Livro de Receitas do GNU Guix

Tutoriais e exemplos para usar o Gerenciador de Pacotes Funcional do GNU Guix

Desenvolvedores do GNU Guix

Sumário

1 Tutoriais sobre Scheme

GNU Guix foi escrito na linguagem de programação de uso geral Scheme, e muitos de seus recursos podem ser acessados e manipulados programaticamente. Você pode usar Scheme para gerar definições de pacotes, modificá-los, construí-los, implantar sistemas operacionais inteiros, etc.

Conhecer o básico de como programar com Scheme irá desbloquear muitos dos recursos avançados que o Guix oferece — e você nem precisa ser um programador experiente para usá-los!

Vamos começar!

1.1 Um curso intensivo de Scheme

Guix usa a implementação Guile do Scheme. Para começar a brincar com a linguagem, instale-a com `guix install guile` e inicie um *REPL*—abreviação de *read-eval-print loop* (`https://en.wikipedia.org/wiki/Read%E2%80%93eval%E2%80%93print_loop`)—executando `guile` na linha de comando.

Alternativamente, você também pode executar `guix shell guile -- guile` se preferir não ter o Guile instalado em seu perfil de usuário.

Nos exemplos a seguir, as linhas mostram o que você digitaria no REPL; linhas que começam com "⇒" mostram resultados de avaliação, enquanto linhas que começam com "⊣" mostram coisas que são impressas. Veja Seção "Using Guile Interactively" em *GNU Guile Reference Manual*, para mais detalhes sobre o REPL.

- A sintaxe do esquema se resume a uma árvore de expressões (ou *s-expression* no jargão Lisp). Uma expressão pode ser um literal, como números e strings, ou um composto que é uma lista entre parênteses de compostos e literais. `#true` e `#false` (abreviados `#t` e `#f`) representam os booleanos "true" e "false", respectivamente.

  Exemplos de expressões válidas:

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

  17
  ⇒ 17

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

- Este último exemplo é uma chamada de função aninhada em outra chamada de função. Quando uma expressão entre parênteses é avaliada, o primeiro termo é a função e o restante são os argumentos passados para a função. Cada função retorna a última expressão avaliada como valor de retorno.
- Funções anônimas —*procedures* na linguagem do Scheme — são declaradas com o termo `lambda`:

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

O procedimento acima retorna o quadrado do seu argumento. Como tudo é uma expressão, a expressão `lambda` retorna um procedimento anônimo, que por sua vez pode ser aplicado a um argumento:

```
((lambda (x) (* x x)) 3)
⇒ 9
```

Os procedimentos são valores regulares, assim como números, strings, booleanos e assim por diante.

- Qualquer coisa pode receber um nome global com `define`:

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

- Os procedimentos podem ser definidos de forma mais concisa com a seguinte sintaxe:

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

- Uma estrutura de lista pode ser criada com o procedimento `list`:

```
(list 2 a 5 7)
⇒ (2 3 5 7)
```

- Os procedimentos padrão são fornecidos pelo módulo `(srfi srfi-1)` para criar e processar listas (veja Seção "SRFI-1" em *GNU Guile Reference Manual*). Aqui estão alguns dos mais úteis em ação:

```
(use-modules (srfi srfi-1))  ;import list processing procedures

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

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

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

Observe como o primeiro argumento para `map`, `filter`, `remove` e `find` é um procedimento!

- O *quote* desativa a avaliação de uma expressão entre parênteses, também chamada de expressão S ou "s-exp": o primeiro termo não é chamado sobre os outros termos (veja Seção "Sintaxe da expressão" em *Manual de referência do GNU Guile*). Assim, ele efetivamente retorna uma lista de termos.

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

'(2 a 5 7)
⇒ (2 a 5 7)
```

- O `quasiquote` (`` ` ``, uma crase) desativa a avaliação de uma expressão entre parênteses até que `unquote` (`,`, uma vírgula) a reative. Assim, nos fornece um controle refinado sobre o que é avaliado e o que não é.

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

  Observe que o resultado acima é uma lista de elementos mistos: números, símbolos (aqui `a`) e o último elemento é uma lista em si.

- Guix define uma variante de expressões simbólicas (S-expressions) com esteróides chamada *G-expressions* ou "gexps", que vem com uma variante de `quasiquote` e `unquote`: `#~` (ou `gexp`) e `#$` (ou `ungexp`). Eles permitem *preparar código para execução posterior*.

  Por exemplo, você encontrará gexps em algumas definições de pacotes onde eles fornecem código a ser executado durante o processo de construção do pacote. Eles se parecem com isto:

```
(use-modules (guix gexp)           ;so we can write gexps
             (gnu packages base))  ;for 'coreutils'

;; Below is a G-expression representing staged code.
#~(begin
    ;; Invoke 'ls' from the package defined by the 'coreutils'
    ;; variable.
    (system* #$(file-append coreutils "/bin/ls") "-l")

    ;; Create this package's output directory.
    (mkdir #$output))
```

  Veja Seção "Expressões-G" em *GNU Guix Reference Manual*, para saber mais sobre gexps.

- Múltiplas variáveis podem ser nomeadas localmente com `let` (veja Seção "Local Bindings" em *GNU Guile Reference Manual*):

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

x
⇒ 10

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

  Use `let*` para permitir que declarações de variáveis posteriores se refiram a definições anteriores.

