'environnement du dépôt.

Une fois prêts, vous devriez pouvoir utiliser les définitions des paquets de l'environnement du dépôt.

N'ayez pas peur de modifier les définitions des paquets que vous trouverez dans '`$GUIX_CHECKOUT/gnu/packages`'.

Le script '`$GUIX_CHECKOUT/pre-inst-env`' vous permet d'utiliser '`guix`' sur la collection de paquets du dépôt (voir Section "Lancer Guix avant qu'il ne soit installé" dans *le manuel de référence de GNU Guix*).

- Recherchez des paquets, comme 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
```

- Construisez un paquet, ici Ruby version 2.1 :

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

- Installez-le dans votre profil utilisateur :

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

- Vérifiez que vous n'avez pas fait l'une des erreurs courantes :

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

Guix essaye de maintenir un bon standard d'empaquetage ; quand vous contribuez au projet Guix, rappelez-vous de

- suivre le style de code (voir Section "Style de code" dans *le manuel de référence de GNU Guix*),
- et de vérifier la check-list du manuel (voir Section "Envoyer des correctifs" dans *le manuel de référence de GNU Guix*).

Une fois que vous êtes satisfait du résultat, vous pouvez envoyer votre contribution pour qu'elle rentre dans Guix. Ce processus est aussi détaillé dans le manuel (voir Section "Contribuer" dans *le manuel de référence de GNU Guix*)

Guix est un projet communautaire, donc plus on est de fous, plus on rit !

2.1.3 Exemple avancé

L'exemple « Hello World » précédent est le plus simple possible. Les paquets peuvent devenir plus complexes que cela et Guix peut gérer des scénarios plus avancés. Voyons un autre paquet plus sophistiqué (légèrement modifié à partir des sources) :

```
(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 compression)
  #:use-module (gnu packages pkg-config)
  #:use-module (gnu packages python)
  #:use-module (gnu packages ssh)
  #:use-module (gnu packages tls)
  #:use-module (gnu packages web))

(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)))
                ;; Suppression des logiciels embarqués.
                (snippet '(delete-file-recursively "deps"))))
        (build-system cmake-build-system)
        (outputs '("out" "debug"))
        (arguments
         `(#:tests? #true                         ; Lancer la suite de tests (c'est la v
           #:configure-flags '("-DUSE_SHA1DC=ON") ; détection de collision SHA-1
           #: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")))))
             ;; Lancer les tests avec plus de verbosité.
             (replace 'check
               (lambda* (#:key tests? #:allow-other-keys)
                 (when tests?
                   (invoke "./libgit2_clar" "-v" "-Q"))))
             (add-after 'unpack 'make-files-writable-for-tests
               (lambda _ (for-each make-file-writable (find-files "."))))))
        (inputs
         (list libssh2 http-parser python-wrapper))
        (native-inputs
         (list pkg-config))
        (propagated-inputs
         ;; Ces deux bibliothèques sont dans « Requires.private », dans libgit2.pc.
         (list openssl 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))))
```

(Dans les cas où vous voulez seulement changer quelques champs d'une définition de paquets, vous devriez utiliser l'héritage au lieu de tout copier-coller. Voir plus bas.)

Parlons maintenant de ces champs en détail.

2.1.3.1 La méthode `git-fetch`

Contrairement à la méthode `url-fetch`, `git-fetch` a besoin d'un `git-reference` qui prend un dépôt Git et un commit. Le commit peut être n'importe quelle référence Git comme des tags, donc si la `version` a un tag associé, vous pouvez l'utiliser directement. Parfois le tag est précédé de `v`, auquel cas vous pouvez utiliser `(commit (string-append "v" version))`.

Pour vous assurer que le code source du dépôt Git est stocké dans un répertoire avec un nom descriptif, utilisez `(file-name (git-file-name name version))`.

Vous pouvez utiliser la procédure `git-version` pour calculer la version quand vous empaquetez des programmes pour un commit spécifique, en suivant le guide de contribution (voir Section "Numéros de version" dans *le manuel de référence de GNU Guix*).

Comment obtenir le hash `sha256`, vous demandez-vous ? En invoquant `guix hash` sur un clone du commit voulu, de cette manière :

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

`guix hash -rx` calcul un SHA256 sur le répertoire entier, en excluant le sous-répertoire `.git` (voir Section "Invoquer guix hash" dans *le manuel de référence de GNU Guix*).

Dans le futur, `guix download` sera sans doute capable de faire cela pour vous, comme il le fait pour les téléchargements directs.

2.1.3.2 Les bouts de code

Les bouts de code (snippet) sont des fragments quotés (c.-à-d. non évalués) de code Scheme utilisés pour modifier les sources. C'est une alternative aux fichiers `.patch` traditionnels, plus proche de l'esprit de Guix. À cause de la quote, le code n'est évalué que lorsqu'il est passé au démon Guix pour la construction. Il peut y avoir autant de bout de code que nécessaire.

Les bouts de code on parfois besoin de modules Guile supplémentaires qui peuvent être importés dans le champ `modules`.

2.1.3.3 Entrées

Il y a trois types d'entrées. En résumé :

native-inputs
: Requis pour construire mais pas à l'exécution – installer un paquet avec un substitut n'installera pas ces entrées.

inputs
: Installées dans le dépôt mais pas dans le profil, et présentes à la construction.

propagated-inputs
: Installées dans le dépôt et dans le profil, et présentes à la construction.

Voir Section "Référence de package" dans *le manuel de référence de GNU Guix* pour plus de détails.

La différence entre les différents types d'entrées est importante : si une dépendance peut être utilisée comme *entrée* plutôt que comme *entrée propagée*, il faut faire ça, sinon elle « polluera » le profil utilisateur sans raison.

Par exemple, si vous installez un programme graphique qui dépend d'un outil en ligne de commande, vous êtes probablement intéressé uniquement par la partie graphique, donc inutile de forcer l'outil en ligne de commande à être présent dans le profil utilisateur. Les dépendances sont gérés par les paquets, pas par les utilisateurs et utilisatrices. Les *entrées* permettent de gérer les dépendances sans ennuyer les utilisateurs et utilisatrices en ajoutant des fichiers exécutables (ou bibliothèque) inutiles dans leur profil.

Pareil pour *native-inputs* : une fois le programme installé, les dépendances à la construction peuvent être supprimées sans problème par le ramasse-miettes. Lorsqu'un substitut est disponible, seuls les *entrées* et les *entrées propagées* sont récupérées : les *entrées natives* ne sont pas requises pour installer un paquet à partir d'un substitut.

> **Remarque:** Vous trouverez ici et là des extraits où les entrées des paquets sont écrites assez différemment, comme ceci :
>
> ```
> ;; « L'ancien style » pour les entrées.
> (inputs
>  `(("libssh2" ,libssh2)
>    ("http-parser" ,http-parser)
>    ("python" ,python-wrapper)))
> ```
>
> C'est « l'ancien style », où chaque entrée est une liste que donne une étiquette explicite (une chaine). C'est une méthode prise en charge mais nous vous recommandons plutôt d'utiliser le style présenté plus haut. Voir Section "Référence de package" dans *le manuel de référence de GNU Guix*, pour plus d'informations.

2.1.3.4 Sorties

De la même manière qu'un paquet peut avoir plusieurs entrées, il peut aussi avoir plusieurs sorties.

Chaque sortie correspond à un répertoire différent dans le dépôt.

Vous pouvez choisir quelle sortie installer ; c'est utile pour préserver l'espace disque et éviter de polluer le profil utilisateur avec des exécutables et des bibliothèques inutiles.

La séparation des sorties est facultative. Lorsque le champ `outputs` n'est pas spécifié, l'unique sortie par défaut (le paquet complet donc) est `"out"`.

Les sorties séparées sont en général `debug` et `doc`.

Vous devriez séparer les sorties seulement si vous pouvez montrer que c'est utile : si la taille de la sortie est importante (vous pouvez comparer avec `guix size`) ou si le paquet est modulaire.

2.1.3.5 Arguments du système de construction

Le champ `arguments` est une liste de mot-clés et de valeurs utilisés pour configurer le processus de construction.

L'argument le plus simple est `#:tests?` et on l'utilise pour désactiver la suite de tests pendant la construction du paquet. C'est surtout utile si le paquet n'a pas de suite de tests. Nous vous recommandons fortement de laisser tourner la suite de tests s'il y en a une.

Un autre argument courant est `#:make-flags`, qui spécifie une liste de drapeaux à ajouter en lançant make, comme ce que vous feriez sur la ligne de commande. Par exemple, les drapeaux suivants

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

```
                    "CC=gcc")
```

se traduisent en

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

Cela indique que le compilateur C sera `gcc` et la variable `prefix` (le répertoire d'installation pour Make) sera `(assoc-ref %outputs "out")`, qui est une variable globale côté construction qui pointe vers le répertoire de destination dans le dépôt (quelque chose comme `/gnu/store/...-my-libgit2-20180408`).

De manière identique, vous pouvez indiquer les drapeaux de configuration :

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

La variable `%build-inputs` est aussi générée dans cette portée. C'est une liste d'association qui fait correspondre les noms des entrées à leur répertoire dans le dépôt.

Le mot-clé `phases` liste la séquence d'étapes du système de construction. Les phases usuelles sont `unpack`, `configure`, `build`, `install` et `check`. Pour en savoir plus, vous devez trouver la bonne définition du système de construction dans '`$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
            make-dynamic-linker-cache
            install-license-files
            reset-gzip-timestamps
            compress-documentation)))
```

Ou depuis la REPL :

```
(add-to-load-path "/path/to/guix/checkout")
,use (guix build gnu-build-system)
(map car %standard-phases)
⇒ (set-SOURCE-DATE-EPOCH set-paths install-locale unpack bootstrap patch-usr-bin-file
```

Si vous voulez en apprendre plus sur ce qui arrive pendant ces phases, consultez les procédures associées.

Par exemple, au moment d'écrire ces lignes, la définition de `unpack` dans le système de construction de GNU est :

```
(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? #t)
        ;; Make the source checkout files writable, for convenience.
        (for-each (lambda (f)
                    (false-if-exception (make-file-writable f)))
                  (find-files ".")))
      (begin
        (cond
         ((string-suffix? ".zip" source)
          (invoke "unzip" source))
         ((tarball? source)
          (invoke "tar" "xvf" source))
         (else
          (let ((name (strip-store-file-name source))
                (command (compressor source)))
            (copy-file source name)
            (when command
              (invoke command "--decompress" name)))))
        ;; Attempt to change into child directory.
        (and=> (first-subdirectory ".") chdir))))
```

Remarquez l'appel à `chdir` : il change de répertoire courant vers la source qui vient d'être décompressée. Ainsi toutes les phases suivantes utiliseront le répertoire des sources comme répertoire de travail, ce qui explique qu'on peut travailler directement sur les fichiers sources. Du moins, tant qu'une phase suivante ne change pas le répertoire de travail.

Nous modifions la liste des `%standard-phases` du système de construction avec la macro `modify-phases` qui indique la liste des modifications, sous cette formes :

- `(add-before` *`phase nouvelle-phase procédure`*`)` : Lance une *procédure* nommée *nouvelle-phase* avant *phase*.
- `(add-after` *`phase nouvelle-phase procédure`*`)` : Pareil, mais après la *phase*.
- `(replace` *`phase procédure`*`)`.
- `(delete` *`phase`*`)`.

La *procédure* prend en charge les arguments `inputs` et `outputs` sous forme de mot-clés. Les entrées (*natives*, *propagées* et simples) et répertoires de sortie sont référencés par leur nom dans ces variables. Ainsi `(assoc-ref outputs "out")` est le répertoire du dépôt de la sortie principale du paquet. Une procédure de phase ressemble à cela :

```
(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")))
  ;; ... ))
```

Its return value is ignored.

2.1.3.6 Échelonnage du code

Si vous avez été attentif, vous aurez remarqué la quasi-quote et la virgule dans le champ argument. En effet, le code de construction dans la déclaration du paquet ne doit pas être évalué côté client, mais seulement après avoir été passé au démon Guix. Ce mécanisme de passage de code entre deux processus s'appelle l'échelonnage de code (`https://arxiv.org/abs/1709.00833`).

2.1.3.7 Fonctions utilitaires

Lorsque vous modifiez les `phases`, vous aurez souvent besoin d'écrire du code qui ressemble aux invocation équivalentes (`make`, `mkdir`, `cp`, etc) couramment utilisées durant une installatio plus standard dans le monde Unix.

Certaines comme `chmod` sont natives dans Guile. Voir *Guile reference manual* pour une liste complète.

Guix fournit des fonctions utilitaires supplémentaires qui sont particulièrement utiles pour la gestion des paquets.

Certaines de ces fonctions se trouvent dans '`$GUIX_CHECKOUT/guix/guix/build/utils.scm`'. La plupart copient le comportement des commandes systèmes Unix traditionnelles :

`which` Fonctionne comme la commande système '`which`'.

`find-files`
: Fonctionne un peu comme la commande '`find`'.

`mkdir-p` Fonctionne comme '`mkdir -p`', qui crée tous les parents si besoin.

`install-file`
: Fonctionne comme '`install`' pour installer un fichier vers un répertoire (éventuellement non existant). Guile a `copy-file` qui fonctionne comme '`cp`'.

`copy-recursively`
: Fonctionne comme '`cp -r`'.

`delete-file-recursively`
: Fonctionne comme '`rm -rf`'.

`invoke` Lance un exécutable. Vous devriez utiliser cela à la place de `system*`.

`with-directory-excursion`
: Lance le corps dans un répertoire de travail différent, puis revient au répertoire de travail précédent.

`substitute*`
: Une fonction similaire à `sed`.

Voir Section "Utilitaires de construction" dans *le manuel de référence de GNU Guix*, pour plus d'informations sur ces utilitaires.

2.1.3.8 Préfixe de module

La licence dans notre dernier exemple a besoin d'un préfixe à cause de la manière dont le module `licenses` a été importé dans le paquet, avec `#:use-module ((guix licenses) #:prefix license:)`. Le mécanisme d'import de module de Guile (voir Section "Using Guile Modules" dans *Guile reference manual*) permet de contrôler complètement l'espace de nom. Cela évite les conflits entre, disons, la variable '`zlib`' de '`licenses.scm`' (un *licence*) et la variable '`zlib`' de '`compression.scm`' (un *paquet*).

2.1.4 Autres systèmes de construction

Ce que nous avons vu jusqu'ici couvre la majeur partie des paquets qui utilisent un système de construction autre que `trivial-build-system`. Ce dernier n'automatise rien et vous laisse tout construire par vous-même. C'est plus exigeant et nous n'en parlerons pas pour l'instant, mais heureusement il est rarement nécessaire d'aller jusqu'à ces extrémités.

Pour les autres systèmes de construction, comme ASDF, Emacs, Perl, Ruby et bien d'autres, le processus est très similaire à celui du système de construction de GNU en dehors de quelques arguments spécialisés.

Voir Section "Systèmes de construction" dans *le manuel de référence de GNU Guix*, pour plus d'informations sur les systèmes de construction, ou voir le code source dans les répertoires '`$GUIX_CHECKOUT/guix/build`' et '`$GUIX_CHECKOUT/guix/build-system`'.

2.1.5 Définition programmable et automatisée

Nous ne le répéterons jamais assez : avoir un langage de programmation complet à disposition nous permet de faire bien plus de choses que la gestion de paquets traditionnelle.

Illustrons cela avec certaines fonctionnalités géniales de Guix !

2.1.5.1 Les importateurs récursifs

Certains systèmes de constructions sont si bons qu'il n'y a presque rien à écrire pour créer un paquet, au point que cela devient rapidement répétitif et pénible. L'une des raisons d'être des ordinateurs est de remplacer les êtres humains pour ces taches barbantes. Disons donc à Guix de faire cela pour nous et de créer les définitions de paquets pour un paquet R venant de CRAN (la sortie est coupée par souci de place) :

```
$ 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
      (list r-ggplot2 r-jmvcore r-r6 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)))
```

L'importateur récursif n'importera pas les paquets pour lesquels Guix a déjà une définition, sauf pour le tout premier.

Toutes les applications ne peuvent pas être empaquetées de cette manière, seules celles qui s'appuient sur un nombre restreint de systèmes pris en charge le peuvent. Vous trouverez la liste complète des importateurs dans la section dédiée du manuel (voir Section "Invoquer guix import" dans *le manuel de référence de GNU*).

2.1.5.2 Mise à jour automatique

Guix peut être assez intelligent pour vérifier s'il y a des mises à jour sur les systèmes qu'il connaît. Il peut rapporter les paquets anciens avec

```
$ guix refresh hello
```

La plupart du temps, mettre à jour un paquet vers une nouvelle version ne demande pas beaucoup plus que de changer le numéro de version et la somme de contrôle. Guix peut aussi le faire automatiquement :

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

2.1.5.3 Héritage

Si vous avez commencé à regarder des définitions de paquets existantes, vous avez peut-être remarqué qu'un certain nombre d'entre elles ont un champ `inherit` :

```
(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 (list `(,gtk+ "bin")))))
```

Tous les champs non spécifiés héritent du paquet parent. C'est très pratique pour créer un paquet alternatif, par exemple avec une source, une version ou des options de compilation différentes.

2.1.6 Se faire aider

Malheureusement, certaines applications peuvent être difficiles à empaqueter. Parfois elles ont besoin d'un correctif pour fonctionner avec la hiérarchie du système de fichiers non standard imposée par de dépôt. Parfois les tests ne se lancent pas correctement (vous pouvez les passer mais ce n'est pas recommandé). Parfois le paquet n'est pas reproductible.

Si vous êtes bloqué·e, incapable de trouver comme corriger un problème d'empaquetage, n'hésitez pas à demander de l'aide à la communauté.

voir la la page d'accueil de Guix (`https://guix.gnu.org/fr/contact/`) pour plus d'informations sur les listes de diffusion, IRC, etc.

2.1.7 Conclusion

Ce didacticiel vous a montré la gestion des paquets sophistiquée dont Guix se targue. Maintenant, nous avons restreint cette introduction au système `gnu-build-system` qui est un niveau d'abstraction essentiel sur lequel des niveaux d'abstraction plus avancés se reposent.

Comment continuer ? Nous devrions ensuite disséquer le fonctionnement interne des systèmes de construction en supprimant toutes les abstractions, avec le `trivial-build-system` : cela vous permettra de bien comprendre le processus avant de voir des techniques plus avancées et certains cas particuliers.

D'autres fonctionnalités que vous devriez explorer sont l'édition interactive et les possibilités de débogage de Guix fournies par la REPL de Guile.

