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 language 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 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
```

- 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)
```

- La *quote* (l'apostrophe) désactive l'évaluation d'une expression parenthésée : 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.

- 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`.

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://mitpress.mit.edu/sites/default/files/sicp/index.html`), 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
  ```

  Un ebook non officiel (`https://sarabander.github.io/sicp/`) est aussi disponible.

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 language 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 '`GUIX_PACKAGE_PATH`'

Remarque : à partir de Guix 0.16, les canaux plus flexibles sont préférables et remplacent '`GUIX_PACKAGE_PATH`'. Voir la section suivante.

Il peut être fastidieux de spécifier le fichier depuis la ligne de commande par rapport à un appel à `guix package --install my-hello` comme on le ferait pour les paquets officiels.

Guix permet d'uniformiser le processus en ajoutant autant de « répertoires de déclaration de paquets » que vous le souhaitez.

Créez un répertoire, disons `~/guix-packages` et ajoutez-le à la variable d'environnement '`GUIX_PACKAGE_PATH`' :

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

Pour ajouter plusieurs répertoires, séparez-les avec un deux-points (`:`).

Notre '`my-hello`' précédent doit être légèrement ajusté :

```
(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 '`my-hello`' devrait faire partie de la collection de paquets comme tous les paquets officiels. Vous pouvez le vérifier avec :

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

2.1.2.3 Canaux Guix

Guix 0.16 a introduit la notion de canaux, qui sont similaires à '`GUIX_PACKAGE_PATH`' mais fournit une meilleure intégration et un meilleur suivi de la provenance. Les canaux ne sont pas nécessairement locaux et ils peuvent être maintenus dans un dépôt Git public par exemple. Bien sûr, vous pouvez utiliser plusieurs canaux en même temps.

Voir Section "Canaux" dans *le manuel de référence de GNU Guix* pour des détails sur la mise en place des canaux.

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

(define-public my-libgit2
  (let ((commit "e98d0a37c93574d2c6107bf7f31140b548c6a7bf")
        (revision "1"))
    (package
      (name "my-libgit2")
      (version (git-version "0.26.6" revision commit))
      (source (origin
                (method git-fetch)
                (uri (git-reference
                      (url "https://github.com/libgit2/libgit2/")
                      (commit commit)))
                (file-name (git-file-name name version))
                (sha256
                 (base32
                  "17pjvprmdrx4h6bb1hhc98w9qi6ki7yl57f090n9kbhswxqfs7s3"))
                (patches (search-patches "libgit2-mtime-0.patch"))
                (modules '((guix build utils)))
                ;; 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 _ (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
            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 first %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? #true))
      (begin
        (if (string-suffix? ".zip" source)
```

```
            (invoke "unzip" source)
            (invoke "tar" "xvf" source))
          (chdir (first-subdirectory "."))))
    #true)
```

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")))
    ;; ...
    #true))
```

La procédure doit renvoyer `#true` si elle réussit. S'appuyer sur la valeur de retour de la dernière expression n'est pas très solide parce qu'il n'y a pas de garantie qu'elle sera `#true`. Donc le `#true` à la fin permet de s'assurer que la bonne valeur est renvoyée si la phase réussit.

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))
  ...)
```

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

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

```
                "\n"))
```

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)))

(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. `%default-extra-linux-options` sont l'ensemble de ces options et elles devaient être ajoutées puisqu'elles ont été remplacées 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.

En fonction de la machine cible, qui peut aller d'une machine `x86_64` standard à un petit ordinateur ARM sur carte unique comme le Pine64, les contraintes sur l'image varient beaucoup. Les fabricants imposent différents formats d'image avec plusieurs tailles de partitions et de positions.

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 :

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

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 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.

3.4.1 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 [...] ».

3.4.2 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/`).

3.5 Personnaliser un gestionnaire de fenêtres

3.5.1 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.s
(xft:cache-fonts)
(set-font (make-instance 'xft:font :family "DejaVu Sans Mono" :subfamily "Book" :size
```

3.5.2 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.

3.5.2.1 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` :

```
(screen-locker-service 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.

3.6 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
                     certs
                     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* nss-certs            ; pour l'accès HTTPS
                 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.

3.7 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.

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

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

```
(file-systems (cons*

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

               source-drive ;; Doit correspondre au nom que vous avez donné au disque

               (file-system
                (device (%source-directory)) ;; Assurez-vous que « source-directory »
                (mount-point "/tmp")
                (type "none") ;; On monte un dossier, pas une partition, donc le mont
                (flags '(bind-mount))
                (dependencies (list source-drive)) ;; Assurez-vous que « source-drive
                )

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

               ))
```

3.8 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://4zwzi66wwdaalbhgnix55ea3ab4pvvw66ll2ow53kjub6se4q2bclc
                      (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
```

3.9 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")))))
```

4 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.

4.1 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.

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 4.1.5 [Profils reproductibles], page 42.
- 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 !

4.1.1 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_P
```

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-pro
```

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-ma
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
```

4.1.2 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.

4.1.3 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.

4.1.4 Les avantages des manifestes

Les manifestes 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.

Voyons un exemple :

1. Vous avez un environnement pour bidouiller un projet pour lequel il n'y a pas encore de paquet Guix. Vous construisez l'environnement avec un manifeste puis lancez `guix environment -m manifest.scm`. Jusqu'ici tout va bien.
2. Plusieurs semaines plus tard vous avez lancé quelques `guix pull` entre temps. Plusieurs dépendances du manifeste ont été mises à jour ; ou bien vous avez lancé `guix gc` et certains paquets requis par le manifeste ont été nettoyés.
3. Finalement, vous vous remettez au travail sur ce projet, donc vous lancez `guix shell -m manifest.scm`. Mais maintenant vous devez attendre que Guix construise et installe des paquets !

Idéalement, vous voudriez éviter de perdre du temps à reconstruire. C'est en fait possible, tout ce dont on a besoin, c'est d'installer 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.

4.1.5 Profils reproductibles

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

- un manifeste,
- et une spécification de canaux 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`'. 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 "$GUIX_EXTRA"/my-project
guix pull --channels=channel-specs.scm --profile "$GUIX_EXTRA/my-project/guix"

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

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.

5 Gestion de l'environnement

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

5.1 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)

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

# Défini les drapeaux configure à l'avance.
configure()
{
    ./configure --localstatedir=/var --prefix=
}
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.

6 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`.

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