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

- *Keywords* normalmente são usados para identificar os parâmetros nomeados de um procedimento. Eles são prefixados por `#:` (hash, dois pontos) seguido por caracteres alfanuméricos: `#:like-this`. Veja Seção "Palavras-chave" em *Manual de referência do GNU Guile*.
- A porcentagem `%` normalmente é usada para variáveis globais somente-leitura no estágio de construção. Observe que é apenas uma convenção, como `_` em C. Scheme trata `%` exatamente da mesma forma que qualquer outra letra.
- Os módulos são criados com `define-module` (veja Seção "Criando Módulos Guile" em *GNU Guile Reference Manual*). Por exemplo

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

  define o módulo `guix build-system ruby` que deve estar localizado em `guix/build-system/ruby.scm` em algum lugar no caminho de carregamento do Guile. Depende do módulo `(guix store)` e exporta duas variáveis, `ruby-build` e `ruby-build-system`.

  Veja Seção "Módulos de pacote" em *GNU Guix Reference Manual*, para informações sobre módulos que definem pacotes.

  Indo além: Scheme é uma linguagem que tem sido amplamente utilizada para ensinar programação e você encontrará muito material usando-a como veículo. Aqui está uma seleção de documentos para saber mais sobre o Scheme:

  - *A Scheme Primer* (`https://spritely.institute/static/papers/scheme-primer.html`), by Christine Lemmer-Webber and the Spritely Institute.
  - *Scheme at a Glance* (`http://www.troubleshooters.com/codecorn/scheme_guile/hello.htm`), por Steve Litt.
  - *Structure and Interpretation of Computer Programs* (`https://sarabander.github.io/sicp/`), por Harold Abelson e Gerald Jay Sussman, com Julie Sussman. Coloquialmente conhecido como "SICP", este livro é uma referência.

    Você também pode instalá-lo e lê-lo em seu computador:

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

  Você encontrará mais livros, tutoriais e outros recursos em `https://schemers.org/`.

2 Empacotamento

Este capítulo é dedicado a ensinar como adicionar pacotes à coleção de pacotes que vem com o GNU Guix. Isso envolve escrever definições de pacotes no Guile Scheme, organizá-las em módulos de pacotes e construí-las.

2.1 Tutorial sobre empacotamento

GNU Guix se destaca como o gerenciador de pacotes *hackeável*, principalmente porque usa GNU Guile (`https://www.gnu.org/software/guile/`), uma poderosa linguagem de programação de alto nível, um dos dialetos Scheme (`https://en.wikipedia.org/wiki/Scheme_%28programming_language%29`) da família Lisp (`https://en.wikipedia.org/wiki/Lisp_%28programming_language%29`).

As definições de pacotes também são escritas em Scheme, o que capacita o Guix de maneiras muito exclusivas, ao contrário da maioria dos outros gerenciadores de pacotes que usam shell scripts ou linguagens simples.

- Utilize funções, estruturas, macros e toda a expressividade do Scheme para definições de seus pacotes.
- A herança facilita a personalização de um pacote, herdando-o e modificando apenas o que é necessário.
- Processamento em lote: toda a coleção de pacotes pode ser analisada, filtrada e processada. Construindo um servidor headless com todas as interfaces gráficas removidas? É possível. Quer reconstruir tudo, desde o código-fonte usando sinalizadores específicos de otimização do compilador? Passe o argumento `#:make-flags "..."` para a lista de pacotes. Não seria exagero pensar em Gentoo USE flags (`https://wiki.gentoo.org/wiki/USE_flag`) aqui, mas isso vai ainda mais longe: as mudanças não precisam ser pensadas de antemão pelo empacotador, elas podem ser *programadas* pelo usuário!

O tutorial a seguir cobre todos os fundamentos da criação de pacotes com Guix. Não pressupõe muito conhecimento do sistema Guix nem da linguagem Lisp. Espera-se apenas que o leitor esteja familiarizado com a linha de comando e tenha alguns conhecimentos básicos de programação.

2.1.1 Um pacote "Hello World"

A seção "Definindo Pacotes" do manual apresenta os fundamentos do empacotamento Guix (veja Seção "Definindo pacotes" em *GNU Guix Reference Manual*). Na seção seguinte, revisaremos parcialmente esses princípios básicos novamente.

GNU Hello é um projeto fictício que serve como exemplo idiomático para empacotamento. Ele usa o sistema de compilação GNU (`./configure && make && make install`). Guix já fornece uma definição de pacote que é um exemplo perfeito para começar. Você pode consultar sua declaração com `guix edit hello` na linha de comando. Vamos ver como fica:

```
(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+)))
```

Como você pode ver, a maior parte é bastante simples. Mas vamos revisar os campos juntos:

'`name`' O nome do projeto. Usando as convenções do Scheme, preferimos mantê-lo em minúsculas, sem sublinhado e usando palavras separadas por traços.

'`source`' Este campo contém uma descrição da origem do código-fonte. O registro `origin` contém estes campos:

1. O método, aqui `url-fetch` para download via HTTP/FTP, mas outros métodos existem, como `git-fetch` para repositórios Git.
2. O URI, que normalmente é alguma `https://` localização para `url-fetch`. Aqui o especial 'mirror://gnu' refere-se a um conjunto de locais bem conhecidos, todos os quais podem ser usados pelo Guix para buscar a fonte, caso alguns deles falhem.
3. A soma de verificação `sha256` do arquivo solicitado. Isto é essencial para garantir que a fonte não está corrompida. Observe que o Guix funciona com strings base32, daí a chamada para a função `base32`.