Ces fonctionnalités avancées sont complètement facultatives et peuvent attendre ; maintenant vous devriez prendre une pause bien méritée. Avec ce dont nous venons de parler ici vous devriez être bien armé·e pour empaqueter de nombreux paquets. Vous pouvez commencer dès maintenant et on espère voir votre contribution bientôt !

2.1.8 Références

- La référence des paquets dans le manuel (`https://guix.gnu.org/manual/devel/fr/html_node/reference-de-package.html`)

- le guide de bidouillage de GNU Guix de Pjotr (`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`), d'Andreas Enge

3 Configuration du système

Guix propose un langage flexible pour déclarer la configuration de votre système Guix. Cette flexibilité peut parfois paraître écrasante. Le but de ce chapitre est de vous montrer quelques concepts de configuration avancés.

voir Section “Configuration du système” dans *le manuel de référence de GNU Guix* pour une référence complète.

3.1 Connexion automatique à un TTY donné

Tandis que le manuel de Guix explique comment connecter automatiquement un utilisateur sur *tous* les TTY (voir Section “connexion automatique à un TTY” dans *le manuel de référence de Guix*), vous pourriez préférer avoir un utilisateur connecté sur un TTY et configurer les autres TTY pour connecter d’autres utilisateurs ou personne. Remarquez que vous pouvez connecter automatiquement un utilisateur sur n’importe quel TTY, mais il est recommandé d’éviter `tty1`, car par défaut, il est utilisé pour afficher les avertissements et les erreurs des journaux systèmes.

Voici comment on peut configurer la connexion d’un utilisateur sur un 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))
```

On peut aussi utiliser `compose` (voir Section “Higher-Order Functions” dans *The Guile Reference Manual*) avec `auto-login-to-tty` pour connecter plusieurs utilisateurs sur différents ttys.

Enfin, une mise en garde. Configurer la connexion automatique à un TTY signifie que n’importe qui peut allumer votre ordinateur et lancer des commandes avec votre utilisateur normal. Cependant, si vous avez une partition racine chiffrée, et donc qu’il faut déjà saisir une phrase de passe au démarrage du système, la connexion automatique peut être un choix pratique.

3.2 Personnalisation du noyau

Guix est, en son cœur, une distribution source avec des substituts (voir Section “Substituts” dans *le manuel de référence de GNU Guix*), et donc construire des paquets à partir de leur code source est normal pendant les installations et les mis à jour de paquets. Malgré tout, c'est aussi normal d'essayer de réduire le temps passé à compiler des paquets, et les changements récents et futurs concernant la construction et la distribution des substituts continue d'être un sujet de discussion dans le projet Guix.

Le noyau, bien qu'il ne demande pas énormément de RAM pour être construit, prend assez long à construire sur une machine usuelle. La configuration du noyau officielle, comme avec la plupart des autres distributions GNU/Linux, penche du côté de l'inclusivité, et c'est vraiment ça qui rend la construction aussi longue à partir des sources.

Le noyau Linux, cependant, peut aussi être décrit comme un simple paquet comme les autres, et peut donc être personnalisé comme n'importe quel autre paquet. La procédure est un peu différente, même si c'est surtout dû à la nature de la définition du paquet.

Le paquet du noyau `linux-libre` est en fait une procédure qui crée un paquet.

```
(define* (make-linux-libre* version gnu-revision source supported-systems
                            #:key
                            (extra-version #f)
                           ;; Un fonction qui prend une architecture et une variante
                           ;; Voir kernel-config si vous voulez un exemple.
                            (configuration-file #f)
                            (defconfig "defconfig")
                            (extra-options (default-extra-linux-options version)))
  ...)
```

Le paquet `linux-libre` actuel pour la série 5.15.x, est déclaré comme ceci :

```
(define-public linux-libre-5.15
  (make-linux-libre* linux-libre-5.15-version
                     linux-libre-5.15-gnu-revision
                     linux-libre-5.15-source
                     '("x86_64-linux" "i686-linux" "armhf-linux"
                       "aarch64-linux" "riscv64-linux")
                     #:configuration-file kernel-config))
```

Les clés qui n'ont pas de valeur associée prennent leur valeur par défaut dans la définition de `make-linux-libre`. Lorsque vous comparez les deux bouts de code ci-dessus, remarquez le commentaire qui correspond à `#:configuration-file`. À cause de cela, il n'est pas facile d'inclure une configuration personnalisée du noyau à partir de la définition, mais ne vous inquiétez pas, il y a d'autres moyens de travailler avec ce qu'on a.

Il y a deux manières de créer un noyau avec une configuration personnalisée. La première consiste à fournir un fichier `.config` standard au processus de construction en ajoutant un fichier `.config` comme entrée native de notre noyau. Voici un bout de code correspondant à la phase `'configure` de la définition de paquet `make-linux-libre` :

```
(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)))
```

Et voici un exemple de paquet de noyau. Le paquet `linux-libre` n'a rien de spécial, on peut en hériter et remplacer ses champs comme n'importe quel autre paquet :

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

Dans le même répertoire que le fichier définissant `linux-libre-E2140` se trouve un fichier nommé `E2140.config`, qui est un fichier de configuration du noyau. Le mot-clé `defconfig` de `make-linux-libre` reste vide ici, donc la configuration du noyau dans le paquet est celle qui sera incluse dans le champ `native-inputs`.

La deuxième manière de créer un noyau personnalisé est de passer une nouvelle valeur au mot-clé `extra-options` de la procédure `make-linux-libre`. Le mot-clé `extra-options` fonctionne avec une autre fonction définie juste en dessous :

```
(define (default-extra-linux-options version)
  `(;; https://lists.gnu.org/archive/html/guix-devel/2014-04/msg00039.html
   ("CONFIG_DEVPTS_MULTIPLE_INSTANCES" . #true)
   ;; Modules requis pour 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"))
```

Et dans le script configure personnalisé du paquet « make-linux-libre » :

```
;; 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")
```

Donc, en ne fournissant pas de fichier de configuration le fichier `.config` est au départ vide et on écrit ensuite l'ensemble des drapeaux que l'on veut. Voici un autre noyau personnalisé :

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

(define-public linux-libre-macbook41
  ;; XXX: Accède à la procédure interne « make-linux-libre* », qui est privée
  ;; et n'est pas exportée, et pourrait changer dans le futur.
  ((@@ (gnu packages linux) make-linux-libre*)
   (@@ (gnu packages linux) linux-libre-version)
   (@@ (gnu packages linux) linux-libre-gnu-revision)
   (@@ (gnu packages linux) linux-libre-source)
   '("x86_64-linux")
   #:extra-version "macbook41"
   #:extra-options %macbook41-config-options))
```

Dans l'exemple ci-dessus `%fale-systems` est un ensemble de drapeaux qui activent la prise en charge de différents systèmes de fichiers, `%efi-support` active la prise en charge de l'EFI et `%emulation` permet à une machine x86_64-linux de fonctionner aussi en mode 32-bits. La procédure `default-extra-linux-options` est définie plus haut, et elles devaient être utilisée pour éviter de perdre les options de configuration par défaut dans le mot-clé `extra-options`.

Tout ça est bien beau, mais comment savoir quels modules sont requis pour un système en particulier ? Il y a deux ressources qui peuvent être utiles pour répondre à cette question : le manuel de Gentoo (`https://wiki.gentoo.org/wiki/Handbook:AMD64/Installation/Kernel`) et la documentation du noyau (`https://www.kernel.org/doc/html/latest/admin-guide/README.html?highlight=localmodconfig`). D'après la documentation du noyau, il semble que la commande `make localmodconfig` soit la bonne.

Pour lancer `make localmodconfig` on doit d'abord récupérer et décompresser le code source du noyau :

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

Une fois dans le répertoire contenant le code source lancez `touch .config` pour créer un fichier `.config` initialement vide pour commencer. `make localmodconfig` fonctionne en remarquant que avec déjà un `.config` et en vous disant ce qu'il vous manque. Si le fichier est vide, il vous manquera tout ce qui est nécessaire. L'étape suivante consiste à lancer :

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

et regardez la sortie. Remarquez que le fichier `.config` est toujours vide. La sortie contient en général deux types d'avertissements. Le premier commence par « WARNING » et peut être ignoré dans notre cas. Le deuxième dit :

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

Pour chacune de ces lignes, copiez la partie `CONFIG_XXXX_XXXX` dans le `.config` du répertoire et ajoutez `=m` pour qu'à la fin il ressemble à cela :

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

Après avoir copié toutes les options de configuration, lancez `make localmodconfig` de nouveau pour vous assurer que vous n'avez pas de sortie commençant par « module ». Après tous ces modules spécifiques à la machine, il y en a encore quelques uns que nous devons aussi définir. `CONFIG_MODULES` est nécessaire pour que nous puissions construire et charger les modules séparément et ne pas tout construire dans le noyau. `CONFIG_BLK_DEV_SD` est requis pour lire les disques durs. Il est possible que vous aillez besoin de quelques autres modules.

Cet article n'a pas pour but de vous guider dans la configuration de votre propre noyau cependant, donc si vous décidez de construire un noyau personnalisé vous devrez chercher d'autres guides pour créer un noyau qui vous convient.

La deuxième manière de configurer le noyau utilise un peu plus les fonctionnalités de Guix et vous permettent de partager des bouts de configuration entre différents noyaux. Par exemple, toutes les machines avec un démarrage EFI ont besoin d'un certain nombre de configurations. Tous les noyaux vont probablement partager une liste de systèmes de fichiers à prendre en charge. En utilisant des variables il est facile de voir du premier coup quelles fonctionnalités sont activées pour vous assurer que vous n'avez pas des fonctionnalités dans un noyau qui manquent dans un autre.

Cependant, nous ne parlons pas de la personnalisation du disque de ram initial. Vous devrez sans doute modifier le disque de ram initial sur les machines qui utilisent un noyau personnalisé, puisque certains modules attendus peuvent ne pas être disponibles.

3.3 L'API de création d'images du système Guix

Historiquement, le système Guix est centré sur une structure `operating-system`. Cette structure contient divers champs qui vont du chargeur d'amorçage et à la déclaration du noyau aux services à installer.

Les contraintes sur l'image peuvent grandement varier en fonction de la machine cible : une machine `x86_64` standard n'a pas les mêmes contraintes qu'un petit ordinateur mono-carte ARM comme le Pine64. Les fabricants de matériel imposent donc des formats d'image distincts avec des tailles et des emplacements de partition variables.

Pour créer des images convenables pour toutes ces machines, une nouvelle abstraction est nécessaire : c'est le but de l'enregistrement `image`. Cet enregistrement contient toutes les informations requises pour être transformé en une image complète, qui peut être directement démarrée sur une machine cible.

```
(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)))
```

Cet enregistrement contient le système d'exploitation à instancier. Le champ `format` défini le type d'image et peut être `efi-raw`, `qcow2` ou `iso9660` par exemple. Plus tard, on prévoit de l'étendre à `docker` et aux autres types d'images.

Un nouveau répertoire dans les sources de Guix est dédié aux définitions des images. Pour l'instant il y a quatre fichiers :

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

Regardons le fichier `pine64.scm`. Il contient la variable `pine64-barebones-os` qui est une définition minimale d'un système d'exploitation dédié à la carte **Pine A64 LTS**.

```
(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))))
```

Les champs `kernel` et `bootloader` pointent vers les paquets dédiés à cette carte.

Ci-dessous, la variable `pine64-image-type` est ainsi définie.

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

Elle utilise un enregistrement dont nous n'avons pas encore parlé, l'enregistrement `image-type`, défini de cette façon :

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

Le but principal de cet enregistrement est d'associer un nom à une procédure transformant un `operating-system` en une image. Pour comprendre pourquoi c'est nécessaire, voyons la commande produisant une image à partir d'un fichier de configuration de type `operating-system` :

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

Cette commande demande une configuration de type `operating-system` mais comment indiquer que l'on veut cibler une carte Pine64 ? Nous devons fournir l'information supplémentaire, `image-type`, en passant le drapeau `--image-type` ou `-t`, de cette manière :

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

Ce paramètre `image-type` pointe vers le `pine64-image-type` défini plus haut. Ainsi, la déclaration `operating-system` dans `my-os.scm` se verra appliquée la procédure `[cut image-with-os arm64-disk-image <>)` pour la transformer en une image.

L'image qui en résulte ressemble à ceci :

```
(image
 (format 'disk-image)
 (target "aarch64-linux-gnu")
 (operating-system my-os)
 (partitions
```

```
    (list (partition
           (inherit root-partition)
           (offset root-offset)))))
```

qui ajoute l'objet `operating-system` défini dans `my-os.scm` à l'enregistrement `arm64-disk-image`.

Mais assez de cette folie. Qu'est-ce que cette API pour les images apporte aux utilisateurs et utilisatrices ?

On peut lancer :

```
mathieu@cervin:~$ guix system --list-image-types
Les types d'image disponibles sont :

   - unmatched-raw
   - rock64-raw
   - pinebook-pro-raw
   - pine64-raw
   - novena-raw
   - hurd-raw
   - hurd-qcow2
   - qcow2
   - iso9660
   - uncompressed-iso9660
   - tarball
   - efi-raw
   - mbr-raw
   - docker
   - wsl2
   - raw-with-offset
   - efi32-raw
```

et en écrivant un fichier de type `operating-system` basé sur `pine64-barebones-os`, vous pouvez personnaliser votre image selon vos préférences dasn un fichier (`my-pine-os.scm`) de cette manière :

```
(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)))))
```

lancez :

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

ou bien,

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

pour récupérer une image que vous pouvez écrire sur un disque dur pour démarrer dessus.

Sans rien changer à `my-hurd-os.scm`, en appelant :

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

vous aurez une image QEMU pour le Hurd à la place.

3.4 Utiliser des clés de sécurité

L'utilisation de clés de sécurité peut améliorer votre sécurité en fournissant une seconde source d'authentification qui ne peut pas être facilement volée ni copiée, au moins pour les adversaires à distance (quelque chose que vous possédez) de votre secret principal (une phrase de passe — quelque chose que vous connaissez), ce qui réduit les risques de vol d'identité.

L'exemple de configuration détaillée plus bas montre la configuration minimale dont vous avez besoin sur votre système Guix pour permettre l'utilisation d'une clé de sécurité Yubico. Nous espérons que la configuration puisse être utile pour d'autres clés de sécurité aussi, avec quelques ajustements.

3.4.1 Configuration pour l'utiliser comme authentification à double facteur (2FA)

Pour être utilisable, les règles udev du systèmes doivent être étendues avec des règles spécifiques à la clé. Ce qui suit montre comment étendre vos règles udev avec le fichier de règles `lib/udev/rules.d/70-u2f.rules` fournit par le paquet `libfido2` du module `(gnu packages security-token)` et ajouter votre utilisateur au groupe '`"plugdev"`' qu'il utilise :

```
(use-package-modules ... security-token ...)
...
(operating-system
 ...
 (users (cons* (user-account
                (name "your-user")
                (group "users")
                (supplementary-groups
'("wheel" "netdev" "audio" "video"
                   "plugdev"))           ;<- added system group
                (home-directory "/home/your-user"))
               %base-user-accounts))
 ...
 (services
  (cons*
   ...
   (udev-rules-service 'fido2 libfido2 #:groups '("plugdev")))))
```

Après la reconfiguration de votre système et vous être authentifié dans votre session graphique pour que le nouveau groupe prenne effet pour votre utilisateur, vous pouvez vérifier que la clé est utilisable en exécutant :

```
guix shell ungoogled-chromium -- chromium chrome://settings/securityKeys
```

et en validant que la clé de sécurité peut être remise à zéro via le menu « réinitialiser votre clé de sécurité ». Si cela fonctionne, bravo, votre clé de sécurité est prête à être utilisée avec les applications qui prennent en charge l'authentification à double facteur (2FA).

3.4.2 Désactiver la génération de code OTP pour une Yubikey

Si vous utilisez une clé de sécurité Yubikey et que vous n'aimez pas les mauvais codes OTP qu'il génère lorsque vous touchez la clé par accident (p. ex. en vous faisant passer pour un spammer dans le canal '`#guix`' alors que vous discutez depuis votre client IRC préféré !), vous pouvez les désactiver avec la commande `ykman` suivante :

```
guix shell python-yubikey-manager -- ykman config usb --force --disable OTP
```

Autrement, vous pouvez utiliser la commande `ykman-gui` fournie par le paquet `yubikey-manager-qt` et soit désactiver complètement l'application '`OTP`' pour l'interface USB ou, depuis la vue '`Applications -> OTP`', supprimer la configuration du slot 1, qui est préconfiguré avec l'application Yubico OTP.

3.4.3 Demander une Yubikey pour ouvrir une base de données KeePassXC

Le gestionnaire de mots de passe KeePassXC prend en charge les Yubikeys, mais cela nécessite d'installer des règles udev pour votre système Guix et de configurer l'application OTP Yubico sur la clé.

Le fichier de règles udev nécessaire provient du paquet `yubikey-personalization` et peut être installé de cette manière :

```
(use-package-modules ... security-token ...)
...
(operating-system
 ...
 (services
  (cons*
   ...
   (udev-rules-service 'yubikey yubikey-personalization))))
```

Après avoir reconfiguré votre système (et avoir reconnecté votre Yubikey), vous voudrez ensuite configurer l'application de défi/réponse OTP do votre Yubikey sur le slot 2, qui est utilisé par KeePassXC. C'est facile à faire via l'outil de configuration graphique Yubikey Manager, qui peut s'invoquer de cette manière :

```
guix shell yubikey-manager-qt -- ykman-gui
```

Tout d'abord, assurez-vous d'avoir activé '`OTP`' dans l'onglet '`Interfaces`', puis rendez-vous dans '`Applications -> OTP`' et cliquez sur le bouton '`Configure`' sous la section '`Long Touch (Slot 2)`'. Choisissez '`Challenge-response`', saisissez ou générez une clé secrète, puis cliquez sur le bouton '`Finish`'. Si vous avez une seconde Yubikey que vous voulez utiliser en réserve, vous devriez la configurer de la même manière avec la *même* clé secrète.

Votre Yubikey devrait maintenant être détectée par KeePassXC. Elle peut être ajoutée à la base de données en se rendant dans le menu '`Base de données -> Sécurité de la base de données...`' de KeePassXC, puis en cliquant sur le bouton '`Ajouter une autre protection...`' puis '`Ajouter une question-réponse`', en choisissant la clé de sécurité dans le menu déroulant et en cliquant sur le bouton '`OK`' pour terminer la configuration.

3.5 Tâche mcron pour le DNS dynamique

Si votre FAI (fournisseur d'accès à internet) ne fournit que des adresse IP dynamiques, il peut être utile de mettre en place un service DNS (Domain Name System) dynamique (aussi appelé DDNS (Dynamic DNS)) pour associer un nom d'hôte statique à une adresse IP publique mais dynamique (qui change souvent). Il y a plusieurs services existants qui peuvent être utilisés pour cela ; dans la tâche mcron suivante, nous utilisons DuckDNS (`https://duckdns.org`). Cela devrait aussi fonctionner avec d'autres services DNS dynamiques qui fournissent une interface similaire pour mettre à jour l'adresse IP, comme `https://freedns.afraid.org/`, avec quelques petits ajustements.

La tâche mcron est fournie plus bas, où *DOMAINE* doit être remplacé par votre propre préfixe de domaine, et le jeton fournit par DuckDNS associé à *DOMAINE* est à ajouter au fichier `/etc/duckdns/`*DOMAINE*`.token`.

```
(define duckdns-job
  ;; Met à jour l'IP du domaine toutes les 5 minutes.
  #~(job '(next-minute (range 0 60 5))
 #$(program-file
            "duckdns-update"
            (with-extensions (list guile-gnutls) ;requis par (web client)
              #~(begin
                   (use-modules (ice-9 textual-ports)
                                (web client))
                   (let ((token (string-trim-both
                                 (call-with-input-file "/etc/duckdns/DOMAINE.token"
                                   get-string-all)))
                         (query-template (string-append "https://www.duckdns.org/"
                                                        "update?domains=DOMAINE"
                                                        "&token=~a&ip=")))
                     (http-get (format #f query-template token))))))
          "duckdns-update"
          #:user "nobody"))
```

La tâche a ensuite besoin d'être ajoutée à la liste des tâches mcron de votre système, avec quelque chose comme cela :

```
(operating-system
 (services
  (cons* (service mcron-service-type
           (mcron-configuration
             (jobs (list duckdns-job ...))))
         ...
         %base-services)))
```

3.6 Se connecter à un VPN Wireguard

Pour se connecter à un serveur VPN Wireguard, il faut que le module du noyau soit chargé en mémoire et qu'un paquet fournissant des outils de réseau le prenne en charge (par exemple, `wireguard-tools` ou `network-manager`).

Voici un exemple de configuration pour Linux-Libre < 5.6, où le module est hors de l'arborescence des sources et doit être chargé manuellement–les révisions suivantes du noyau l'ont intégré et n'ont donc pas besoin d'une telle 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)))
```

Après avoir reconfiguré et redémarré votre système, vous pouvez utiliser les outils Wireguard ou NetworkManager pour vous connecter à un serveur VPN.

3.6.1 Utilisation des outils Wireguard

Pour tester votre configuration Wireguard, vous pouvez utiliser `wg-quick`. Donnez-lui simplement un fichier de configuration : `wg-quick up ./wg0.conf`, ou placez ce fichier dans `/etc/wireguard` et lancez `wg-quick up wg0` à la place.

> **Remarque:** Soyez averti que l'auteur a décrit cette commande comme un : « [...] script bash écrit à la va-vite [...] ».

3.6.2 En utilisant NetworkManager

Grâce à la prise en charge de NetworkManager pour Wireguard, nous pouvons nous connecter à notre VPN en utilisant la commande `nmcli`. Jusqu'ici, ce guide suppose que vous utilisez le service Network Manager fourni par `%desktop-services`. Dans le cas contraire, vous devez ajuster votre liste de services pour charger `network-manager-service-type` et reconfigurer votre système Guix.

Pour importer votre configuration VPN, exécutez la commande d'import de nmcli :

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

Cela va créer un fichier de configuration dans `/etc/NetworkManager/wg0.nmconnection`.
Ensuite connectez-vous au serveur Wireguard :

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

Par défaut, NetworkManager se connectera automatiquement au démarrage du système.
Pour changer ce comportement vous devez modifier votre configuration :

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

Pour des informations plus spécifiques sur NetworkManager et wireguard voir ce billet par thaller (`https://blogs.gnome.org/thaller/2019/03/15/wireguard-in-networkmanager/`).

3.7 Personnaliser un gestionnaire de fenêtres

3.7.1 StumpWM

Vous pouvez installer StumpWM sur un système Guix en ajoutant `stumwm` et éventuellement `` `(,stumpwm "lib") `` dans les paquets du fichier de système d'exploitation, p. ex. `/etc/config.scm`.

Voici un exemple de configuration :

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

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

Par défaut StumpWM utilise les polices X11, qui peuvent être petites ou pixelisées sur votre système. Vous pouvez corriger cela en installant le module Lisp pour StumpWM `sbcl-ttf-fonts`, en l'ajoutant aux paquets de votre système :

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

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

Ensuite vous devrez ajouter le code suivant à au fichier de configuration de StumpWM `~/.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.sexp"))
(xft:cache-fonts)
(set-font (make-instance 'xft:font :family "DejaVu Sans Mono"
                                   :subfamily "Book" :size 11))
```

3.7.2 Verrouillage de session

En fonction de votre environnement, le verrouillage de l'écran peut être inclus, ou vous devrez le configurer vous-même. La fonctionnalité est souvent intégrée aux environnements de bureau comme GNOME ou KDE. Si vous utilisez un gestionnaire de fenêtre comme StumpWM ou EXWM, vous devrez sans doute le configurer vous-même.

3.7.2.1 Xorg

Si vous utilisez Xorg, vous pouvez utiliser l'utilitaire xss-lock (`https://www.mankier.com/1/xss-lock`) pour verrouiller votre session. xss-lock est lancé par DPMS qui est détecté et activé automatiquement par Xorg 1.8 si ACPI est aussi activé à l'exécution dans le noyau.

Pour utiliser xss-lock, vous pouvez simplement l'exécuter et le laisser tourner en tache de fond avant de démarrer votre gestionnaire de fenêtre, par exemple dans votre `~/.xsession` :

```
xss-lock -- slock &
exec stumpwm
```

Dans cet exemple, xss-lock utilise `slock` pour effectivement verrouiller l'écran quand il pense que c'est nécessaire, comme lorsque vous mettez votre machine en veille.

Pour que slock puisse verrouiller l'écran de la session graphique, il doit être en setuid-root pour qu'il puisse authentifier les utilisateurs, et il a besoin d'un service PAM. On peut y arriver en ajoutant le service suivant dans notre `config.scm` :

```
(service screen-locker-service-type
         (screen-locker-configuration
          (name "slock")
          (program (file-append slock "/bin/slock"))))
```

Si vous verrouillez l'écran manuellement, p. ex. en appelant slock directement si vous voulez verrouiller l'écran sans mettre l'ordinateur en veille, il vaut mieux notifier xss-lock pour éviter la confusion. Vous pouvez faire cela en exécutant `xset s activate` juste avant d'exécuter slock.

3.8 Lancer Guix sur un serveur Linode

Pour lancer Guix sur un serveur hébergé par Linode (`https://www.linode.com`), commencez par un serveur Debian recommandé. Nous vous recommandons d'utiliser la distribution par défaut pour amorcer Guix. Créez vos clés SSH.

```
ssh-keygen
```

Assurez-vous d'ajouter votre clé SSH pour vous connecter facilement sur le serveur distant. C'est facilité par l'interface graphique de Linode. Allez sur votre profil et cliquez sur le bouton pour ajouter une clé SSH. Copiez la sortie de :

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

Éteignez votre Linode.

Dans l'onglet de stockage du Linode, modifiez la taille du disque Debian pour qu'il soit plus petit. Nous recommandons 30 Go d'espace libre. Ensuite, cliquez sur « Ajouter un disque » et remplissez le formulaire de cette manière :

- Label: "Guix"
- Filesystem: ext4
- Donnez-lui la taille restante

Dans l'onglet de configuration, cliquez sur « modifier » sur le profil Debian par défaut. Dans « Block Device Assignment » cliquez sur « Add a Device ». Il devrait apparaître en tant que `/dev/sdc` et vous pouvez sélectionner le disque « Guix ». Sauvegardez les changements.

Maintenant « Add a Configuration », avec ce qui suit :

- Label: Guix
- Kernel: GRUB 2 (c'est à la toute fin ! Cette étape est **IMPORTANTE !**)
- Périphériques blocs assignés :
- `/dev/sda` : Guix
- `/dev/sdb` : swap
- Périphérique racine : `/dev/sda`
- Désactivez tous les programmes d'aide pour les systèmes de fichiers et le démarrage

Maintenant démarrez le serveur avec la configuration Debian. Une fois lancé, connectez vous en ssh au serveur avec `ssh root@<IP-de-votre-serveur-ici>`. (Vous pouvez trouver l'adresse IP de votre serveur dans la section résumé de Linode). Maintenant vous pouvez lancer les étapes d'installation de voir Section "Installation binaire" dans *GNU Guix* :

```
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
```

Maintenant il est temps d'écrire une configuration pour le serveur. Voici ce que vous devrez obligatoirement écrire, en plus de vos propres configurations. Enregistrez le fichier avec le nom `guix-config.scm`.

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

(operating-system
  (host-name "my-server")
  (timezone "America/New_York")
  (locale "en_US.UTF-8")
  ;; Ce code va générer un grub.cfg
  ;; sans installer le chargeur d'amorçage grub sur le disque.
  (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"    ; Requis pour trouver le disque
                        %base-initrd-modules))

  (users (cons (user-account
                (name "janedoe")
                (group "users")
                ;; Ajoute le compte au groupe « wheel »
                ;; pour en faire un sudoer.
                (supplementary-groups '("wheel"))
                (home-directory "/home/janedoe"))
               %base-user-accounts))

  (packages (cons* 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)))
```

Remplacez les champs suivants dans la configuration ci-dessus :

```
(host-name "my-server")       ; remplacez avec le nom de votre serveur
; si vous avez choisi un serveur Linode en dehors des U.S.,
; utilisez tzselect pour trouver le bon fuseau horaire
(timezone "America/New_York") ; remplacez le fuseau horaire si besoin
(name "janedoe")              ; remplacez avec votre nom d'utilisateur
("janedoe" ,(local-file "janedoe_rsa.pub")) ; remplacez par votre clé ssh
("root" ,(local-file "janedoe_rsa.pub")) ; remplacez par votre clé ssh
```

Cette dernière ligne vous permet de vous connecter au serveur en root et de créer le mot de passe initial de root (voir la note à la fin de cette recette sur la connexion en root). Après avoir fait cela, vous pouvez supprimer cette ligne de votre configuration et reconfigurer pour empêcher la connexion directe en root.

Copiez votre clé ssh publique (ex : `~/.ssh/id_rsa.pub`) dans `<votre-nom-d'utilisateur>_rsa.pub` et ajoutez votre `guix-config.scm` au même répertoire. Dans un nouveau terminal lancez ces commandes.

```
sftp root@<adresse IP du serveur distant>
put /path/to/files/<username>_rsa.pub .
put /path/to/files/guix-config.scm .
```

Dans votre premier terminal, montez le disque guix :

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

À cause de la manière dont nous avons paramétré la section du chargeur d'amorçage dans le fichier guix-config.scm, nous installons seulement notre fichier de configuration grub. Donc on doit copier certains fichiers GRUB déjà installés sur le système Debian :

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

Maintenant initialisez l'installation de Guix :

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

Ok, éteignez maintenant le serveur ! Depuis la console Linode, démarrez et choisissez « Guix ».

Une fois démarré, vous devriez pouvoir vous connecter en SSH ! (La configuration du serveur aura cependant changé). Vous pouvez rencontrer une erreur de ce type :

```
$ 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.
```

Vous pouvez soit supprimer `~/.ssh/known_hosts`, soit supprimer la ligne qui pose problème, qui commence par l'adresse IP de votre serveur.

Assurez-vous de configurer votre mot de passe et celui de root.

```
ssh root@<remote ip address>
passwd  ; pour le mot de passe root
passwd <username> ; pour le mot de passe utilisateur
```

Il se peut que vous ne puissiez pas lancer les commandes précédentes si vous n'arrivez pas à vous connecter à distance via SSH, auquel cas vous devrez peut-être configurer vos mot de passes utilisateurs et root en cliquant sur « Launch Console » dans votre espace Linode. Choisissez « Glish » au lieu de « Weblish ». Maintenant vous devriez pouvoir vous connecter en ssh à la machine.

Hourra ! Maintenant vous pouvez étendre le serveur, supprimer le disque Debian et redimensionner celui de Guix. Bravo !

Au fait, si vous sauvegardez le résultat dans une image disque maintenant, vous pourrez plus facilement démarrer de nouvelles images de guix ! Vous devrez peut-être réduire la taille de l'image Guix à 6144 Mo, pour la sauvegarder en tant qu'image. Ensuite vous pouvez redimensionner la partition à la taille maximum.

3.9 Lancer Guix sur un serveur Kimsufi

Pour lancer Guix sur un serveur hébergé par Kimsufi (`https://www.kimsufi.com/`), cliquez sur l'onglet netboot puis choisissez rescue64-pro et redémarrez.

OVH vous enverra un courriel avec les identifiants requis pour vous connecter en ssh à un système Debian.

Maintenant vous pouvez exécuter les étapes de « installer guix de voir Section "Installation binaire" dans *GNU Guix* » :

```
wget https://git.savannah.gnu.org/cgit/guix.git/plain/etc/guix-install.sh
chmod +x guix-install.sh
./guix-install.sh
guix pull
```

Partitionnez les lecteurs et formatez-les, en commençant par arrêter la grappe raid :

```
mdadm --stop /dev/md127
mdadm --zero-superblock /dev/sda2 /dev/sdb2
```

Ensuite, effacez les disques et créez les partitions. Nous allons créer une grappe RAID 1.

```
wipefs -a /dev/sda
wipefs -a /dev/sdb

parted /dev/sda --align=opt -s -m -- mklabel gpt
parted /dev/sda --align=opt -s -m -- \
 mkpart bios_grub 1049kb 512MiB \
 set 1 bios_grub on
parted /dev/sda --align=opt -s -m -- \
 mkpart primary 512MiB -512MiB
 set 2 raid on
parted /dev/sda --align=opt -s -m -- mkpart primary linux-swap 512MiB 100%

parted /dev/sdb --align=opt -s -m -- mklabel gpt
parted /dev/sdb --align=opt -s -m -- \
     mkpart bios_grub 1049kb 512MiB \
     set 1 bios_grub on
parted /dev/sdb --align=opt -s -m -- \
     mkpart primary 512MiB -512MiB \
     set 2 raid on
parted /dev/sdb --align=opt -s -m -- mkpart primary linux-swap 512MiB 100%
```

Créez la grappe :

```
mdadm --create /dev/md127 --level=1 --raid-disks=2 \
```

```
--metadata=0.90 /dev/sda2 /dev/sdb2
```

Maintenant, créez les systèmes de fichiers sur les bonnes partitions, en commençant par la partition de démarrage :

```
mkfs.ext4  /dev/sda1
mkfs.ext4  /dev/sdb1
```

Puis la partition racine :

```
mkfs.ext4 /dev/md127
```

Initialisez les partitions d'échange :

```
mkswap /dev/sda3
swapon /dev/sda3
mkswap /dev/sdb3
swapon /dev/sdb3
```

Montez le disque guix :

```
mkdir /mnt/guix
mount /dev/md127 /mnt/guix
```

Maintenant vous pouvez écrire une déclaration de système d'exploitation dans un fichier `os.scm`. Voici un exemple :

```
(use-modules (gnu) (guix))
(use-service-modules networking ssh vpn virtualization sysctl admin mcron)
(use-package-modules ssh tls tmux vpn virtualization)

(operating-system
  (host-name "kimsufi")

  (bootloader (bootloader-configuration
       (bootloader grub-bootloader)
       (targets (list "/dev/sda" "/dev/sdb"))
       (terminal-outputs '(console))))

  ;; Ajouter un module noyau pour le RAID-1 (appelé « miroir »).
  (initrd-modules (cons* "raid1"  %base-initrd-modules))

  (mapped-devices
   (list (mapped-device
          (source (list "/dev/sda2" "/dev/sdb2"))
          (target "/dev/md127")
          (type raid-device-mapping))))

  (swap-devices
   (list (swap-space
          (target "/dev/sda3"))
         (swap-space
          (target "/dev/sdb3"))))

  (issue
```

```
   ;; Contenu par défaut de /etc/issue.
   "\
Ceci est le système GNU sur Kimsufi.  Bienvenue.\n")

  (file-systems (cons* (file-system
         (mount-point "/")
         (device "/dev/md127")
         (type "ext4")
         (dependencies mapped-devices))
       %base-file-systems))

  (users (cons (user-account
        (name "guix")
        (comment "guix")
        (group "users")
        (supplementary-groups '("wheel"))
        (home-directory "/home/guix"))
       %base-user-accounts))

  (sudoers-file
   (plain-file "sudoers" "\
root ALL=(ALL) ALL
%wheel ALL=(ALL) ALL
guix ALL=(ALL) NOPASSWD:ALL\n"))

 ;; Paquets installés sur tout le système.
  (packages (cons* tmux gnutls wireguard-tools %base-packages))
  (services
   (cons*
    (service static-networking-service-type
     (list (static-networking
    (addresses (list (network-address
      (device "enp3s0")
      (value "adresse-ip-du-serveur/24"))))
    (routes (list (network-route
   (destination "default")
   (gateway "passerelle-du-serveur"))))
    (name-servers '("213.186.33.99")))))

    (service unattended-upgrade-service-type)

    (service openssh-service-type
     (openssh-configuration
      (openssh openssh-sans-x)
      (permit-root-login #f)
      (authorized-keys
       `(("guix" ,(plain-file "nom-de-clé-ssh.pub"
```

```
                                          "contenu-de-clé-ssh"))))))
      (modify-services %base-services
        (sysctl-service-type
         config =>
         (sysctl-configuration
   (settings (append '(("net.ipv6.conf.all.autoconf" . "0")
      ("net.ipv6.conf.all.accept_ra" . "0"))
    %default-sysctl-settings))))))))
```

N'oubliez pas de modifier les variables *adresse-ip-du-serveur*, *passerelle-du-serveur*, *nom-de-clé-ssh* et *contenu-de-clé-ssh* avec vos propres valeurs.

La passerelle est la dernière IP utilisable dans votre bloc donc si vous avez un serveur avec l'IP '`37.187.79.10`', sa passerelle sera '`37.187.79.254`'.

Transférez votre déclaration de système d'exploitation `os.scm` sur le serveur via les commandes `scp` ou `sftp`.

Maintenant, tout ce qu'il reste à faire est d'installer Guix avec `guix system init` et redémarrer.

Cependant nous devons d'abord paramétrer un chroot, car la partition racine du système de secours est monté sur une partition aufs et si vous essayez d'installer Guix cela ne marchera pas à l'étape d'installation de GRUB. Il se plaindra du chemin canonique de « aufs ».

Installez les paquets qui seront utilisés dans le chroot :

```
guix install bash-static parted util-linux-with-udev coreutils guix
```

Ensuite, exécutez ce qui suit pour créer les répertoires nécessaires au chroot :

```
cd /mnt && \
mkdir -p bin etc gnu/store root/.guix-profile/ root/.config/guix/current \
  var/guix proc sys dev
```

Copiez le resolv.conf hôte dans le chroot :

```
cp /etc/resolv.conf etc/
```

Montez les périphériques de type bloc, le dépôt et sa base de données et la configuration guix actuelle :

```
mount --rbind /proc /mnt/proc
mount --rbind /sys /mnt/sys
mount --rbind /dev /mnt/dev
mount --rbind /var/guix/ var/guix/
mount --rbind /gnu/store gnu/store/
mount --rbind /root/.config/ root/.config/
mount --rbind /root/.guix-profile/bin/ bin
mount --rbind /root/.guix-profile root/.guix-profile/
```

Entrez dans le chroot sur /mnt et installez le système :

```
chroot /mnt/ /bin/bash

guix system init /root/os.scm /guix
```

Enfin, à partir de l'interface utilisateur web, modifiez '`netboot`' en '`boot to disk`' et redémarrez (également depuis l'interface web).

Attendez quelques minutes et essayez de vous connecter en ssh avec `ssh guix@adresse-ip-du-serveur> -i chemin-vers-votre-clé-ssh`

Votre système Guix devrait être opérationnel sur Kimsufi. Félicitations !

3.10 Mettre en place un montage dupliqué

Pour dupliquer le montage d'un système de fichier (*bind mount*), on doit d'abord ajouter quelques définitions avant la section `operating-system` de la définition de système d'exploitation. Dans cet exemple nous allons dupliquer le montage d'un dossier d'un disque dur vers `/tmp`, pour éviter d'épuiser le SSD principal, sans dédier une partition entière à `/tmp`.

Déjà, le disque source qui héberge de dossier dont nous voulons dupliquer le montage doit être défini, pour que le montage dupliqué puisse en dépendre.