'`build-system`'
É aqui que o poder de abstração fornecido pela linguagem Scheme realmente brilha: neste caso, o `gnu-build-system` abstrai as famosas invocações de shell `./configure && make && make install`. Outros sistemas de compilação incluem o `trivial-build-system` que não faz nada e exige que o empacotador programe todas as etapas de compilação, o `python-build-system`, o `emacs-build- system` e muito mais (veja Seção "Sistemas de compilação" em *GNU Guix Reference Manual*).

'`synopsis`'
Deve ser um resumo conciso do que o pacote faz. Para muitos pacotes, uma etiqueta da página inicial do projeto pode ser usado como sinopse.

'`description`'
Assim como na sinopse, não há problema em reutilizar a descrição do projeto na página inicial. Observe que o Guix usa a sintaxe de Texinfo.

‘página inicial’
: Use HTTPS, se disponível.

‘license’
: Consulte `guix/licenses.scm` na fonte do projeto para obter uma lista completa de licenças disponíveis.

É hora de construir nosso primeiro pacote! Nada sofisticado aqui por enquanto: vamos nos ater a um `my-hello` fictício, uma cópia da declaração acima.

Tal como acontece com o ritualístico “Hello World” ensinado com a maioria das linguagens de programação, esta será possivelmente a abordagem mais “manual”. Trabalharemos em uma configuração ideal mais tarde; por enquanto seguiremos o caminho mais simples.

Salve o seguinte em um arquivo `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+))
```

Explicaremos o código extra em um momento.

Sinta-se à vontade para brincar com os diferentes valores dos vários campos. Se você alterar a fonte, precisará atualizar a soma de verificação. Na verdade, Guix se recusa a construir qualquer coisa se a soma de verificação fornecida não corresponder à soma de verificação calculada do código-fonte. Para obter a soma de verificação correta da declaração do pacote, precisamos baixar o código-fonte, calcular a soma de verificação sha256 e convertê-la para base32.

Felizmente, o Guix pode automatizar essa tarefa para nós; tudo o que precisamos é fornecer o 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
```

Neste caso específico a saída nos informa qual espelho foi escolhido. Se o resultado do comando acima não for o mesmo do trecho acima, atualize sua declaração `my-hello` de acordo.

Observe que os tarballs do pacote GNU vêm com uma assinatura OpenPGP, então você definitivamente deve verificar a assinatura deste tarball com 'gpg' para autenticá-lo antes de prosseguir:

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

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

Você pode então correr alegremente

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

Agora você deve ter `my-hello` em seu perfil!

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

Fomos o mais longe que pudemos sem qualquer conhecimento do Scheme. Antes de passar para pacotes mais complexos, agora é o momento certo para aprimorar seus conhecimentos sobre o Scheme. veja Seção 1.1 [Um curso intensivo de Scheme], Página 1, para se atualizar.

2.1.2 Configuração

No restante deste capítulo contaremos com alguns conhecimentos básicos de programação de esquemas. Agora vamos detalhar as diferentes configurações possíveis para trabalhar em pacotes Guix.

Existem várias maneiras de configurar um ambiente de empacotamento Guix.

Recomendamos que você trabalhe diretamente no checkout do código-fonte do Guix, pois facilita a contribuição de todos para o projeto.

Mas primeiro, vamos examinar outras possibilidades.

2.1.2.1 Arquivo local

Isto é o que fizemos anteriormente com '`my-hello`'. Com os princípios básicos do esquema que cobrimos, agora podemos explicar os principais pedaços. Conforme declarado em `guix package --help`:

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

Assim, a última expressão *deve* retornar um pacote, que é o caso do nosso exemplo anterior.

A expressão `use-modules` informa quais módulos precisamos no arquivo. Módulos são uma coleção de valores e procedimentos. Eles são comumente chamados de "bibliotecas" ou "pacotes" em outras linguagens de programação.

2.1.2.2 Canais

Guix e sua coleção de pacotes podem ser estendidos através de *canais*. Um canal é um repositório Git, público ou não, contendo arquivos `.scm` que fornecem pacotes (veja Seção "Definindo pacotes" em *GNU Guix Reference Manual*) ou serviços (veja Seção "Definindo serviços" em *Manual de Referência GNU Guix*).

Como você criaria um canal? Primeiro, crie um diretório que conterá seus arquivos `.scm`, digamos `~/my-channel`:

```
mkdir ~/my-channel
```

Suponha que você queira adicionar o pacote '`my-hello`' que vimos anteriormente; primeiro precisa de alguns ajustes:

```
(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+)))
```

Observe que atribuímos o valor do pacote a um nome de variável exportado com `define-public`. Isso é efetivamente atribuir o pacote à variável `my-hello` para que ele possa ser referenciado, entre outras coisas, como dependência de outros pacotes.

Se você usar `guix package --install-from-file=my-hello.scm` no arquivo acima, ele falhará porque a última expressão, `define-public`, não retorna um pacote. Mesmo assim, se você quiser usar `define-public` neste caso de uso, certifique-se de que o arquivo termine com uma avaliação de `my-hello`:

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

my-hello
```

Este último exemplo não é muito típico.

Agora, como você torna esse pacote visível para os comandos `guix` para poder testar seus pacotes? Você precisa adicionar o diretório ao caminho de pesquisa usando a opção de linha de comando `-L`, como nestes exemplos:

```
guix show -L ~/my-channel my-hello
guix build -L ~/my-channel my-hello
```

A etapa final é transformar `~/my-channel` em um canal real, disponibilizando sua coleção de pacotes perfeitamente *via* qualquer comando `guix`. Para fazer isso, primeiro você precisa torná-lo um repositório Git:

```
cd ~/my-channel
git init
git add my-hello.scm
git commit -m "First commit of my channel."
```

E pronto, você tem um canal! A partir daí, você pode adicionar este canal à configuração do seu canal em `~/.config/guix/channels.scm` (veja Seção “Especificando canais adicionais” em *GNU Guix Reference Manual*); supondo que você mantenha seu canal local por enquanto, o `channels.scm` ficaria mais ou menos assim:

```
(append (list (channel
                (name 'my-channel)
                (url (string-append "file://" (getenv "HOME")
                                    "/my-channel"))))
        %default-channels)
```

Da próxima vez que você executar `guix pull`, seu canal será selecionado e os pacotes que ele definir estarão prontamente disponíveis para todos os comandos `guix`, mesmo se você não passar `-L`. O comando `guix description` mostrará que o Guix está, de fato, usando os canais `my-channel` e `guix`.

Veja Seção “Criando um canal” em *GNU Guix Reference Manual*, para detalhes.

2.1.2.3 Direct checkout hacking

É recomendado trabalhar diretamente no projeto Guix: isso reduz o atrito quando chega a hora de enviar suas alterações ao upstream para permitir que a comunidade se beneficie de seu trabalho árduo!

Ao contrário da maioria das distribuições de software, o repositório Guix mantém em um só lugar as ferramentas (incluindo o gerenciador de pacotes) e as definições dos pacotes. Essa escolha foi feita para dar aos desenvolvedores a flexibilidade de modificar a API sem quebras, atualizando todos os pacotes ao mesmo tempo. Isto reduz a inércia do desenvolvimento.

Confira o repositório oficial Git (`https://git-scm.com/`):

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

No restante deste artigo, usamos '`$GUIX_CHECKOUT`' para nos referir ao local do checkout.

Siga as instruções no manual (veja Seção "Contribuindo" em *GNU Guix Reference Manual*) para configurar o ambiente do repositório.

Quando estiver pronto, você poderá usar as definições de pacote do ambiente do repositório.

Sinta-se à vontade para editar as definições de pacotes encontradas em '`$GUIX_CHECKOUT/gnu/packages`'.

O script '`$GUIX_CHECKOUT/pre-inst-env`' permite usar '`guix`' sobre a coleção de pacotes do repositório (veja Seção "Executando guix antes dele ser instalado" em *GNU Guix Reference Manual*) .

- Pesquisa de pacotes, como 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
```

- Construa um pacote, como Ruby versão 2.1:

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

- Instale-o em seu perfil de usuário:

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

- Verifique se há erros comuns:

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

Guix se esforça para manter um alto padrão de empacotamento; ao contribuir para o projeto Guix, lembre-se de

- seguir o estilo de codificação (veja Seção "Estilo de código" em *GNU Guix Reference Manual*),
- e revise a lista de verificação do manual (veja Seção "Enviando patches" em *GNU Guix Reference Manual*).

Quando estiver satisfeito com o resultado, você pode enviar sua contribuição para torná-lo parte do Guix. Este processo também é detalhado no manual. (veja Seção "Contribuindo" em *Manual de Referência GNU Guix*)

É um esforço da comunidade, então quanto mais participar, melhor o Guix se torna!

2.1.3 Exemplo estendido

O exemplo "Hello World" acima é tão simples quanto parece. Os pacotes podem ser mais complexos do que isso e o Guix pode lidar com cenários mais avançados. Vejamos outro pacote mais sofisticado (ligeiramente modificado em relação à fonte):

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

(define-public my-libgit2
  (let ((commit "e98d0a37c93574d2c6107bf7f31140b548c6a7bf")
        (revision "1"))
    (package
      (name "my-libgit2")
      (version (git-version "0.26.6" revision commit))
      (source (origin
                (method git-fetch)
                (uri (git-reference
                      (url "https://github.com/libgit2/libgit2/")
                      (commit commit)))
                (file-name (git-file-name name version))
                (sha256
                 (base32
                  "17pjvprmdrx4h6bb1hhc98w9qi6ki7yl57f090n9kbhswxqfs7s3"))
                (patches (search-patches "libgit2-mtime-0.patch"))
                (modules '((guix build utils)))
                ;; Remove bundled software.
                (snippet '(delete-file-recursively "deps"))))
      (build-system cmake-build-system)
      (outputs '("out" "debug"))
      (arguments
       `(#:tests? #true                         ; Run the test suite (this is the defa
         #:configure-flags '("-DUSE_SHA1DC=ON") ; SHA-1 collision detection
         #:phases
         (modify-phases %standard-phases
           (add-after 'unpack 'fix-hardcoded-paths
             (lambda _
               (substitute* "tests/repo/init.c"
```