```
(define source-drive ;; vous pouvez nommer « source-drive » comme vous le souhaitez.
   (file-system
    (device (uuid "indiquez l'UUID ici"))
    (mount-point "/chemin-vers-le-disque-dur")
    (type "ext4"))) ;Assurez-vous d'indiquer le bon type pour la partition
```

Le dossier source doit aussi être défini, pour que guix sache qu'il ne s'agit pas d'un périphérique bloc, mais d'un dossier.

```
;; vous pouvez nommer « source-directory » comme vous le souhaitez.
(define (%source-directory) "/chemin-vers-le-disque-dur/tmp")
```

Enfin, dans la définition `file-systems`, on doit ajouter le montage lui-même.

```
(file-systems (cons*

               ...<d'autres montages omis pour rester concis>...

              ;; Doit correspondre au nom que vous avez donné au disque source
              ;; dans la définition précédente.
               source-drive

               (file-system
               ;; Assurez-vous que « source-directory » corresponde à
               ;; la définition précédente.
                (device (%source-directory))
                (mount-point "/tmp")
               ;; On monte un dossier, pas une partition, donc le montage est
               ;; de type « none »
                (type "none")
                (flags '(bind-mount))
               ;; Assurez-vous que « source-drive » corresponde au nom de
               ;; la variable pour le disque.
                (dependencies (list source-drive))
                )

                ...<d'autres montages omis pour rester concis>...
```

```
))
```

3.11 Récupérer des substituts via Tor

Le démon Guix peut utiliser un mandataire HTTP pour récupérer des substituts. Nous le configurons ici pour les récupérer par Tor.

> **Attention:** *Tout* le trafic du démon de passera *pas* par Tor ! Seuls HTTP/HTTPS passer par le mandataire ; les connexions FTP, avec le protocol Git, SSH, etc, passeront toujours par le réseau en clair. De nouveau, cette configuration n'est pas parfaite et une partie de votre trafic ne sera pas routé par Tor du tout. Utilisez-la à vos risques et périls.
>
> Remarquez aussi que la procédure décrite ici ne s'applique qu'à la substitution de paquets. Lorsque vous mettez à jour la distribution avec `guix pull`, vous aurez encore besoin de `torsocks` si vous voulez router la connexion vers les serveurs de dépôts git à travers Tor.

Le serveur de substitut de Guix est disponible sur un service Onion. Si vous voulez l'utiliser pour récupérer des substituts par Tor, configurez votre système de cette manière :

```
(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)
                     ;; service Onion de ci.guix.gnu.org
                      (substitute-urls
                       "\
https://4zwzi66wwdaalbhgnix55ea3ab4pvvw66ll2ow53kjub6se4q2bclcyd.onion")
                      (http-proxy "http://localhost:9250")))))))
```

Cela fera tourner le processus tor et fournira un tunnel HTTP CONNECT qui sera utilisé par `guix-daemon`. Le démon peut utiliser d'autres protocoles que HTTP(S) pour récupérer des ressources distantes. Les requêtes utilisant ces protocoles ne passeront pas par Tor puisqu'il s'agit d'un tunnel HTTP uniquement. Remarquez que `substitutes-urls` doit utiliser HTTPS et non HTTP, sinon ça ne fonctionne pas. C'est une limite du tunnel de Tor ; vous voudrez peut-être utiliser `privoxy` à la place pour éviter ces limites.

Si vous ne voulez pas toute le temps récupérer des substituts à travers Tor mais l'utiliser seulement de temps en temps, alors ne modifiez pas l'objet `guix-configuration`. Lorsque vous voulez récupérer un substitut par le tunnel Tor, lancez :

```
sudo herd set-http-proxy guix-daemon http://localhost:9250
guix build \
  --substitute-urls=https://4zwzi66wwdaalbhgnix55ea3ab4pvvw66ll2ow53kjub6se4q2bclcyd.o
```

3.12 Configurer NGINX avec Lua

Les fonctionnalités de NGINX peuvent être étendues avec des scripts Lua.

Guix fournit un service NGINX qui est capable de charger des modules et des paquets Lua spécifiques, et de répondre aux requêtes en évaluant des scripts Lua.

L'exemple suivant montre une définition de système avec une configuration qui évalue le script Lua `index.lua` lors d'une requête HTTP à `http://localhost/hello` :

```
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")))))))
```

3.13 Serveur de musique avec l'audio bluetooth

MPD, le démon lecteur de musique, est une application serveur flexible pour jouer de la musique. Les programmes clients sur différentes machines du réseau — un téléphone portable, un ordinateur portable, un ordinateur de bureau — peuvent s'y connecter pour contrôler la lecture de fichiers audio de votre collection musicale locale. MPD décode les fichiers audio et les joue sur une ou plusieurs sorties.

Par défaut MPD jouera sur le périphérique audio par défaut. Dans l'exemple ci-dessous nous rendons les choses un peu plus intéressantes en configurant un serveur de musique sans affichage. Il n'y aura pas d'interface graphique, pas de démon Pulseaudio, ni de sortie audio locale. Au lieu de cela, nous configurons MPD avec deux sorties : un haut-parleur bluetooth et un serveur web pour servir des flux audio à n'importe quel lecture de musique.

Le Bluetooth est souvent frustrant à configurer. Vous devrez appairer vos périphériques Bluetooth et vous assurer que le périphérique se connecte automatiquement dès qu'il est branché. Le service système Bluetooth renvoyé par la procédure `bluetooth-service` fournit l'infrastructure requise pour cette configuration.

Reconfigurez votre système avec au moins les services et les paquets suivants :

```
(operating-system
  ;; ...
  (packages (cons* bluez bluez-alsa
                   %base-packages))
  (services
   ;; ...
   (dbus-service #:services (list bluez-alsa))
   (bluetooth-service #:auto-enable? #t)))
```

Démarrez le service `bluetooth` puis utilisez `bluetoothctl` pour scanner les périphériques Bluetooth. Essayez d'identifier votre haut-parleur Bluetooth et choisissez son ID de périphérique dans la liste de périphériques qui est indubitablement polluée par une armée de gadgets connectés chez votre voisin. Vous ne devrez le faire qu'une seule fois :

```
$ bluetoothctl
[NEW] Controller 00:11:22:33:95:7F BlueZ 5.40 [default]

[bluetooth]# power on
[bluetooth]# Changing power on succeeded

[bluetooth]# agent on
[bluetooth]# Agent registered

[bluetooth]# default-agent
```

```
[bluetooth]# Default agent request successful

[bluetooth]# scan on
[bluetooth]# Discovery started
[CHG] Controller 00:11:22:33:95:7F Discovering: yes
[NEW] Device AA:BB:CC:A4:AA:CD My Bluetooth Speaker
[NEW] Device 44:44:FF:2A:20:DC My Neighbor's TV
...

[bluetooth]# pair AA:BB:CC:A4:AA:CD
Attempting to pair with AA:BB:CC:A4:AA:CD
[CHG] Device AA:BB:CC:A4:AA:CD Connected: yes

[My Bluetooth Speaker]# [CHG] Device AA:BB:CC:A4:AA:CD UUIDs: 0000110b-0000-1000-8000-
[CHG] Device AA:BB:CC:A4:AA:CD UUIDs: 0000110c-0000-1000-8000-00xxxxxxxxxx
[CHG] Device AA:BB:CC:A4:AA:CD UUIDs: 0000110e-0000-1000-8000-00xxxxxxxxxx
[CHG] Device AA:BB:CC:A4:AA:CD Paired: yes
Pairing successful

[CHG] Device AA:BB:CC:A4:AA:CD Connected: no

[bluetooth]#
[bluetooth]# trust AA:BB:CC:A4:AA:CD
[bluetooth]# [CHG] Device AA:BB:CC:A4:AA:CD Trusted: yes
Changing AA:BB:CC:A4:AA:CD trust succeeded

[bluetooth]#
[bluetooth]# connect AA:BB:CC:A4:AA:CD
Attempting to connect to AA:BB:CC:A4:AA:CD
[bluetooth]# [CHG] Device AA:BB:CC:A4:AA:CD RSSI: -63
[CHG] Device AA:BB:CC:A4:AA:CD Connected: yes
Connection successful

[My Bluetooth Speaker]# scan off
[CHG] Device AA:BB:CC:A4:AA:CD RSSI is nil
Discovery stopped
[CHG] Controller 00:11:22:33:95:7F Discovering: no
```

Félicitations, vous pouvez maintenant vous connecter automatiquement à votre haut-parleur Bluetooth !

Il est maintenant temps de configurer ALSA pour utiliser le module Bluetooth *bluealsa*, pour que vous puissiez définir un périphérique pcm ALSA correspondant à votre haut-parleur Bluetooth. Un serveur sans affichage utilisant *bluealsa* avec un périphérique fixe est probablement plus facile à configurer que Pulseaudio et son comportement de changement de flux. Nous configurons ALSA en créant un `alsa-configuration` personnalisé pour le service `alsa-service-type`. La configuration déclarera un type `pcm bluealsa` du module

`bluealsa` fournit par le paquet `bluez-alsa`, puis définira un périphérique `pcm` de ce type pour le haut-parleur Bluetooth.

Tout ce qui reste à faire est de faire en sorte que MPD envoie les données audio à ce périphérique ALSA. Nous ajoutons aussi une sortie MPD secondaire qui rend les fichiers audio en lecture disponibles en streaming sur un serveur web sur le port 8080. Lorsque la sortie est activée, un appareil sur le réseau peut écouter le flux audio en se connectant avec n'importe quel lecteur multimédia au serveur HTTP sur le port 8080, indépendamment du statut du haut-parleur Bluetooth.

Voici les grandes lignes d'une déclaration `operating-system` qui devrait accomplir les tâches sus-mentionnées :

```
(use-modules (gnu))
(use-service-modules audio dbus sound #;... etc)
(use-package-modules audio linux #;... etc)
(operating-system
 ;; ...
  (packages (cons* bluez bluez-alsa
                   %base-packages))
  (services
  ;; ...
   (service mpd-service-type
            (mpd-configuration
             (user "your-username")
             (music-dir "/path/to/your/music")
             (address "192.168.178.20")
             (outputs (list (mpd-output
                             (type "alsa")
                             (name "MPD")
                             (extra-options
                             ;; Utilisez le même nom que dans la configuration
                             ;; ALSA plus bas.
                              '((device . "pcm.btspeaker"))))
                            (mpd-output
                             (type "httpd")
                             (name "streaming")
                             (enabled? #false)
                             (always-on? #true)
                             (tags? #true)
                             (mixer-type 'null)
                             (extra-options
                              '((encoder . "vorbis")
                                (port    . "8080")
                                (bind-to-address . "192.168.178.20")
                                (max-clients . "0") ;no limit
                                (quality . "5.0")
                                (format  . "44100:16:1")))))))))
   (dbus-service #:services (list bluez-alsa))
```

```
   (bluetooth-service #:auto-enable? #t)
   (service alsa-service-type
            (alsa-configuration
             (pulseaudio? #false);we don't need it
             (extra-options
              #~(string-append "\
# Déclare un type de périphérique audio Bluetooth \"bluealsa\" du module bluealsa
pcm_type.bluealsa {
    lib \""
#$(file-append bluez-alsa "/lib/alsa-lib/libasound_module_pcm_bluealsa.so") "\"
}

# Déclare un type de périphérique de contrôle \"bluealsa\" du même module
ctl_type.bluealsa {
    lib \""
#$(file-append bluez-alsa "/lib/alsa-lib/libasound_module_ctl_bluealsa.so") "\"
}

# Définie le périphérique Bluetooth audio.
pcm.btspeaker {
    type bluealsa
    device \"AA:BB:CC:A4:AA:CD\" # unique device identifier
    profile \"a2dp\"
}

# Définie un contrôleur associé.
ctl.btspeaker {
    type bluealsa
}
"))))))
```

Profitez de votre musique avec le client MPD de votre choix ou un lecteur multimédia capable de streamer en HTTP !

4 Conteneurs

Le noyau Linux fournit un certain nombre de dispositifs partagés qui sont disponibles pour les processus du système. Ces dispositifs sont entre autres un vue partagée du système de fichiers, des autres processus, des périphériques réseau, des identités de groupe et d'utilisateur et quelques autres. Depuis Linux 3.19 vous pouvez choisir de *départager* certains de ces dispositifs partagés pour des processus choisis, en leur fournissant (ainsi qu'à leurs processus enfant) une vue différente du système.

Un processus avec un espace de nom de montage (`mount`) départagé par exemple, a sa propre vue du système de fichier — il ne pourra voir que les répertoires qui ont été explicitement liés à son espace de nom de montage. Un processus avec son propre espace de nom de processus (`proc`) considérera qu'il est le seul processus lancé sur le système, avec le PID 1.

Guix utilise ces fonctionnalités du noyau pour fournir des environnement complètement isolés e même des conteneurs complets pour le système Guix, des machines virtuelles légères qui partagent le noyau de l'hôte. Cette fonctionnalité est particulièrement pratique si vous utilise Guix sur une distribution externe pour éviter les interférence avec les bibliothèques ou les fichiers de configuration externes disponibles sur l'ensemble du système.

4.1 Conteneurs Guix

La manière la plus simple de démarrer est d'utiliser `guix shell` avec l'option `--container`. Voir Section "Invoquer guix shell" dans *le manuel de référence de GNU Guix* pour la référence des options valides.

Le bout de code suivant démarre un shell minimal avec la plupart des espaces de noms départagés du système. Le répertoire de travail actuel est visible pour le processus, mais tout le reste du système de fichiers est indisponible. Cette isolation extrême peut être très utile si vous voulez écarter toute interférence des variables d'environnement, des bibliothèques installées globalement ou des fichiers de configuration.

```
guix shell --container
```

C'est un environnement vierge, aride et désolé. Vous trouverez que même GNU coreutils n'y est pas disponible, donc pour explorer cet environnement désert vous devrez utiliser les commandes internes au shell. Même le répertoire `/gnu/store` habituellement énorme est réduit à peau de chagrin.

```
$ echo /gnu/store/*
/gnu/store/...-gcc-10.3.0-lib
/gnu/store/...-glibc-2.33
/gnu/store/...-bash-static-5.1.8
/gnu/store/...-ncurses-6.2.20210619
/gnu/store/...-bash-5.1.8
/gnu/store/...-profile
/gnu/store/...-readline-8.1.1
```

Vous ne pouvez pas faire grand chose de plus dans un environnement comme celui-ci à part en sortir. Vous pouvez utiliser `Ctrl-D` ou `exit` pour quitter cet environnement shell limité.

Vous pouvez rendre d'autres répertoires disponibles à l'intérieur de l'environnement du conteneur. Utilisez `--expose=RÉPERTOIRE` pour créer un montage lié au répertoire donné en lecture-seule à l'intérieur du conteneur, ou utilisez `--share=RÉPERTOIRE` pour rendre cet emplacement inscriptible. Avec un argument de correspondance supplémentaire après le nom du répertoire vous pouvez contrôler le nom du répertoire lié dans le conteneur. Dans l'exemple suivant nous faisons correspondre `/etc` du système hôte à `/l/hôte/etc` dans un conteneur dans lequel GNU coreutils est installé.

```
$ guix shell --container --share=/etc=/l/hôte/etc coreutils
$ ls /l/hôte/etc
```

De même, vous pouvez empêcher le répertoire actuel d'être ajouté au conteneur avec l'option `--no-cwd`. Une autre bonne idée est de créer un répertoire dédié qui servira de répertoire personnel dans le conteneur et de démarrer le shell du conteneur dans ce répertoire.

Sur un système externe un environnement de conteneur peut être utilisé pour compiler un logiciel qui ne peut pas être lié aux bibliothèques du système ou avec la chaine d'outils du système. Un cas d'utilisation courant dans le contexte de la recherche est l'installation de paquets à partir d'une session R. En dehors de l'environnement du conteneur il est fort probable que la chaine de compilation externe et les bibliothèques systèmes incompatibles soient trouvés en premier, ce qui créer des binaires incompatibles qui ne peuvent pas être utilisés dans R. Dans un conteneur ce problème disparait car les bibliothèques du système et les executables ne sont simplement pas disponibles à cause de l'espace de nom `mount` départagé.

Prenons un manifeste complet pour fournir un environnement de développement confortable pour R :

```
(specifications->manifest
  (list "r-minimal"

        ;; paquets de base
        "bash-minimal"
        "glibc-locales"
        "nss-certs"

        ;; Outils en ligne de commande usuels, sans lesquels le conteneur est trop vid
        "coreutils"
        "grep"
        "which"
        "wget"
        "sed"

        ;; outils markdown pour R
        "pandoc"

        ;; Chaine d'outils et bibliothèques courantes pour « install.packages »
        "gcc-toolchain@10"
        "gfortran-toolchain"
        "gawk"
```

```
        "tar"
        "gzip"
        "unzip"
        "make"
        "cmake"
        "pkg-config"
        "cairo"
        "libxt"
        "openssl"
        "curl"
        "zlib"))
```

Utilisons cela pour lancer R dans un environnement de conteneur. Pour se simplifier la vie, nous partageons l'espace de nom `net` pour utiliser les interfaces réseau du système hôte. Maintenant nous pouvons construire des paquets R à partir des sources de la manière traditionnelle sans avoir à se soucier des incompatibilités d'ABI.

```
$ guix shell --container --network --manifest=manifest.scm -- R

R version 4.2.1 (2022-06-23) -- "Funny-Looking Kid"
Copyright (C) 2022 The R Foundation for Statistical Computing
...
> e <- Sys.getenv("GUIX_ENVIRONMENT")
> Sys.setenv(GIT_SSL_CAINFO=paste0(e, "/etc/ssl/certs/ca-certificates.crt"))
> Sys.setenv(SSL_CERT_FILE=paste0(e, "/etc/ssl/certs/ca-certificates.crt"))
> Sys.setenv(SSL_CERT_DIR=paste0(e, "/etc/ssl/certs"))
> install.packages("Cairo", lib=paste0(getwd()))
...
* installing *source* package 'Cairo' ...
...
* DONE (Cairo)

The downloaded source packages are in
'/tmp/RtmpCuwdwM/downloaded_packages'
> library("Cairo", lib=getwd())
> # success!
```

Utiliser des shells conteneurs est amusant, mais ils peuvent devenir un peu embêtant quand vous voulez plus qu'un seul processus interactif. Certaines tâches deviennent plus faciles lorsqu'elles se reposent sur les fondations solides d'un système Guix et de son riche ensemble de services systèmes. La section suivante vous montre comment lancer un système Guix complet à l'intérieur d'un conteneur.

4.2 Conteneurs pour le système Guix

Le système Guix fournit un large éventail de services systèmes interconnectés configurés déclarativement pour former les fondations d'un système GNU fiable et sans état pour n'importe quelle tâche que vous lui donnez. Même lorsque vous utilisez Guix sur une distribution externe, vous pouvez bénéficier de la conception du système Guix en lançant une instance du système dans un conteneur. Avec les mêmes fonctionnalités d'espaces de

noms départagés mentionnés dans la section précédente, l'instance du système Guix qui en résulte est isolée du système hôte et ne partage que les emplacements de fichiers que vous avez explicitement déclarés.

Un conteneur du système Guix est différent du processus shell créé par `guix shell --container` de plusieurs façons importantes. Dans un shell conteneur le processus de conteneurisation est le processus de shell Bash alors qu'une conteneur du système Guix fait tourner le Shepherd en PID 1. Dans un conteneur système tous les services systèmes (voir Section "Services" dans *le manuel de référence de GNU Guix*) sont paramétrés de la même manière que sur un système Guix dans une machine virtuelle ou directement sur le matériel. Cela comprend les démons gérés par le GNU Shepherd (voir Section "Services Shepherd" dans *le manuel de référence de GNU Guix*) ainsi que d'autres types d'extensions du système d'exploitation (voir Section "Composition de services" dans *le manuel de référence de GNU Guix*).

La complexité perçue comme croissante d'un conteneur du système Guix est facilement justifiée lorsque vous devez traiter avec des applications plus complexes qui ont des prérequis plus grands ou plus rigides sur leur contexte d'exécution — des fichiers de configuration, des comptes utilisateurs dédiés, des répertoires pour les le cache ou les fichiers journaux, etc. Sur le système Guix, la demande de ce genre de logiciels est satisfaite en déployant des services systèmes.

4.2.1 Un conteneur de base de données

Un bon exemple pourrait être un serveur de base de données PostgreSQL. La majeure partie de la complexité de configuration d'un tel serveur de base de données est encapsulée dans cette déclaration trompeusement courtes :

```
(service postgresql-service-type
         (postgresql-configuration
          (postgresql postgresql-14)))
```

Une déclaration de système d'exploitation complète utilisable avec un conteneur du système Guix ressemblerait à ceci :

```
(use-modules (gnu))
(use-package-modules databases)
(use-service-modules databases)

(operating-system
  (host-name "container")
  (timezone "Europe/Berlin")
  (file-systems (cons (file-system
                        (device (file-system-label "does-not-matter"))
                        (mount-point "/")
                        (type "ext4"))
                      %base-file-systems))
  (bootloader (bootloader-configuration
               (bootloader grub-bootloader)
               (targets '("/dev/sdX"))))
  (services
   (cons* (service postgresql-service-type
```

```
                    (postgresql-configuration
                     (postgresql postgresql-14)
                     (config-file
                      (postgresql-config-file
                       (log-destination "stderr")
                       (hba-file
                        (plain-file "pg_hba.conf"
                                    "\
local all all trust
host all all 10.0.0.1/32  trust"))
                        (extra-config
                         '(("listen_addresses" "*")
                           ("log_directory"    "/var/log/postgresql")))))))
          (service postgresql-role-service-type
                   (postgresql-role-configuration
                    (roles
                     (list (postgresql-role
                            (name "test")
                            (create-database? #t))))))
          %base-services)))
```

Avec `postgresql-role-service-type` nous définissons un rôle « test » et créons une base de données correspondante, pour que nous puissions le tester immédiatement sans autre paramétrage manuel. Les paramètres de `postgresql-config-file` permettent à un client de se connecter à partir de l'adresse IP 10.0.0.1 sans authentification — une mauvaise idée en production, mais pratique pour cet exemple.

Construisons un script qui exécutera une instance de ce système Guix dans un conteneur. Écrivez la déclaration `operating-system` ci-dessus dans un fichier `os.scm` puis utilisez `guix system container` pour construire le lanceur (voir Section “Invoquer guix system” dans *le manuel de référence de GNU Guix*).

```
$ guix system container os.scm
Les dérivations suivantes seront construites :
  /gnu/store/...-run-container.drv
  ...
construction de /gnu/store/...-run-container.drv...
/gnu/store/...-run-container
```

Maintenant que nous avons le script de lancement nous pouvons le lancer pour démarrer le nouveau système avec un service PostgreSQL. Remarquez qu'à cause de limites non encore résolues, nous devons lancer le lanceur en root, par exemple avec `sudo`.

```
$ sudo /gnu/store/...-run-container
le conteneur système tourne avec le PID 5983
...
```

Mettez le processus en fond avec `Ctrl-z` suivie de `bg`. Prenez note de l'ID du processus dans la sortie ; nous en aurons besoin pour nous connecter au conteneur plus tard. Vous savez quoi ? Essayons de nous attacher au conteneur immédiatement. Nous utiliserons `nsenter`, un outil fournit par le paquet `util-linux` :

```
$ guix shell util-linux
$ sudo nsenter -a -t 5983
root@container /# pgrep -a postgres
49 /gnu/store/...-postgresql-14.4/bin/postgres -D /var/lib/postgresql/data --config-fi
51 postgres: checkpointer
52 postgres: background writer
53 postgres: walwriter
54 postgres: autovacuum launcher
55 postgres: stats collector
56 postgres: logical replication launcher
root@container /# exit
```

Le service PostgreSQL tourne dans le conteneur !

4.2.2 Utilisation du réseau dans le conteneur

Que vaut un système Guix dans lequel tourne un service de bases de données PostgreSQL dans un conteneur si nous ne pouvons lui parler qu'avec des processus originaire de ce conteneur ? Il serait bien mieux de pouvoir parler à la base de données via le réseau.

La manière la plus simple de faire cela est de créer une paire de périphérique Ethernet virtuels (connus sous le nom de `veth`). Nous déplaçons l'un des périphériques (`ceth-test`) dans l'espace de nom `net` du conteneur et laissons l'autre côté (`veth-test`) de la connexion sur le système hôte.

```
pid=5983
ns="guix-test"
host="veth-test"
client="ceth-test"

# Attache le nouvel espace de nom réseau « guix-test » au PID du conteneur.
sudo ip netns attach $ns $pid

# Crée une paire de périphériques
sudo ip link add $host type veth peer name $client

# Déplace le périphérique client dans l'espace de nom réseau du conteneur
sudo ip link set $client netns $ns
```

Puis nous configurons le côté de l'hôte :

```
sudo ip link set $host up
sudo ip addr add 10.0.0.1/24 dev $host
```

. . . puis nous configurons le côté client :

```
sudo ip netns exec $ns  ip link set lo up
sudo ip netns exec $ns  ip link set $client up
sudo ip netns exec $ns  ip addr add 10.0.0.2/24 dev $client
```

Maintenant l'hôte peut atteindre le conteneur à l'adresse IP 10.0.0.2 et le conteneur peut atteindre l'hôte à l'adresse IP 10.0.0.1. C'est tout ce dont nous avons besoin pour communiquer avec le serveur de bases de données dans le conteneur à partir du système hôte, à l'extérieur.

```
$ psql -h 10.0.0.2 -U test
psql (14.4)
Type "help" for help.

test=> CREATE TABLE hello (who TEXT NOT NULL);
CREATE TABLE
test=> INSERT INTO hello (who) VALUES ('world');
INSERT 0 1
test=> SELECT * FROM hello;
  who
-------
 world
(1 row)
```

Maintenant que nous avons fini cette petite démonstration, nettoyons tout ça :

```
sudo kill $pid
sudo ip netns del $ns
sudo ip link del $host
```

5 Machines virtuelles

Guix peut produire des images disque (voir Section “Invoquer guix system” dans *le manuel de référence de GNU Guix*) qui peuvent être utilisées avec des solutions de machines virtuelles telles que virt-manager, GNOME Boxes ou les plus simples QEMU, entre autres.

Ce chapitre vise à fournir des exemples pratiques concernant l'utilisation et la configuration de machines virtuelles sur un système Guix.

5.1 Pont réseau pour QEMU

Par défaut, QEMU utilise un moteur réseau hôte en « mode utilisateur », qui est pratique comme il n'a pas besoin de configuration. Malheureusement, il est aussi très limité. Dans ce mode, la VM (machine virtuelle) invitée peut accéder au réseau de la même manière que l'hôte, mais elle ne peut pas être atteinte depuis l'hôte. De plus, comme le mode de réseau utilisateur de QEMU se base sur ICMP, les outils de réseau basés sur ICMP comme `ping` ne fonctionnent *pas* dans ce mode. Ainsi, il est souvent préférable de configurer un pont réseau, qui permet à l'invité de participer pleinement au réseau. Cela est nécessaire, par exemple, lorsque l'invité est utilisé comme serveur.

5.1.1 Créer une interface réseau pont

Il y a plusieurs manières de créer un pont réseau. La commande suivante indique comment utiliser NetworkManager et son outil en ligne de commande `nmcli`, qui devrait déjà être disponible si votre déclaration de système d'exploitation se base sur l'un des modèles de systèmes de bureau :

```
# nmcli con add type bridge con-name br0 ifname br0
```

Pour que ce pont fasse partie de votre réseau, vous devez associer votre pont réseau à l'interface Ethernet utilisée pour vous connecter au réseau. En supposant que cette interface est nommée ‘`enp2s0`’, la commande suivante peut être utilisée pour cela :

```
# nmcli con add type bridge-slave ifname enp2s0 master br0
```

Important: Seules les interfaces Ethernet peut être ajoutées à un pont. Pour les interfaces sans fil, envisagez l'approche par réseau routé détaillée dans Voir Section 5.2 [Réseau routé pour libvirt], page 59.

By default, the network bridge will allow your guests to obtain their IP address via DHCP, if available on your local network. For simplicity, this is what we will use here. To easily find the guests, they can be configured to advertise their host names via mDNS.

5.1.2 Configuring the QEMU bridge helper script

QEMU comes with a helper program to conveniently make use of a network bridge interface as an unprivileged user voir Section “Network options” dans *QEMU Documentation*. The binary must be made setuid root for proper operation; this can be achieved by adding it to the `privileged-programs` field of your (host) `operating-system` definition, as shown below:

```
(privileged-programs
 (cons (privileged-program
        (program (file-append qemu "/libexec/qemu-bridge-helper"))
```

```
          (setuid? #t))
        %default-privileged-programs))
```

The file `/etc/qemu/bridge.conf` must also be made to allow the bridge interface, as the default is to deny all. Add the following to your list of services to do so:

```
(extra-special-file "/etc/qemu/host.conf" "allow br0\n")
```

5.1.3 Invoking QEMU with the right command line options

When invoking QEMU, the following options should be provided so that the network bridge is used, after having selected a unique MAC address for the guest.

> **Important:** By default, a single MAC address is used for all guests, unless provided. Failing to provide different MAC addresses to each virtual machine making use of the bridge would cause networking issues.

```
$ qemu-system-x86_64 [...] \
    -device virtio-net-pci,netdev=user0,mac=XX:XX:XX:XX:XX:XX \
    -netdev bridge,id=user0,br=br0 \
    [...]
```

To generate MAC addresses that have the QEMU registered prefix, the following snippet can be employed:

```
mac_address="52:54:00:$(dd if=/dev/urandom bs=512 count=1 2>/dev/null \
                           | md5sum \
                           | sed -E 's/^(..)(..)(..).*$/\1:\2:\3/')"
echo $mac_address
```

5.1.4 Networking issues caused by Docker

If you use Docker on your machine, you may experience connectivity issues when attempting to use a network bridge, which are caused by Docker also relying on network bridges and configuring its own routing rules. The solution is add the following `iptables` snippet to your `operating-system` declaration:

```
(service iptables-service-type
             (iptables-configuration
              (ipv4-rules (plain-file "iptables.rules" "\
*filter
:INPUT ACCEPT [0:0]
:FORWARD DROP [0:0]
:OUTPUT ACCEPT [0:0]
-A FORWARD -i br0 -o br0 -j ACCEPT
COMMIT
"))
```

5.2 Réseau routé pour libvirt

If the machine hosting your virtual machines is connected wirelessly to the network, you won't be able to use a true network bridge as explained in the preceding section (voir Section 5.1 [Pont réseau pour QEMU], page 58). In this case, the next best option is to use a *virtual* bridge with static routing and to configure a libvirt-powered virtual machine to use it

(via the `virt-manager` GUI for example). This is similar to the default mode of operation of QEMU/libvirt, except that instead of using NAT (Network Address Translation), it relies on static routes to join the VM (virtual machine) IP address to the LAN (local area network). This provides two-way connectivity to and from the virtual machine, which is needed for exposing services hosted on the virtual machine.

5.2.1 Creating a virtual network bridge

A virtual network bridge consists of a few components/configurations, such as a TUN (network tunnel) interface, DHCP server (dnsmasq) and firewall rules (iptables). The `virsh` command, provided by the `libvirt` package, makes it very easy to create a virtual bridge. You first need to choose a network subnet for your virtual bridge; if your home LAN is in the '`192.168.1.0/24`' network, you could opt to use e.g. '`192.168.2.0/24`'. Define an XML file, e.g. `/tmp/virbr0.xml`, containing the following:

```
<network>
  <name>virbr0</name>
  <bridge name="virbr0" />
  <forward mode="route"/>
  <ip address="192.168.2.0" netmask="255.255.255.0">
    <dhcp>
      <range start="192.168.2.1" end="192.168.2.254"/>
    </dhcp>
  </ip>
</network>
```

Then create and configure the interface using the `virsh` command, as root:

```
virsh net-define /tmp/virbr0.xml
virsh net-autostart virbr0
virsh net-start virbr0
```

The '`virbr0`' interface should now be visible e.g. via the '`ip address`' command. It will be automatically started every time your libvirt virtual machine is started.

5.2.2 Configuring the static routes for your virtual bridge

If you configured your virtual machine to use your newly created '`virbr0`' virtual bridge interface, it should already receive an IP via DHCP such as '`192.168.2.15`' and be reachable from the server hosting it, e.g. via '`ping 192.168.2.15`'. There's one last configuration needed so that the VM can reach the external network: adding static routes to the network's router.

In this example, the LAN network is '`192.168.1.0/24`' and the router configuration web page may be accessible via e.g. the `http://192.168.1.1` page. On a router running the libreCMC (`https://librecmc.org/`) firmware, you would navigate to the Network → Static Routes page (`https://192.168.1.1/cgi-bin/luci/admin/network/routes`), and you would add a new entry to the '`Static IPv4 Routes`' with the following information:

'`Interface`'
: lan

'`Target`'
: 192.168.2.0

‘`IPv4-Netmask`’
: 255.255.255.0

‘`IPv4-Gateway`’
: *server-ip*

‘`Route type`’
: unicast

where *server-ip* is the IP address of the machine hosting the VMs, which should be static.

After saving/applying this new static route, external connectivity should work from within your VM; you can e.g. run ‘`ping gnu.org`’ to verify that it functions correctly.

6 Gestion avancée des paquets

Guix est un gestionnaire de paquets fonctionnel qui propose de nombreuses fonctionnalités en plus de ce que les gestionnaires de paquets traditionnels peuvent faire. Pour quelqu'un qui n'est pas initié, ces fonctionnalités peuvent ne pas paraître utiles au premier coup d'œil. Le but de ce chapitre est de vous montrer certains concepts avancés en gestion de paquets.

voir Section "Gestion des paquets" dans *le manuel de référence de GNU Guix* pour une référence complète.

6.1 Les profils Guix en pratique

Guix fournit une fonctionnalité utile que peut être plutôt étrange pour les débutants et débutantes : les *profils*. C'est une manière de regrouper l'installation de paquets ensemble et chaque utilisateur ou utilisatrice du même système peuvent avoir autant de profils que souhaité.

Que vous programmiez ou non, vous trouverez sans doute plus de flexibilité et de possibilité avec plusieurs profils. Bien qu'ils changent un peu du paradigme des *gestionnaires de paquets traditionnels*, ils sont pratiques à utiliser une fois que vous avec saisi comment les configurer.

> **Remarque:** This section is an opinionated guide on the use of multiple profiles. It predates `guix shell` and its fast profile cache (voir Section "Invoking guix shell" dans *GNU Guix Reference Manual*).
>
> Dans de nombreux cas, vous trouverez qu'utiliser `guix shell` pour configurer l'environnement dont vous avez besoin, quand vous en avez besoin, représente moins de travail que de maintenir un profil dédié. À vous de choisir !

Si vous connaissez '`virtualenv`' de Python, vous pouvez conceptualiser un profil comme une sorte de '`virtualenv`' universel qui peut contenir n'importe quel sorte de logiciel, pas seulement du code Python. En plus, les profils sont auto-suffisants : ils capturent toutes les dépendances à l'exécution qui garantissent que tous les programmes d'un profil fonctionneront toujours à tout instant.

Avoir plusieurs profils présente de nombreux intérêts :

- Une séparation sémantique claire des divers paquets dont vous avez besoin pour différents contextes.
- On peut rendre plusieurs profils disponibles dans l'environnement soit à la connexion, soit dans un shell dédié.
- Les profils peuvent être chargés à la demande. Par exemple, vous pouvez utiliser plusieurs shells, chacun dans un profil différent.
- L'isolation : les programmes d'un profil n'utiliseront pas ceux d'un autre, et vous pouvez même installe plusieurs versions d'un même programme dans deux profils différents, sans conflit.
- Déduplication : les profils partagent les dépendances qui sont exactement les mêmes. Avoir plusieurs profils ne gâche donc pas d'espace.
- Reproductible : lorsque vous utilisez des manifestes déclaratifs, un profil peut être entièrement spécifié par le commit Guix qui a été utilisé pour le créer. Cela signifie

que vous pouvez recréer n'importe où et à n'importe quel moment (`https://guix.gnu.org/blog/2018/multi-dimensional-transactions-and-rollbacks-oh-my/`) exactement le même profil, avec juste l'information du numéro de commit. Voir la section sur les Section 6.1.5 [Profils reproductibles], page 67.

- Des mises à jours et une maintenance plus faciles : avoir plusieurs profils facilite la gestion des listes de paquets à la main.

Concrètement voici des profils courants :

- Les dépendances d'un projet sur lequel vous travaillez.
- Des bibliothèques de votre langage de programmation favori.
- Des programmes spécifiques pour les ordinateurs portables (comme '`powertop`') dont vous n'avez pas besoin sur un ordinateur de bureau.
- TEXlive (il peut être bien pratique si vous avez besoin d'installer un seul paquet pour un document que vous avez reçu par courriel).
- Jeux.

Voyons cela de plus près !

6.1.1 Utilisation de base avec des manifestes

Un profil Guix peut être paramétré par un *manifeste*. Un manifeste est un bout de code Scheme qui spécifie l'ensemble des paquets que vous voulez avoir dans votre profil ; il ressemble à ceci :