```
                  (("#!/bin/sh") (string-append "#!" (which "sh"))))
                (substitute* "tests/clar/fs.h"
                  (("/bin/cp") (which "cp"))
                  (("/bin/rm") (which "rm")))))
            ;; Run checks more verbosely.
            (replace 'check
              (lambda* (#:key tests? #:allow-other-keys)
                (when tests?
                  (invoke "./libgit2_clar" "-v" "-Q"))))
            (add-after 'unpack 'make-files-writable-for-tests
              (lambda _ (for-each make-file-writable (find-files "."))))))
      (inputs
       (list libssh2 http-parser python-wrapper))
      (native-inputs
       (list pkg-config))
      (propagated-inputs
       ;; These two libraries are in 'Requires.private' in 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))))
```

(Nos casos em que você deseja ajustar apenas alguns campos de uma definição de pacote, você deve confiar na herança em vez de copiar e colar tudo. Veja abaixo.)

Vamos discutir esses campos em profundidade.

2.1.3.1 Método `git-fetch`

Ao contrário do método `url-fetch`, `git-fetch` espera um `git-reference` que usa um repositório Git e um commit. O commit pode ser qualquer referência do Git, como tags, portanto, se `version` estiver marcado, ele poderá ser usado diretamente. Às vezes, a tag é prefixada com `v`; nesse caso, você usaria `(commit (string-append "v" version))`.

Para garantir que o código-fonte do repositório Git seja armazenado em um diretório com um nome descritivo, usamos `(nomedo- arquivo (nome-do-arquivo-git versão))`.

O procedimento `git-version` pode ser usado para derivar a versão ao empacotar programas para um commit específico, seguindo as diretrizes do contribuidor Guix (veja Seção "Números de versão" em *GNU Guix Reference Manual*).

Como obter o hash `sha256` que está aí, você pergunta? Invocando `guix hash` em um checkout do commit desejado, da seguinte forma:

```
git clone https://github.com/libgit2/libgit2/
cd libgit2
git checkout v0.26.6
```

```
guix hash -rx .
```

`guix hash -rx` calcula um hash SHA256 em todo o diretório, excluindo o subdiretório `.git` (veja Seção "Invocando guix hash" em *GNU Guix Reference Manual*).

No futuro, `guix download` será capaz de executar essas etapas para você, assim como faz para downloads regulares.

2.1.3.2 Snippets

Os trechos (`"Snippets"`) são códigos de esquema citados (ou seja, não avaliados) que são um meio de corrigir a fonte. Eles são uma alternativa Guix-y aos arquivos `.patch` tradicionais. Por causa da citação, o código só é avaliado quando passado para o daemon Guix para construção. Pode haver quantos trechos forem necessários.

Os snippets podem precisar de módulos Guile adicionais que podem ser importados do campo `modules`.

2.1.3.3 Entradas

Existem 3 tipos de entradas diferentes. Resumidamente:

native-inputs
: Necessário para construção, mas não para tempo de execução - instalar um pacote por meio de um substituto não instalará essas entradas.

entradas
: Instalado na loja mas não no perfil, além de estar presente na hora da construção.

propagated-inputs
: Instalado na loja e no perfil, além de estar presente na hora da construção.

Veja Seção "referência de pacote" em *Manual de referência GNU Guix* para mais detalhes.

A distinção entre as diversas entradas é importante: se uma dependência puder ser tratada como uma *input* em vez de uma *propagated input*, isso deverá ser feito, caso contrário ela "poluirá" o perfil do usuário sem nenhuma boa razão.

Por exemplo, um usuário que instala um programa gráfico que depende de uma ferramenta de linha de comando pode estar interessado apenas na parte gráfica, portanto não há necessidade de forçar a ferramenta de linha de comando no perfil do usuário. A dependência é uma preocupação do pacote, não do usuário. *Inputs* tornam possível lidar com dependências sem incomodar o usuário, adicionando arquivos executáveis (ou bibliotecas) indesejados ao seu perfil.

O mesmo vale para *native-inputs*: depois que o programa é instalado, as dependências em tempo de construção podem ser coletadas como lixo com segurança. Também importa quando um substituto está disponível, caso em que apenas *inputs* e *propagated inputs* serão obtidos: os *native inputs* não são necessários para instalar um pacote de um substituto.

> **Nota:** Você pode ver aqui e ali trechos onde as entradas do pacote são escritas de maneira bem diferente, assim:
>
> ```
> ;; The "old style" for inputs.
> (inputs
>  `(("libssh2" ,libssh2)
> ```

```
("http-parser" ,http-parser)
("python" ,python-wrapper)))
```

Este é o "estilo antigo", onde cada entrada na lista recebe explicitamente um rótulo (uma string). Ainda é compatível, mas recomendamos usar o estilo acima. Veja Seção "referência de pacote" em *GNU Guix Reference Manual*, para mais informações.

2.1.3.4 Saídas

Assim como um pacote pode ter múltiplas entradas, ele também pode produzir múltiplas saídas.

Cada saída corresponde a um diretório separado na loja.

O usuário pode escolher qual saída instalar; isso é útil para economizar espaço ou evitar poluir o perfil do usuário com executáveis ou bibliotecas indesejadas.

A separação de saída é opcional. Quando o campo `outputs` é omitido, a saída padrão e única (o pacote completo) é referida como `"out"`.

Nomes de saída separados típicos incluem `debug` e `doc`.

É aconselhável separar as saídas apenas quando você mostrar que vale a pena: se o tamanho da saída for significativo (compare com `guix size`) ou caso o pacote seja modular.

2.1.3.5 Argumentos do sistema de compilação

O `arguments` é uma lista de valores de palavras-chave usadas para configurar o processo de construção.

O argumento mais simples `#:tests?` pode ser usado para desabilitar o conjunto de testes ao construir o pacote. Isso é útil principalmente quando o pacote não apresenta nenhum conjunto de testes. É altamente recomendável manter o conjunto de testes ativado, se houver.

Outro argumento comum é `:make-flags`, que especifica uma lista de sinalizadores a serem acrescentados ao executar o make, como faria na linha de comando. Por exemplo, os seguintes sinalizadores

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

traduzir para

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

Isso define o compilador C para `gcc` e a variável `prefix` (o diretório de instalação no jargão Make) para `(assoc-ref %outputs "out")`, que é um estágio de construção global variável apontando para o diretório de destino na loja (algo como `/gnu/store/...-my-libgit2-20180408`).

Da mesma forma, é possível definir os sinalizadores de configuração:

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

A variável `%build-inputs` também é gerada no escopo. Ela é uma tabela de associações que mapeia os nomes de entrada para seus diretórios de armazenamento.

A palavra-chave `phases` lista as etapas sequenciais do sistema de compilação. Normalmente as fases incluem `unpack`, `configure`, `build`, `install` e `check`. Para saber mais

sobre essas fases, você precisa definir a definição apropriada do sistema de compilação em '`$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 do 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
```

Se quiser saber mais sobre o que acontece nessas fases, consulte os procedimentos associados.

Por exemplo, no momento em que este livro foi escrito, a definição de `unpack` para o sistema de compilação GNU era:

```
(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)
```

Observe a chamada `chdir`: ela altera o diretório de trabalho para onde a fonte foi descompactada. Assim, cada fase após o `unpack` usará a fonte como diretório de trabalho, e é por isso que podemos trabalhar diretamente nos arquivos de origem. Isto é, a menos que uma fase posterior mude o diretório de trabalho para outro.

Modificamos a lista de `%standard-phases` do sistema de construção com a macro `modify-phases` conforme a lista de modificações especificadas, que pode ter os seguintes formatos:

- `(add-before` *`phase new-phase procedure`*`)`: Run *procedure* named *new-phase* before *phase*.
- `(add-after` *`phase new-phase procedure`*`)`: O mesmo, mas depois.
- `(replace` *`phase procedure`*`)`.
- `(delete` *`phase`*`)`.

O *procedure* suporta os argumentos de palavra-chave `inputs` e `outputs`. Cada entrada (seja *native*, *propagated* ou não) e diretório de saída são referenciados por seus nomes nessas variáveis. Assim `(assoc-ref outputs "out")` é o diretório de armazenamento da saída principal do pacote. Um procedimento de fase pode ser assim:

```
(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))
```

O procedimento deve retornar `#true` em caso de sucesso. É frágil confiar no valor de retorno da última expressão usada para ajustar a fase porque não há garantia de que seria um `#true`. Daí o `#true` final para garantir que o valor correto seja retornado em caso de sucesso.

2.1.3.6 Preparação de código

O leitor astuto deve ter notado a sintaxe de quase aspas e vírgula no campo do argumento. Na verdade, o código de construção na declaração do pacote não deve ser avaliado no lado do cliente, mas apenas quando passado para o daemon Guix. Esse mecanismo de passagem de código entre dois processos em execução é chamado preparação de código (`https://arxiv.org/abs/1709.00833`).

2.1.3.7 Funções utilitárias

Ao personalizar `phases`, muitas vezes precisamos escrever código que imite as invocações equivalentes do sistema (`make`, `mkdir`, `cp`, etc.) comumente usado durante “ Instalações regulares no estilo Unix”.

Alguns como `chmod` são nativos do Guile. Veja *manual de referência do Guile* para obter uma lista completa.

Guix fornece funções auxiliares adicionais que são especialmente úteis no contexto de gerenciamento de pacotes.

Algumas dessas funções podem ser encontradas em '`$GUIX_CHECKOUT/guix/guix/build/utils.scm`'. A maioria delas reflete o comportamento dos comandos tradicionais do sistema Unix:

`which` Como o comando do sistema '`which`'.

`find-files`
Semelhante ao comando do sistema '`find`'.

`mkdir-p` Como '`mkdir -p`', que cria todos os diretórios conforme necessário.

`install-file`
Semelhante a '`install`' ao instalar um arquivo em um diretório (possivelmente inexistente). Guile tem `copy-file` que funciona como '`cp`'.

`copy-recursively`
Como '`cp -r`'.

`delete-file-recursively`
Como '`rm -rf`'.

`invoke` Executar um executável. Deve ser usado em vez de `system*`.

`with-directory-excursion`
Execute o corpo em um diretório de trabalho diferente e restaure o diretório de trabalho anterior.

`substitute*`
Uma função do tipo "`sed`".

Veja Seção "Build Utilities" em *GNU Guix Reference Manual*, para obter mais informações sobre esses utilitários.

2.1.3.8 Prefixo do módulo

A licença em nosso último exemplo precisa de um prefixo: isso se deve à forma como o módulo `license` foi importado no pacote, como `#:use-module ((guix Licenses) #:prefix License:)`. O mecanismo de importação do módulo Guile (veja Seção "Using Guile Modules" em *manual de referência do Guile*) dá ao usuário controle total sobre o namespace: isso é necessário para evitar conflitos entre, digamos, a variável '`zlib`' de '`licenses.scm`' (um valor *license*) e a variável '`zlib`' de '`compression.scm`' (um valor *package*).

2.1.4 Outros sistemas de construção

O que vimos até agora cobre a maioria dos pacotes que usam um sistema de compilação diferente do `trivial-build-system`. Este último não automatiza nada e deixa você construir tudo manualmente. Isso pode ser mais exigente e não abordaremos isso aqui por enquanto, mas felizmente raramente é necessário recorrer a este sistema.