```
(specifications->manifest
  '("package-1"
    ;; Version 1.3 de package-2.
    "package-2@1.3"
    ;; La sortie « lib » de package-3.
    "package-3:lib"
    ; ...
    "package-N"))
```

Voir Section "Écrire un manifeste" dans *le manuel de référence de GNU Guix*, pour plus d'informations sur la syntaxe.

On peut créer une spécification de manifeste par profil et les installer de cette manière :

```
GUIX_EXTRA_PROFILES=$HOME/.guix-extra-profiles
mkdir -p "$GUIX_EXTRA_PROFILES"/my-project # s'il n'existe pas encore
guix package --manifest=/path/to/guix-my-project-manifest.scm \
 --profile="$GUIX_EXTRA_PROFILES"/my-project/my-project
```

On spécifie ici une variable arbitraire '`GUIX_EXTRA_PROFILES`' pour pointer vers le répertoire où seront stockés nos profils dans le reste de cet article.

C'est un peu plus propre de placer tous vos profils dans un répertoire unique, où chaque profil a son propre sous-répertoire. De cette manière, chaque sous-répertoire contiendra tous les liens symboliques pour exactement un profil. En plus, il devient facile d'énumérer les profils depuis n'importe quel langage de programmation (p. ex. un script shell) en énumérant simplement les sous-répertoires de '`$GUIX_EXTRA_PROFILES`'.

Remarquez qu'il est aussi possible d'utiliser la sortie de

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

même si vous devrez sans doute enlever `~/.config/guix/current`.

Pour activer tous les profils à la connexion, ajoutez cela à votre `~/.bash_profile` (ou similaire) :

```
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
```

Remarque pour les utilisateurs du système Guix : ce qui précède ressemble à la manière dont votre profil par défaut `~/.guix-profile` est activé dans `/etc/profile`, ce dernier étant chargé par défaut par `~/.bashrc`.

Vous pouvez évidemment choisir de n'en activer qu'une partie :

```
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
```

Lorsqu'un profil est désactivé, il est facile de l'activer pour un shell individuel sans « polluer » le reste de la session :

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

Le secret pour activer un profil est de *sourcer* son fichier '`etc/profile`'. Ce fichier contient du code shell qui exporte les bonnes variables d'environnement nécessaires à activer les logiciels présents dans le profil. Il est créé automatiquement par Guix et doit être sourcé. Il contient les mêmes variables que ce que vous obtiendrez en lançant :

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

Encore une fois, Voir Section "Invoquer guix package" dans *le manuel de référence de GNU Guix* pour les options de la ligne de commande.

Pour mettre à jour un profil, installez de nouveau le manifeste :

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

Pour mettre à jour tous les profils, vous pouvez simplement les énumérer. Par exemple, en supposant que vous spécifications sont dans `~/.guix-manifests/guix-$profile-manifest.scm`, où '`$profile`' est le nom du profil (p. ex « projet1 »), vous pouvez utiliser ce qui suit dans le shell :

```
for profile in "$GUIX_EXTRA_PROFILES"/*; do
```

```
  guix package --profile="$profile" \
   --manifest="$HOME/.guix-manifests/guix-$profile-manifest.scm"
done
```

Chaque profil a ses propres générations :

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

Vous pouvez revenir à n'importe quelle génération d'un profil donné :

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

Enfin, si vous voulez passer à un profil sans hériter l'environnement actuel, vous pouvez l'activer dans un shell vide :

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

6.1.2 Paquets requis

Activer un profil consiste en substance à exporter un ensemble de variables d'environnement. C'est le rôle de '`etc/profile`' dans le profil.

Remarque : seules les variables d'environnement des paquets qui les utilisent seront modifiées.

Par exemple, '`MANPATH`' ne sera pas modifié s'il n'y a pas d'application qui utilise les pages de manuel dans le profil. Donc si vous voulez pouvoir accéder aux pages de manuel facilement une fois le profil chargé, vous avez deux possibilités :

- Exporter la variable manuellement, p. ex

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

- Inclure '`man-db`' dans le manifeste du profil.

Il en va de même pour '`INFOPATH`' (vous pouvez installer '`info-reader`'), '`PKG_CONFIG_PATH`' (installer '`pkg-config`'), etc.

6.1.3 Profil par défaut

Que faire du profil par défaut que Guix garder dans `~/.guix-profile` ?

Vous pouvez lui assigner le rôle que vous souhaitez. Habituellement, vous y installerez un manifeste des paquets que vous voulez pouvoir utiliser dans toutes les situations.

Autrement, vous pouvez en faire un profil sans manifeste pour des paquets sans importance que vous voulez juste garder quelques jours. C'est une manière de pouvoir facilement lancer

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

sans avoir à spécifier un profil.

6.1.4 Les avantages des manifestes

Les manifestes vous permettent de *déclarer* l'ensemble de paquets que vous souhaitez voir dans un profil (voir Section "Écrire un manifeste" dans *le manuel de référence de GNU Guix*). Ils sont pratiques pour garder la liste des paquets et, par exemple, les synchroniser entre plusieurs machines avec un système de gestion de versions.

Les gens se plaignent souvent que les manifestes sont lents à installer quand ils contiennent beaucoup de paquets. C'est particulièrement embêtant quand vous voulez juste mettre à jour un paquet dans un gros manifeste.

C'est une raison de plus d'utiliser plusieurs profils, qui sont bien pratiques pour diviser les manifestes en plusieurs ensembles de paquets de même type. Plusieurs petits profils sont plus flexibles et plus maniables.

Les manifestes ont de nombreux avantages. En particulier, ils facilitent la maintenance :

- Lorsqu'un profil est créé à partir d'un manifeste, le manifeste lui-même est suffisant pour garder la liste des paquets sous le coude et réinstaller le profil plus tard sur un autre système. Pour les profils ad-hoc, il faudrait générer une spécification de manifeste à la main et noter les versions de paquets pour les paquets qui n'utilisent pas la version par défaut.
- `guix package --upgrade` essaye toujours de mettre à jour les paquets qui ont des entrées propagées, même s'il n'y à rien à faire. Les manifestes de Guix résolvent ce problème.
- Lorsque vous mettez partiellement à jour un profil, des conflits peuvent survenir (à cause des dépendances différentes entre les paquets à jour et ceux qui ne le sont pas) et ça peut être embêtant à corriger à la main. Les manifestes suppriment ce problème puisque tous les paquets sont toujours mis à jour en même temps.
- Comme on l'a mentionné plus haut, les manifestes permettent d'avoir des profils reproductibles, alors que les commandes impératives `guix install`, `guix upgrade`, etc, ne le peuvent pas, puisqu'elles produisent un profil différent à chaque fois qu'elles sont lancées, même avec les même paquets. Voir la discussion sur ce problème (`https://issues.guix.gnu.org/issue/33285`).
- Les spécifications de manifestes sont utilisables par les autres commandes '`guix`'. Par exemple, vous pouvez lancer `guix weather -m manifest` pour voir combien de substituts sont disponibles, ce qui peut vous aider à décider si vous voulez faire la mise à jour maintenant ou un peu plus tard. Un autre exemple : vous pouvez lancer `guix package -m manifest.scm` pour créer un lot contenant tous les paquets du manifeste (et leurs références transitives).
- Enfin, les manifestes ont une représentation Scheme, le type d'enregistrement '`<manifest>`'. Vous pouvez les manipuler en Scheme et les passer aux diverses API (`https://fr.wikipedia.org/wiki/Api`) de Guix.

Vous devez bien comprendre que même si vous pouvez utiliser les manifestes pour déclarer des profils, les deux ne sont pas strictement équivalents : les profils pour l'effet de bord « d'épingler » les paquets dans le dépôt, ce qui évite qu'ils ne soient nettoyés (voir Section "Invoquer guix gc" dans *le manuel de référence de GNU Guix*) et s'assure qu'ils seront toujours disponibles à n'importe quel moment dans le futur. La commande `guix shell` protège également les profils récemment utilisés du ramasse-miettes : les profiles qui n'ont pas été utilisé pendant un certain temps peuvent être cependant être nettoyés, avec les paquets auxquels ils se réfèrent.

Pour être sûr à 100 % qu'un profil donné ne sera pas nettoyé, installez le manifeste dans un profil et d'utiliser `GUIX_PROFILE=/le/profil; . "$GUIX_PROFILE"/etc/profile` comme on l'a expliqué plus haut : cela garantie que l'environnement de bidouillage sera toujours disponible.

Avertissement de sécurité : bien que garder d'anciens profils soit pratique, gardez à l'esprit que les anciens paquets n'ont pas forcément reçu les dernières corrections de sécurité.

6.1.5 Profils reproductibles

Pour reproduire un profil bit-à-bit, on a besoin de deux informations :

- un manifeste (voir Section "Écrire un manifeste" dans *le manuel de référence de GNU Guix*),
- une spécification de canal Guix (voir Section "Répliquer Guix" dans *le manuel de référence de GNU Guix*).

En effet, les manifestes seuls ne sont pas forcément suffisants : différentes versions de Guix (ou différents canaux) peuvent produire des sorties différentes avec le même manifeste.

Vous pouvez afficher la spécification de canaux Guix avec '`guix describe --format=channels`' (voir Section "Invoquer guix describe" dans *le manuel de référence de GNU Guix*). Enregistrez-la dans un fichier, par exemple '`channel-specs.scm`'.

Sur un autre ordinateur, vous pouvez utiliser le fichier de spécification de canaux et le manifeste pour reproduire exactement le même profil :

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

mkdir -p "$GUIX_EXTRA"/my-project
guix pull --channels=channel-specs.scm --profile="$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-manifest.scm \
 --profile="$GUIX_EXTRA_PROFILES"/my-project/my-project
```

Vous pouvez supprimer le profil des canaux Guix que vous venez d'installer avec la spécification de canaux, le profil du projet n'en dépend pas.

7 Développement logiciel

Guix est un outil pratique pour les développeurs ; `guix shell`, notamment, fournit un environnement de développement autonome et complet pour votre paquet peu importe le langage dans lequel il est écrit (voir Section “Invoquer guix shell” dans *Manuel de référence GNU Guix*). Pour en bénéficier, vous devez écrire une définition de paquet accessible depuis Guix officiel, un canal, ou bien depuis les sources de votre projet dans le fichier `guix.scm`. Cette dernière option est particulièrement alléchante : la seule chose à faire pour initialiser son environnement de travail est d'exécuter `guix shell` sans argument dans le dépôt du projet.

Les besoins des projets logiciels ne se limitent cependant pas à l'environnement d'exécution. Comment effectuer l'intégration continue du code dans des environnements de compilation Guix ? Comment livrer le code directement aux utilisatrices aventureuses ? Ce chapitre décrit l'ensemble des fichiers qu'une développeuse peut ajouter à son dépôt pour créer un environnement de développement, d'intégration continue et de livraison continue basé sur Guix uniquement[1].

7.1 Guide de démarrage

Comment entreprendre la “Guixification” d'un dépôt ? La première étape, comme vu précédemment, consiste à ajouter un fichier `guix.scm` à la racine du dépôt. Nous prendrons comme exemple Guile () au long de chapitre. Guile est (principalement) écrit en Scheme et en C, possède plusieurs dépendances—Une suite d'outils de compilation C, des bibliothèques C, Autoconf et sa clique, LaTeX, et ainsi de suite... Le fichier `guix.scm` ressemblera à une définition de paquet traditionnelle (voir Section “Définition des paquets” dans *Manuel de référence de GNU Guix*) sans le `define-public` :

```
;; The ‘guix.scm’ file for Guile, for use by ‘guix shell’.

(use-modules (guix)
             (guix build-system gnu)
             ((guix licenses) #:prefix license:)
             (gnu packages autotools)
             (gnu packages base)
             (gnu packages bash)
             (gnu packages bdw-gc)
             (gnu packages compression)
             (gnu packages flex)
             (gnu packages gdb)
             (gnu packages gettext)
             (gnu packages gperf)
             (gnu packages libffi)
             (gnu packages libunistring)
             (gnu packages linux)
```

[1] Ce chapitre est une adaptation de cet article de blog (`https://guix.gnu.org/fr/blog/2023/from-development-environments-to-continuous-integrationthe-ultimate-guide-to-software-development-with-guix/`) (en anglais) publié en juin 2023 sur le site de Guix.

```
              (gnu packages pkg-config)
              (gnu packages readline)
              (gnu packages tex)
              (gnu packages texinfo)
              (gnu packages version-control))

(package
  (name "guile")
  (version "3.0.99-git")                          ;funky version number
  (source #f)                                     ;no source
  (build-system gnu-build-system)
  (native-inputs
   (append (list autoconf
                 automake
                 libtool
                 gnu-gettext
                 flex
                 texinfo
                 texlive-base                 ;for "make pdf"
                 texlive-epsf
                 gperf
                 git
                 gdb
                 strace
                 readline
                 lzip
                 pkg-config)

           ;; When cross-compiling, a native version of Guile itself is
           ;; needed.
           (if (%current-target-system)
               (list this-package)
               '())))
  (inputs
   (list libffi bash-minimal))
  (propagated-inputs
   (list libunistring libgc))

  (native-search-paths
   (list (search-path-specification
          (variable "GUILE_LOAD_PATH")
          (files '("share/guile/site/3.0")))
         (search-path-specification
          (variable "GUILE_LOAD_COMPILED_PATH")
          (files '("lib/guile/3.0/site-ccache")))))
  (synopsis "Scheme implementation intended especially for extensions")
  (description
```

```
   "Guile is the GNU Ubiquitous Intelligent Language for Extensions,
and it's actually a full-blown Scheme implementation!")
  (home-page "https://www.gnu.org/software/guile/")
  (license license:lgpl3+))
```

Même si ça peut paraître fastidieux, une personne souhaitant bidouiller Guile n'aura désormais qu'à exécuter :

```
guix shell
```

pour obtenir un environnement d'exécution avec toutes les dépendances de Guile : celle listées ci-dessus, mais également les *dépendances implicites* : la suite d'outils GCC, GNU Make, sed, grep, etc. Voir Section "Invoquer guix shell" dans *Manuel de référence de GNU Guix* pour plus d'informations sur `guix shell`.

La recommandation du chef: Nous suggérons la création d'environnements de développement comme suit :

```
guix shell --container --link-profile
```

. . . ou plus brièvement :

```
guix shell -CP
```

Nous obtenons un conteneur avec toutes les dépendances qui apparaissent dans `$HOME/.guix-profile` qui permet de profiter des caches comme le `config.cache` (voir Section "Cache Files" dans *Autoconf*) ainsi que des noms de fichier absolus stockés dans les éventuels `Makefile` autogénérés ou autres. L'exécution de l'environnement logiciel dans un conteneur apporte la garantie que seuls les dépendances de Guix et le dossier courant sont accessibles ; aucun élément du système ne peut interférer avec l'environnement de développement ainsi créé.

7.2 Niveau 1 : Compiler avec Guix

Maintenant que nous avons notre définition de paquet (voir Section 7.1 [Guide de démarrage], page 68), pour ne pas en profiter pour compiler Guile avec Guix ? Nous avions omis le champ `source` car `guix shell` ne se préoccupe que des *dépendances* du paquet—afin de mettre en place l'environnement de développement.

Pour compiler le paquet il nous faut remplir le champ `source` avec quelque chose du style :

```
(use-modules (guix)
             (guix git-download)  ;for 'git-predicate'
             ...)

(define vcs-file?
  ;; Return true if the given file is under version control.
  (or (git-predicate (current-source-directory))
      (const #t)))                          ;not in a Git checkout

(package
  (name "guile")
  (version "3.0.99-git")                    ;funky version number
```

```
(source (local-file "." "guile-checkout"
                    #:recursive? #t
                    #:select? vcs-file?))
...)
```

Voici la différence entre la version actuelle et la section précédente :

1. Nous avons ajouté `(guix git-download)` à la liste des modules importés pour la procédure `git-predicate`.
2. Nous avons défini la procédure `vcs-file?` qui nous permet de savoir si un fichier est versionné. Pour faire bonne mesure nous répondons toujours oui lorsque nous ne sommes pas dans un dépôt Git.
3. Nous remplissons la `source` avec un `local-file` (`https://guix.gnu.org/manual/devel/fr/html_node/G_002dExpressions.html#index-local_002dfile`)—une copie récursive des fichiers versionnés (avec le `#:select?`) du dossier courant (`"."`)

À partir de là, notre `guix.scm` prend un second rôle : il nous permet de compiler le logiciel avec Guix. L'intérêt réside en ce que cette compilation est "pure"—nous sommes certains qu'aucun élément de notre copie de travail ou du système n'interfère avec la compilation– et de tester plein de choses. Premièrement, nous pouvons réaliser une compilation native standard :

```
guix build -f guix.scm
```

Mais nous pouvons également compiler pour un autre système (après avoir mis en place voir Section "Réglages du déchargement du démon" dans *Manuel de référence de GNU Guix* ou voir Section "Services de virtualisation" dans *Manuel de référence de GNU Guix*) :

```
guix build -f guix.scm -s aarch64-linux -s riscv64-linux
```

. . . ou en compilation croisée :

```
guix build -f guix.scm --target=x86_64-w64-mingw32
```

Nous pouvons également utiliser les *transformations de paquets* pour tester des variantes (voir Section "Options de transformation de paquets" dans *Manuel de référence de GNU Guix*) :

```
# What if we built with Clang instead of GCC?
guix build -f guix.scm \
  --with-c-toolchain=guile@3.0.99-git=clang-toolchain

# What about that under-tested configure flag?
guix build -f guix.scm \
  --with-configure-flag=guile@3.0.99-git=--disable-networking
```

Pratique !

7.3 Niveau 2 : Le dépôt comme canal

Nous avons maintenant un dépôt Git contenant (entre autres) une définition de paquet (voir Section 7.2 [Compilation avec Guix], page 70). Ne pourrions-nous pas l'utiliser comme *canal* (voir Section "Canaux" dans *Manuel de référence de GNU Guix*) ? Après tout, les canaux sont conçues pour livrer des définitions de paquets aux utilisateurs, et c'est ce que nous faisons avec notre `guix.scm`.

Il s'avère que nous pouvons effectivement faire de notre dépôt un canal, mais à une condition : nous devons créer un dossier à part pour le(s) fichier(s) `.scm` du canal, afin que `guix pull` ne télécharge pas des `.scm` liées au dépôt lors d'une mise à jour — il y en a beaucoup dans Guile ! Voici donc notre disposition initiale, en conservant un lien symbolique à la racine vers `guix.scm` pour la commande `guix shell` :

```
mkdir -p .guix/modules
mv guix.scm .guix/modules/guile-package.scm
ln -s .guix/modules/guile-package.scm guix.scm
```

Pour rendre notre fichier `guix.scm` utilisable dans le canal, nous devons l'adapter en *module de paquets* (voir Section "Modules de paquets" dans *Manuel de référence de GNU Guix*) en utilisant `define-module` à la place de `use-modules` au début. Nous devons également *exporter* une variable de paquet avec `define-public` tout en retournant le paquet à la fin du fichier pour qu'il reste utilisable avec `guix shell` et `guix build -f guix.scm`. Le résultat final ressemble à ça (en omettant les parties identiques) :

```
(define-module (guile-package)
  #:use-module (guix)
  #:use-module (guix git-download)   ;for 'git-predicate'
  ...)

(define vcs-file?
  ;; Return true if the given file is under version control.
  (or (git-predicate (dirname (dirname (current-source-directory))))
      (const #t)))                                  ;not in a Git checkout

(define-public guile
  (package
    (name "guile")
    (version "3.0.99-git")                           ;funky version number
    (source (local-file "../.." "guile-checkout"
                        #:recursive? #t
                        #:select? vcs-file?))
    ...))

;; Return the package object define above at the end of the module.
guile
```

Il nous manque plus que le dernier élément : le fichier `.guix-channel` (`https://guix.gnu.org/manual/devel/fr/html_node/Modules-de-paquets-dans-un-sous_002drepertoire.html`) qui permet à Guix de trouver les modules de paquets de notre dépôt :

```
;; This file lets us present this repo as a Guix channel.

(channel
  (version 0)
  (directory ".guix/modules"))  ;look for package modules under .guix/modules/
```

Pour récapituler, nous avons :

.

```
.guix-channel
guix.scm → .guix/modules/guile-package.scm
.guix
    modules
       guile-package.scm
```

Et voilà, nous avons un canal ! (Nous pourrions faire mieux en ajoutant le support de *l'authentification de canal* (`https://guix.gnu.org/manual/devel/fr/html_node/Specifier-les-autorisations-des-canaux.html`) pour garantir l'authenticité du code. Nous n'entrerons pas dans les détails mais il peut être important de le considérer!) Les utilisateurs peuvent récupérer notre canal avec le fichier `~/.config/guix/channels.scm` (`https://guix.gnu.org/manual/devel/fr/html_node/Specifier-des-canaux-supplementaires.html`) en y ajoutant quelque chose du genre :

```
(append (list (channel
               (name 'guile)
               (url "https://git.savannah.gnu.org/git/guile.git")
               (branch "main")))
        %default-channels)
```

Après un `guix pull`, nous pouvons voir les nouveaux paquets :

```
$ guix describe
Generation 264  May 26 2023 16:00:35    (current)
  guile 36fd2b4
    repository URL: https://git.savannah.gnu.org/git/guile.git
    branch: main
    commit: 36fd2b4920ae926c79b936c29e739e71a6dff2bc
  guix c5bc698
    repository URL: https://git.savannah.gnu.org/git/guix.git
    commit: c5bc698e8922d78ed85989985cc2ceb034de2f23
$ guix package -A ^guile$
guile   3.0.99-git      out,debug       guile-package.scm:51:4
guile   3.0.9           out,debug       gnu/packages/guile.scm:317:2
guile   2.2.7           out,debug       gnu/packages/guile.scm:258:2
guile   2.2.4           out,debug       gnu/packages/guile.scm:304:2
guile   2.0.14          out,debug       gnu/packages/guile.scm:148:2
guile   1.8.8           out             gnu/packages/guile.scm:77:2
$ guix build guile@3.0.99-git
[...]
/gnu/store/axnzbl89yz7ld78bmx72vpqp802dwsar-guile-3.0.99-git-debug
/gnu/store/r34gsij7f0glg2fbakcmmk0zn4v62s5w-guile-3.0.99-git
```

Voici comment, en tant que développeuse, vous pouvez livrer votre logiciel directement aux utilisateurs ! Sans aucun intermédiaire, sans perte de transparence et un suivi de la provenance.

De plus, il devient également trivial de créer des images Docker, des paquets Deb/RPM ou une archive avec `guix pack` (voir Section "Invoquer guix pack" dans *Manuel de référence de GNU Guix*) :

```
# How about a Docker image of our Guile snapshot?
```

```
guix pack -f docker -S /bin=bin guile@3.0.99-git

# And a relocatable RPM?
guix pack -f rpm -R -S /bin=bin guile@3.0.99-git
```

7.4 Bonus: Package Variants

We now have an actual channel, but it contains only one package (voir Section 7.3 [Le dépôt comme canal], page 71). While we're at it, we can define *package variants* (voir Section "Defining Package Variants" dans *GNU Guix Reference Manual*) in our `guile-package.scm` file, variants that we want to be able to test as Guile developers—similar to what we did above with transformation options. We can add them like so:

```
;; This is the '.guix/modules/guile-package.scm' file.

(define-module (guile-package)
  ...)

(define-public guile
  ...)

(define (package-with-configure-flags p flags)
  "Return P with FLAGS as additional 'configure' flags."
  (package/inherit p
    (arguments
     (substitute-keyword-arguments (package-arguments p)
       ((#:configure-flags original-flags #~(list))
        #~(append #$original-flags #$flags))))))

(define-public guile-without-threads
  (package
    (inherit (package-with-configure-flags guile
                                           #~(list "--without-threads")))
    (name "guile-without-threads")))

(define-public guile-without-networking
  (package
    (inherit (package-with-configure-flags guile
                                           #~(list "--disable-networking")))
    (name "guile-without-networking")))


;; Return the package object defined above at the end of the module.
guile
```

We can build these variants as regular packages once we've pulled the channel. Alternatively, from a checkout of Guile, we can run a command like this one from the top level:

```
guix build -L $PWD/.guix/modules guile-without-threads
```

7.5 Level 3: Setting Up Continuous Integration

The channel we defined above (voir Section 7.3 [Le dépôt comme canal], page 71) becomes even more interesting once we set up *continuous integration* (`https://en.wikipedia.org/wiki/Continuous_integration`) (CI). There are several ways to do that.

You can use one of the mainstream continuous integration tools, such as GitLab-CI. To do that, you need to make sure you run jobs in a Docker image or virtual machine that has Guix installed. If we were to do that in the case of Guile, we'd have a job that runs a shell command like this one:

```
guix build -L $PWD/.guix/modules guile@3.0.99-git
```

Doing this works great and has the advantage of being easy to achieve on your favorite CI platform.

That said, you'll really get the most of it by using Cuirass (`https://guix.gnu.org/en/cuirass`), a CI tool designed for and tightly integrated with Guix. Using it is more work than using a hosted CI tool because you first need to set it up, but that setup phase is greatly simplified if you use its Guix System service (voir Section "Continuous Integration" dans *GNU Guix Reference Manual*). Going back to our example, we give Cuirass a spec file that goes like this:

```
;; Cuirass spec file to build all the packages of the ‘guile’ channel.
(list (specification
        (name "guile")
        (build '(channels guile))
        (channels
         (append (list (channel
                        (name 'guile)
                        (url "https://git.savannah.gnu.org/git/guile.git")
                        (branch "main")))
                 %default-channels))))
```

It differs from what you'd do with other CI tools in two important ways:

- Cuirass knows it's tracking *two* channels, `guile` and `guix`. Indeed, our own `guile` package depends on many packages provided by the `guix` channel—GCC, the GNU libc, libffi, and so on. Changes to packages from the `guix` channel can potentially influence our `guile` build and this is something we'd like to see as soon as possible as Guile developers.
- Build results are not thrown away: they can be distributed as *substitutes* so that users of our `guile` channel transparently get pre-built binaries! (voir Section "Substitutes" dans *GNU Guix Reference Manual*, for background info on substitutes.)

From a developer's viewpoint, the end result is this status page (`https://ci.guix.gnu.org/jobset/guile`) listing *evaluations*: each evaluation is a combination of commits of the `guix` and `guile` channels providing a number of *jobs*—one job per package defined in `guile-package.scm` times the number of target architectures.

As for substitutes, they come for free! As an example, since our `guile` jobset is built on ci.guix.gnu.org, which runs `guix publish` (voir Section "Invoking guix publish" dans *GNU Guix Reference Manual*) in addition to Cuirass, one automatically gets substitutes for `guile` builds from ci.guix.gnu.org; no additional work is needed for that.

7.6 Bonus: Build manifest

The Cuirass spec above is convenient: it builds every package in our channel, which includes a few variants (voir Section 7.5 [Mettre en place une intégration continue], page 75). However, this might be insufficiently expressive in some cases: one might want specific cross-compilation jobs, transformations, Docker images, RPM/Deb packages, or even system tests.

To achieve that, you can write a *manifest* (voir Section "Writing Manifests" dans *GNU Guix Reference Manual*). The one we have for Guile has entries for the package variants we defined above, as well as additional variants and cross builds:

```
;; This is ‘.guix/manifest.scm’.

(use-modules (guix)
             (guix profiles)
             (guile-package))   ;import our own package module

(define* (package->manifest-entry* package system
                                   #:key target)
  "Return a manifest entry for PACKAGE on SYSTEM, optionally cross-compiled to
TARGET."
  (manifest-entry
    (inherit (package->manifest-entry package))
    (name (string-append (package-name package) "." system
                         (if target
                             (string-append "." target)
                             "")))
    (item (with-parameters ((%current-system system)
                            (%current-target-system target))
            package))))

(define native-builds
  (manifest
   (append (map (lambda (system)
                  (package->manifest-entry* guile system))

                '("x86_64-linux" "i686-linux"
                  "aarch64-linux" "armhf-linux"
                  "powerpc64le-linux"))
           (map (lambda (guile)
                  (package->manifest-entry* guile "x86_64-linux"))
                (cons (package
                        (inherit (package-with-c-toolchain
                                  guile
                                  `(("clang-toolchain"
                                     ,(specification->package
                                       "clang-toolchain")))))
                        (name "guile-clang"))
```

```
                        (list guile-without-threads
                              guile-without-networking
                              guile-debug
                              guile-strict-typing))))))

(define cross-builds
  (manifest
   (map (lambda (target)
          (package->manifest-entry* guile "x86_64-linux"
                                    #:target target))
        '("i586-pc-gnu"
          "aarch64-linux-gnu"
          "riscv64-linux-gnu"
          "i686-w64-mingw32"
          "x86_64-linux-gnu"))))

(concatenate-manifests (list native-builds cross-builds))
```

We won't go into the details of this manifest; suffice to say that it provides additional flexibility. We now need to tell Cuirass to build this manifest, which is done with a spec slightly different from the previous one:

```
;; Cuirass spec file to build all the packages of the ‘guile’ channel.
(list (specification
        (name "guile")
        (build '(manifest ".guix/manifest.scm"))
        (channels
         (append (list (channel
                        (name 'guile)
                        (url "https://git.savannah.gnu.org/git/guile.git")
                        (branch "main")))
                 %default-channels))))
```

We changed the `(build ...)` part of the spec to `'(manifest ".guix/manifest.scm")` so that it would pick our manifest, and that's it!

7.7 Récapitulatif

We picked Guile as the running example in this chapter and you can see the result here:

- `.guix-channel` (`https://git.savannah.gnu.org/cgit/guile.git/tree/.guix-channel?id=cd57379b3df636198d8cd8e76c1bfbc523762e79`);
- `.guix/modules/guile-package.scm` (`https://git.savannah.gnu.org/cgit/guile.git/tree/.guix/modules/guile-package.scm?id=cd57379b3df636198d8cd8e76c1bfbc523762e79`) with the top-level `guix.scm` symlink;
- `.guix/manifest.scm` (`https://git.savannah.gnu.org/cgit/guile.git/tree/.guix/manifest.scm?id=cd57379b3df636198d8cd8e76c1bfbc523762e79`).

These days, repositories are commonly peppered with dot files for various tools: `.envrc`, `.gitlab-ci.yml`, `.github/workflows`, `Dockerfile`, `.buildpacks`, `Aptfile`,

`requirements.txt`, and whatnot. It may sound like we're proposing a bunch of *additional* files, but in fact those files are expressive enough to *supersede* most or all of those listed above.

With a couple of files, we get support for:

- development environments (`guix shell`);
- pristine test builds, including for package variants and for cross-compilation (`guix build`);
- continuous integration (with Cuirass or with some other tool);
- continuous delivery to users (*via* the channel and with pre-built binaries);
- generation of derivative build artifacts such as Docker images or Deb/RPM packages (`guix pack`).

This a nice (in our view!) unified tool set for reproducible software deployment, and an illustration of how you as a developer can benefit from it!

8 Gestion de l'environnement

Guix fournit plusieurs outils pour gérer l'environnement. Ce chapitre vous montre ces outils.

8.1 Environnement Guix avec direnv

Guix fournit un paquet '`direnv`', qui peut étendre le shell après avoir changé de répertoire de travail. Vous pouvez utiliser cet outil pour préparer un environnement Guix pur.

L'exemple suivant fournit une fonction shell dans `~/.direnvrc`, qui peut être utilisée dans le dépôt Git de Guix dans `~/src/guix/.envrc` pour créer un environnement de construction similaire à celui décrit dans Section "Construire depuis Git" dans *le manuel de référence de GNU Guix*.

Créez un fichier `~/.direnv` avec le code Bash suivant :

```
# Grâce à <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" >> "$target" 2>/dev/null
    # Remarquez qu'on ajoute des variables shells au déclencheur de la fonction.
    echo "$name \$*" >> "$target"
    chmod +x "$target"
  fi
}

use_guix()
{
    # indique un jeton Github.
    export GUIX_GITHUB_TOKEN="xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"

    # Nettoie « GUIX_PACKAGE_PATH ».
    export GUIX_PACKAGE_PATH=""

    # Recrée une racine du ramasse miettes.
    gcroots="$HOME/.config/guix/gcroots"
    mkdir -p "$gcroots"
    gcroot="$gcroots/guix"
    if [ -L "$gcroot" ]
    then
        rm -v "$gcroot"
    fi
```

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

# Paquets de l'environnement.
PACKAGES=(help2man guile-sqlite3 guile-gcrypt)

# Merci <https://lists.gnu.org/archive/html/guix-devel/2016-09/msg00859.html>
eval "$(guix shell --search-paths --root="$gcroot" --pure \
 --development guix ${PACKAGES[@]} ${PACKAGES_MAINTENANCE[@]} "$@")"

# Défini les drapeaux configure à l'avance.
configure()
{
    ./configure
}
export_function configure

# Lance make et construit éventuellement quelque chose.
build()
{
    make -j 2
    if [ $# -gt 0 ]
    then
        ./pre-inst-env guix build "$@"
    fi
}
export_function build

# Défini une commande Git pour pousser.
push()
{
    git push --set-upstream origin
}
export_function push
```

```
    clear                        # Nettoie l'écran.
    git-cal --author='Votre Nom' # Montre le calendrier des contributions.