Para outros sistemas de construção, como ASDF, Emacs, Perl, Ruby e muitos outros, o processo é muito semelhante ao sistema de construção GNU, exceto por alguns argumentos especializados.

Veja Seção "Sistemas de compilação" em *GNU Guix Reference Manual*, para obter mais informações sobre sistemas de construção, ou verifique o código-fonte em '`$GUIX_CHECKOUT/guix/build`' e '`$GUIX_CHECKOUT/guix/build -sistema`' diretórios.

2.1.5 Definição de pacote programável e automatizada

Não podemos repetir o suficiente: ter uma linguagem de programação completa em mãos nos capacita de maneiras que vão muito além do gerenciamento tradicional de pacotes.

Vamos ilustrar isso com alguns recursos incríveis do Guix!

2.1.5.1 Importadores recursivos

Você pode achar alguns sistemas de compilação bons o suficiente para que haja pouco a fazer para escrever um pacote, a ponto de se tornar repetitivo e tedioso depois de um tempo. Uma *razão de ser* dos computadores é substituir os seres humanos nessas tarefas chatas. Então, vamos dizer ao Guix para fazer isso para nós e criar a definição de pacote de um pacote R do CRAN (a saída é cortada para ser concisa):

```
$ 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)))
```

O importador recursivo não importará pacotes para os quais o Guix já possui definições de pacote, exceto o primeiro.

Nem todos os aplicativos podem ser empacotados dessa forma, apenas aqueles que dependem de um número selecionado de sistemas suportados. Leia sobre a lista completa de importadores na seção de importação de guix do manual (veja Seção "Invoking guix import" em *GNU Guix Reference Manual*).

2.1.5.2 Atualização automática

O Guix pode ser inteligente o suficiente para verificar atualizações nos sistemas que conhece. Ele pode relatar definições de pacotes desatualizadas com

```
$ guix refresh hello
```

Na maioria dos casos, atualizar um pacote para uma versão mais recente requer pouco mais do que alterar o número da versão e a soma de verificação. Guix também pode fazer isso automaticamente:

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

2.1.5.3 Herança

Se você começou a navegar pelas definições de pacotes existentes, deve ter notado que um número significativo deles possui um campo `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")))))
```

Todos os campos não especificados são herdados do pacote pai. Isto é muito conveniente para criar pacotes alternativos, por exemplo, com diferentes fontes, versões ou opções de compilação.

2.1.6 Obtendo ajuda

Infelizmente, alguns aplicativos podem ser difíceis de empacotar. Às vezes, eles precisam de um patch para funcionar com a hierarquia do sistema de arquivos não padrão imposta pelo armazenamento. Às vezes, os testes não funcionam corretamente. (Eles podem ser ignorados, mas isso não é recomendado.) Outras vezes, o pacote resultante não será reproduzível.

Se você estiver emperrado, incapaz de descobrir como resolver qualquer tipo de problema de empacotamento, não hesite em pedir ajuda à comunidade.

Consulte página inicial do Guix (`https://www.gnu.org/software/guix/contact/`) para obter informações sobre listas de discussão, IRC, etc.

2.1.7 Conclusão

Este tutorial foi uma amostra do sofisticado gerenciamento de pacotes que o Guix possui. Neste ponto, restringimos principalmente esta introdução ao `gnu-build-system`, que é uma camada de abstração central na qual se baseiam abstrações mais avançadas.

Para onde vamos daqui? Em seguida, devemos dissecar as entranhas do sistema de construção, removendo todas as abstrações, usando o `trivial-build-system`: isso deve nos dar uma compreensão completa do processo antes de investigar algumas técnicas de empacotamento mais avançadas e casos extremos.

Outros recursos que valem a pena explorar são os recursos interativos de edição e depuração do Guix fornecidos pelo Guile REPL.

Esses recursos sofisticados são totalmente opcionais e podem esperar; agora é um bom momento para fazer uma pausa bem merecida. Com o que apresentamos aqui você deve estar bem preparado para empacotar muitos programas. Você pode começar imediatamente e esperamos ver suas contribuições em breve!

2.1.8 Referências

- O referência do pacote no manual (`https://www.gnu.org/software/guix/manual/en/html_node/Defining-Packages.html`)
- guia de hacking de Pjotr para GNU Guix (`https://gitlab.com/pjotrp/guix-notes/blob/master/HACKING.org`)
- “GNU Guix: Package without a scheme!” (`https://www.gnu.org/software/guix/guix-ghm-andreas-20130823.pdf`), by Andreas Enge

3 Configuração do sistema

Guix oferece uma linguagem flexível para configurar declarativamente seu sistema Guix. Essa flexibilidade às vezes pode ser esmagadora. O objetivo deste capítulo é demonstrar alguns conceitos avançados de configuração.

veja Seção “Configuração do sistema” em *Manual de referência do GNU Guix* para uma referência completa.

3.1 Login automático em um TTY específico

Embora o manual do Guix explique o login automático de um usuário para *todas* TTYs (veja Seção “auto-login to TTY” em *GNU Guix Reference Manual*), alguns podem preferir uma situação em que um usuário está logado em um TTY com os outros TTYs configurados para fazer login com usuários diferentes ou com nenhum. Observe que é possível fazer login automático de um usuário em qualquer TTY, mas geralmente é aconselhável evitar `tty1`, que, por padrão, é usado para registrar avisos e erros.

Aqui está como se pode configurar o login automático para um usuário em um 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))
```

Pode-se também `compose` (veja Seção “Higher-Order Functions” em *guile*) `auto-login-to-tty` para fazer login de vários usuários em vários ttys.