    # Montre l'aide des commandes.
    echo "
build          construit un paquet ou juste un projet si aucun argument n'est fournit
configure      lance ./configure avec les paramètres pré-définis
push           pousse un dépôt Git
"
}
```

Tous les projets contenant un `.envrc` avec une chaine `use guix` aura des variables d'environnement et des procédures prédéfinies.

Lancez `direnv allow` pour mettre en place l'environnement pour la première fois.

9 Installer Guix sur une grappe de calcul

Guix est intéressant pour les scientifiques et les professionnels du HPC (calcul haute-performance). Il facilite en effet le déploiement de piles logicielles complexes et vous permet de le faire de manière reproductible — vous pouvez redéployer exactement le même logiciel sur des machines différentes et à des moments différents.

Dans ce chapitre nous allons voir comment l'administrateur·ice système d'une grappe peut installer Guix sur le système, de sorte qu'il puisse être utilisé sur tous les nœuds de la grappe, et nous discuterons de différents compromis possibles[1].

> **Remarque:** Nous supposons ici que la grappe utilise une distribution GNU/Linux autre que le Système Guix et que nous allons installer Guix par-dessus.

9.1 Mettre en place un nœud principal

L'approche recommandée est d'installer un *nœud principal* qui exécuterait `guix-daemon` et exporterait `/gnu/store` sur NFS vers les nœuds de calcul.

Rappelez-vous que `guix-daemon` est responsable du démarrage des constructions et des téléchargements pour ses clients (voir Section "Invoquer guix-daemon" dans *le manuel de référence de GNU Guix*), et plus généralement de l'accès à `/gnu/store` qui contient tous les binaires des paquets construits par tous les utilisateur·ices (voir Section "Le dépôt" dans *le manuel de référence de GNU Guix*). Les « clients » signifient toutes les commandes Guix que voient les utilisateurs et utilisatrices, comme `guix install`. Sur une grappe, ces commandes peuvent être lancées sur les nœuds de calcul et l'on souhaiterait qu'elles parlent au `guix-daemon` du nœud principal.

Pour commencer, le nœud principal peut être installé suivant les instructions d'installation binaires habituelles (voir Section "Installation binaire" dans *le manuel de référence de GNU Guix*). Grâce au script d'installation, cela devrait être rapide. Une fois l'installation terminée, nous devons faire quelques ajustements.

Comme nous voulons que `guix-daemon` soit joignable non seulement depuis le nœud principal, mais aussi depuis les nœuds de calcul, nous devons nous arranger pour qu'il se mette en écoute de connexions sur TCP/IP. Pour ce faire, nous allons modifier le fichier de démarrage systemd de `guix-daemon`, `/etc/systemd/system/guix-daemon.service`, et ajouter un argument `--listen` à la ligne `ExecStart` ur qu'elle ressemble à quelque chose comme :

```
ExecStart=/var/guix/profiles/per-user/root/current-guix/bin/guix-daemon \
    --build-users-group=guixbuild \
    --listen=/var/guix/daemon-socket/socket --listen=0.0.0.0
```

Pour que ces changements fassent effet, le service doit être redémarré :

```
systemctl daemon-reload
systemctl restart guix-daemon
```

[1] Ce chapitre est adapté d'un billet de blog publié sur le site web de Guix-HPC en 2017 (`https://hpc.guix.info/blog/2017/11/installing-guix-on-a-cluster/`).

> **Remarque:** `--listen=0.0.0.0` indique que `guix-daemon` traitera *toutes* les connexions TCP entrantes sur le port 44146 (voir Section "Invoquer guix-daemon" dans *le manuel de référence de GNU Guix*). C'est généralement acceptable sur une grappe dont le nœud principal est exclusivement accessible à partir du réseau local de la grappe. Vous ne voulez pas l'exposer sur Internet !

L'étape suivante consiste à définir nos exports NFS dans `/etc/exports` (`https://linux.die.net/man/5/exports`) en ajoutant quelque chose comme :

```
/gnu/store    *(ro)
/var/guix     *(rw, async)
/var/log/guix *(ro)
```

Le répertoire `/gnu/store` peut être exporté en lecture-seule car seul le `guix-daemon` du nœud principal le modifiera jamais. `/var/guix` contient les *profils utilisateurs* gérés par `guix package`. Ainsi, pour permettre à tout le monde d'installer des paquets avec `guix package`, il doit être en lecture-écriture.

Les utilisateurs et utilisatrices peuvent créer autant de profils qu'ils et elles le souhaitent en plus du profil par défaut, `~/.guix-profile`. Par exemple, `guix package -p ~/dev/python-dev -i python` installe Python dans un profil accessible depuis le lien symbolique `~/dev/python-dev`. Pour vous assurer que ce profil est protégé contre le ramasse-miettes, c.-à-d. que Python ne sera pas supprimé de `/gnu/store` alors que le profil existe, *les répertoires personnels devraient également être montés sur le nœud principal* pour que `guix-daemon` connaisse ces profils non standards et évite de supprimer les logiciels auxquels ils se réfèrent.

Vous devriez régulièrement supprimer les paquets inutilisés de `/gnu/store` en exécutant `guix gc` (voir Section "Invoquer guix gc" dans *le manuel de référence de GNU Guix*). Vous pouvez le faire en ajoutant une entrée à la crontab du nœud principal :

```
root@master# crontab -e
```

... avec quelque chose comme cela :

```
# Tous les jours à 5h, lancer le ramasse-miettes pour s'assurer
# d'avoir au moins 10 Gio libres sur /gnu/store.
0 5 * * 1  /usr/local/bin/guix gc -F10G
```

Nous en avons terminé avec le nœud principal ! Voyons maintenant les nœuds de calcul.

9.2 Mettre en place des nœuds de calcul

Tout d'abord, les nœuds de calcul doivent monter les répertoires NFS exportés par le nœud principal. Vous pouvez le faire en ajoutant les lignes suivantes à `/etc/fstab` (`https://linux.die.net/man/5/fstab`) :

```
head-node:/gnu/store    /gnu/store    nfs  defaults,_netdev,vers=3 0 0
head-node:/var/guix     /var/guix     nfs  defaults,_netdev,vers=3 0 0
head-node:/var/log/guix /var/log/guix nfs  defaults,_netdev,vers=3 0 0
```

... où *head-node* est le nom ou l'IP de votre nœud principal. À partir de là, en supposant que les points de montage existent, vous devriez pouvoir monter tous ces répertoires sur chaque nœud de calcul.

Ensuite, nous devons fournir une commande `guix` par défaut que les utilisateur·ices peuvent exécuter lors de leur première connexion à la grappe (ils et elles finiront par invoquer `guix pull`, qui leur fournira leur propre commande `guix`). Comme l'a fait le script d'installation sur le nœud principal, nous le stockons dans `/usr/local/bin` :

```
mkdir -p /usr/local/bin
ln -s /var/guix/profiles/per-user/root/current-guix/bin/guix \
      /usr/local/bin/guix
```

Nous devons ensuite dire à `guix` de parler au démon qui s'exécute sur notre nœud principal, en ajoutant ces lignes à `/etc/profile` :

```
GUIX_DAEMON_SOCKET="guix://head-node"
export GUIX_DAEMON_SOCKET
```

Pour éviter des avertissements et vous assurer que `guix` utilise le bon environnement linguistique, nous devons lui dire d'utiliser les données linguistiques fournies par Guix (voir Section "Réglages applicatifs" dans *le manuel de référence de GNU Guix Reference Manual*) :

```
GUIX_LOCPATH=/var/guix/profiles/per-user/root/guix-profile/lib/locale
export GUIX_LOCPATH

# Nous devons utiliser un nom de paramètre linguistique valide. Essayez « ls $GUIX_LOC
# pour voir les noms utilisables.
LC_ALL=fr_FR.utf8
export LC_ALL
```

For convenience, `guix package` automatically generates `~/.guix-profile/etc/profile`, which defines all the environment variables necessary to use the packages—`PATH`, `C_INCLUDE_PATH`, `PYTHONPATH`, etc. Likewise, `guix pull` does that under `~/.config/guix/current`. Thus it's a good idea to source both from `/etc/profile`:

```
for GUIX_PROFILE in "$HOME/.config/guix/current" "$HOME/.guix-profile"
do
  if [ -f "$GUIX_PROFILE/etc/profile" ]; then
    . "$GUIX_PROFILE/etc/profile"
  fi
done
```

Enfin, Guix propose la complétion des commandes notamment pour Bash et zsh. Dans `/etc/bashrc`, pensez à ajouter cette ligne :

```
. /var/guix/profiles/per-user/root/current-guix/etc/bash_completion.d/guix
```

Et voilà !

Vous pouvez vérifier que tout est en place en vous connectant à un nœud de calcul et en exécutant :

```
guix install hello
```

Le démon sur le nœud principal devrait télécharger les binaires préconstruits pour vous et les déballer dans `/gnu/store` et `guix install` devrait créer un `~/.guix-profile` contenant la commande `~/.guix-profile/bin/hello`.

9.3 Accès réseau

Guix a besoin d'un accès réseau pour télécharger le code source et les binaires préconstruits. La bonne nouvelle, c'est que seul le nœud principal en a besoin car les nœuds de calcul lui délèguent cette tâche.

Les nœuds d'une grappe ont traditionnellement accès au mieux à une liste blanche d'hôtes. Notre nœud principal a besoin au minimum que `ci.guix.gnu.org` soit dans cette liste car c'est là qu'il récupère les binaires préconstruits par défaut, pour tous les paquets dans Guix lui-même.

Au passage, `ci.guix.gnu.org` sert également de *miroir addressé par le contenu* du code source de ces paquets. En conséquence, il suffit d'avoir *uniquement* `ci.guix.gnu.org` dans cette liste blanche.

Les paquets logiciels maintenus dans un dépôt séparé comme ceux des divers canaux HPC (`https://hpc.guix.info/channels`) ne sont bien sur pas disponibles sur `ci.guix.gnu.org`. Pour ces paquets, vous devriez étendre la liste blanche pour que les sources et les binaires préconstruits (en supposant que des serveurs tiers fournissent des binaires pour ces paquets) puissent être téléchargés. En dernier recours, les utilisateur·ices peuvent toujours télécharger les sources sur leur machine de travail et les ajouter au `/gnu/store` de la grappe, de cette manière :

```
GUIX_DAEMON_SOCKET=ssh://compute-node.example.org \
  guix download http://starpu.gforge.inria.fr/files/starpu-1.2.3/starpu-1.2.3.tar.gz
```

La commande ci-dessus télécharge `starpu-1.2.3.tar.gz` *et* l'envoie à l'instance `guix-daemon` du nœud principal par SSH.

Les grappes sans connexion nécessitent plus de travail. Pour le moment, notre suggestion est de télécharger tout le code source nécessaire sur une station de travail qui exécute Guix. Par exemple, avec l'option `--sources` de `guix build` (voir Section "Invoquer guix build" dans *le manuel de référence de GNU Guix*), l'exemple ci-dessous télécharge tout le code source dont dépend le paquet `openmpi` :

```
$ guix build --sources=transitive openmpi

...

/gnu/store/xc17sm60fb8nxadc4qy0c7rqph499z8s-openmpi-1.10.7.tar.bz2
/gnu/store/s67jx92lpipy2nfj5cz818xv430n4b7w-gcc-5.4.0.tar.xz
/gnu/store/npw9qh8a46lrxiwh9xwk0wpi3jlzmjnh-gmp-6.0.0a.tar.xz
/gnu/store/hcz0f4wkdbsvsdky3c0vdvcawhdkyldb-mpfr-3.1.5.tar.xz
/gnu/store/y9akh452n3p4w2v631nj0injx7y0d68x-mpc-1.0.3.tar.gz
/gnu/store/6g5c35q8avfnzs3v14dzl54cmrvddjm2-glibc-2.25.tar.xz
/gnu/store/p9k48dk3dvvk7gads7fk30xc2pxsd66z-hwloc-1.11.8.tar.bz2
/gnu/store/cry9lqidwfrfmgl0x389cs3syr15p13q-gcc-5.4.0.tar.xz
/gnu/store/7ak0v3rzpqm2c5q1mp3v7cj0rxz0qakf-libfabric-1.4.1.tar.bz2
/gnu/store/vh8syjrsilnbfcf582qhmvpg1v3rampf-rdma-core-14.tar.gz
...
```

(Si vous vous posez la question, c'est plus de 320 Mio de code source compressé)

Nous pouvons ensuite créer une grosse archive contenant tout cela (voir Section "Invoquer guix archive" dans *le manuel de référence de GNU Guix*) :

```
$ guix archive --export \
    `guix build --sources=transitive openmpi` \
    > openmpi-source-code.nar
```

. . . et nous pouvons enfin transférer cette archive vers la grappe avec une clé usb et la déballer :

```
$ guix archive --import < openmpi-source-code.nar
```

Ce processus doit être répété chaque fois qu'un nouveau code source doit être apporté à la grappe.

Au moment d'écrire ces lignes, les instituts de recherche qui participent à Guix-HPC n'ont pas de grappe déconnectées. Si vous avez de l'expérience avec ce genre de configuration, nous aimerions entendre vos retours et vos suggestions.

9.4 Utilisation du disque

Une inquiétude courante des administrateur·ices systèmes est de savoir si cela va prendre beaucoup de place. Si cela doit avoir lieu, les jeux de données scientifiques prendront plus probablement toute la place, plutôt que les logiciels compilés. C'est notre expérience après presque dix ans d'utilisation de Guix sur des grappes HPC. Néanmoins, il vaut mieux jeter un œil à la manière dont Guix contribue à l'utilisation du disque.

Tout d'abord, avoir plusieurs versions ou variantes d'un paquet donné dans `/gnu/store` ne coûte pas forcément très cher, car `guix-daemon` implémente la déduplication des fichiers identiques et les variantes de paquets ont tendance à avoir de nombreux fichiers en commun.

Comme nous l'avons mentionné plus haut, nous recommandons d'utiliser une tâche cron pour exécuter `guix gc` régulièrement, pour supprimer les logiciels *inutilisés* de `/gnu/store`. Cependant, il est toujours possible que les utilisateur·ices gardent de nombreux logiciels dans leurs profils, qui sont « vivants » et ne peuvent pas être supprimés du point de vu de `guix gc`.

La solution à ce problème est de demander aux utilisateur·ices de régulièrement supprimer les anciennes générations de leurs profils. Par exemple, la commande suivante supprime les générations de plus de deux mois :

```
guix package --delete-generations=2m
```

De même, il vaut mieux inviter les utilisateur·ices à régulièrement mettre à jour leur profil, ce qui peut réduire le nombre de variantes d'un logiciel donnés stockés dans `/gnu/store` :

```
guix pull
guix upgrade
```

En dernier recours, il est toujours possibles pour les administrateur·ices systèmes de le faire pour leurs utilisateur·ices. Néanmoins, l'une des forces de Guix est la liberté et le contrôle que ses utilisateur·ices ont sur leur environnement logiciel, donc nous vous recommandons fortement de les laisser aux manettes.

9.5 Considérations de sécurité

Sur un grappe PC, guix est généralement utilisé pour gérer des logiciels scientifiques. Les logiciels critiques pour la sécurité comme le noyau du système d'exploitation et les services systèmes comme `sshd` et l'ordonnanceur par lot restent sous votre contrôle.

Le projet Guix a un bon historique de délivrance de mises à jour de sécurité à temps (voir Section “Mises à jour de sécurité” dans *le manuel de référence de GNU Guix*). Pour récupérer les mises à jour de sécurité, les utilisateur·ices doivent exécuter `guix pull && guix upgrade`.

Comme Guix identifie de manière unique les variantes des logiciels, il est facile de voir si un logiciel vulnérable est utilisé. Par exemple, pour vérifier si la variante glibc@2.25 sans correctif d'atténuation de « Stack Clash (`https://www.qualys.com/2017/06/19/stack-clash/stack-clash.txt`) », on peut vérifier si les profils des utilisateur·ices s'y réfèrent ou non :

```
guix gc --referrers /gnu/store/...-glibc-2.25
```

Cela rapportera si des profils qui se réfèrent à cette variante spécifique de glibc existent.

10 Guix System Management

Since Guix does not handle packaging, system configuration and services the way other (more "classical") distributions do, some workflows tend to unfold slightly different as we are used to and need slight adjustment. This chapter intends to help with such manners.

10.1 Upgrade Postgres for Cuirass

With the deprecation of the default value for the postgres package in postgresql-configuration (see b93434e656eba4260df82158a96c295000d3ff44), system upgrades need some manual action before they can take place. Here's a handy guide on how to.

Please note that this is a straight-forward way for smaller datasets. For larger databases `pg_upgrade` (`https://www.postgresql.org/docs/current/pgupgrade.html`) may be the better choice. Handling the service and system upgrade as described in this guide still applies, though.

1. Stop and disable cuirass.

   Prevent the service from starting and failing after a reconfiguration:

   ```
   sudo herd stop cuirass && sudo herd disable cuirass
   ```

2. Dump the database contents.

   ```
   sudo su - postgres -s /bin/sh -c pg_dumpall > /tmp/pg.dump
   ```

3. Add or alter the postgres service.

   Depending on whether your postgres service is defined implicitly (through the dependency from the cuirass service) or its own entry in your operating system's `(services)` property, you need to either add or alter the already existing configuration to reflect your intended version upgrade.

   Be careful not to upgrade directly to postgres-16 – cuirass service for some reason doesn't like that. I had to find and purge the relevant files and then re-initialize after a failed upgrade to postgres 16.

   ```
   (service postgresql-service-type
            (postgresql-configuration
             (postgresql (@ (gnu packages databases) postgresql-15))))
   ```

   Note: If you for some reason didn't read the text here but somewhat blindly followed the examples and *did upgrade to 16*, here's how you reset the state:

   1. Delete the database instance files.

      They default to live under `/var/lib/postgres/data`.

   2. Re-initialize postgres.

      ```
      sudo su - postgres -s /bin/sh -c 'pg_ctl init -D /var/lib/postgres/data'
      ```

4. Reconfigure your system.

   ```
   sudo guix system reconfigure path/to/your/altered/config.scm
   ```

5. Restore database contents.

   ```
   sudo su - postgres -s /bin/sh -c 'psql -d postgres -f /tmp/pg.dump'
   ```

6. Enable and start the service.

   ```
   sudo herd enable cuirass
   sudo herd start cuirass
   ```

11 Remerciements

Guix se base sur le `https://nixos.org/nix/ gestionnaire de paquets Nix` conçu et implémenté par Eelco Dolstra, avec des contributions d'autres personnes (voir le fichier `nix/AUTHORS` dans Guix). Nix a inventé la gestion de paquet fonctionnelle et promu des fonctionnalités sans précédents comme les mises à jour de paquets transactionnelles et les retours en arrière, les profils par utilisateurs et les processus de constructions transparents pour les références. Sans ce travail, Guix n'existerait pas.

Les distributions logicielles basées sur Nix, Nixpkgs et NixOS, ont aussi été une inspiration pour Guix.

GNU Guix lui-même est un travail collectif avec des contributions d'un grand nombre de personnes. Voyez le fichier `AUTHORS` dans Guix pour plus d'information sur ces personnes de qualité. Le fichier `THANKS` liste les personnes qui ont aidé en rapportant des bogues, en prenant soin de l'infrastructure, en fournissant des images et des thèmes, en faisant des suggestions et bien plus. Merci !

Ce document contient des sections adaptées d'articles précédemment publiés sur le blog de Guix sur `https://guix.gnu.org/blog` et le blog de Guix-HPC sur `https://hpc.guix.info/blog`.

Annexe A La licence GNU Free Documentation

Version 1.3, 3 November 2008

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or non-commercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

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

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

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released

under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

   If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

   If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

   If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

   It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

   You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

   A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any,

be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their

titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

   You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

   The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

   In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

   If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

   However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

   Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

   Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `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.

Index des concepts

W

Y