Finalmente, aqui está uma nota de cautela. Configurar o login automático em um TTY significa que qualquer pessoa pode ligar seu computador e executar comandos como seu usuário normal. No entanto, se você tiver uma partição raiz criptografada e, portanto, já precisar inserir uma senha quando o sistema inicializar, o login automático pode ser uma opção conveniente.

3.2 Customizando o Kernel

Guix é, em sua essência, uma distribuição baseada em código-fonte com substitutos (veja Seção “Substitutos” em *GNU Guix Reference Manual*) e, como tal, construir pacotes a

partir de seu código-fonte é uma parte esperada das instalações e atualizações regulares de pacotes. Dado este ponto de partida, faz sentido que sejam feitos esforços para reduzir a quantidade de tempo gasto na compilação de pacotes, e as recentes mudanças e atualizações na construção e distribuição de substitutos continuam a ser um tópico de discussão dentro do Guix.

O kernel, embora não exija uma superabundância de RAM para ser construído, leva muito tempo em uma máquina média. A configuração oficial do kernel, como é o caso de muitas distribuições GNU/Linux, erra pelo lado da inclusão, e é isso que realmente faz com que a construção demore tanto tempo quando o kernel é compilado a partir do código-fonte.

O kernel do Linux, entretanto, também pode ser descrito apenas como um pacote antigo normal e, como tal, pode ser personalizado como qualquer outro pacote. O procedimento é um pouco diferente, embora isso se deva principalmente à natureza de como a definição do pacote é escrita.

A definição do pacote do kernel `linux-libre` é na verdade um procedimento que cria um pacote.

```
(define* (make-linux-libre* version gnu-revision source supported-systems
                            #:key
                            (extra-version #f)
                            ;; A function that takes an arch and a variant.
                            ;; See kernel-config for an example.
                            (configuration-file #f)
                            (defconfig "defconfig")
                            (extra-options (default-extra-linux-options version)))
  ...)
```

O pacote `linux-libre` atual é para a série 5.15.x e é declarado assim:

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

Quaisquer chaves às quais não sejam atribuídos valores herdam seu valor padrão da definição `make-linux-libre`. Ao comparar os dois trechos acima, observe o comentário do código que se refere a `#:configuration-file`. Por causa disso, não é realmente fácil incluir uma configuração de kernel personalizada na definição, mas não se preocupe, existem outras maneiras de trabalhar com o que temos.

Existem duas maneiras de criar um kernel com uma configuração de kernel personalizada. A primeira é fornecer um arquivo `.config` padrão durante o processo de construção, incluindo um arquivo `.config` real como uma entrada nativa para nosso kernel personalizado. A seguir está um trecho da fase `'configure` personalizada da definição do pacote `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)))
```

Abaixo está um exemplo de pacote de kernel. O pacote `linux-libre` não é nada especial e pode ser herdado e ter seus campos substituídos como qualquer outro pacote:

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

No mesmo diretório do arquivo que define `linux-libre-E2140` está um arquivo chamado `E2140.config`, que é um arquivo de configuração real do kernel. A palavra-chave `defconfig` de `make-linux-libre` é deixada em branco aqui, então a única configuração do kernel no pacote é aquela que foi incluída no campo `native-inputs`.

A segunda maneira de criar um kernel customizado é passar um novo valor para a palavra-chave `extra-options` do procedimento `make-linux-libre`. A palavra-chave `extra-options` funciona com outra função definida logo abaixo dela:

```
(define (default-extra-linux-options version)
  `(;; 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"))
```

E no script de configuração personalizado do pacote ‘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")
```

Portanto, ao não fornecer um arquivo de configuração, o `.config` começa em branco e então escrevemos nele a coleção de flags que desejamos. Aqui está outro kernel personalizado:

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

(define-public linux-libre-macbook41
  ;; XXX: Access the internal 'make-linux-libre*' procedure, which is
  ;; private and unexported, and is liable to change in the future.
  ((@@ (gnu packages linux) make-linux-libre*)
   (@@ (gnu packages linux) linux-libre-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))
```

In the above example `%file-systems` is a collection of flags enabling different file system support, `%efi-support` enables EFI support and `%emulation` enables a x86_64-linux machine to act in 32-bit mode also. The `default-extra-linux-options` procedure is the one defined above, which had to be used to avoid loosing the default configuration options of the `extra-options` keyword.

Tudo isso parece viável, mas como saber quais módulos são necessários para um sistema específico? Dois lugares que podem ser úteis para tentar responder a esta pergunta são o Gentoo Handbook (`https://wiki.gentoo.org/wiki/Handbook:AMD64/Installation/Kernel`) e o documentação do próprio kernel (`https://www .kernel.org/doc/html/latest/admin-guide/README.html?highlight=localmodconfig`). Pela documentação do kernel, parece que `make localmodconfig` é o comando que queremos.

Para realmente executar `make localmodconfig` primeiro precisamos obter e descompactar o código-fonte do kernel:

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

Uma vez dentro do diretório que contém o código-fonte, execute `touch .config` para criar um `.config` inicial e vazio para começar. `make localmodconfig` funciona vendo o que você já tem em `.config` e informando o que está faltando. Se o arquivo estiver em branco, você está perdendo tudo. O próximo passo é executar:

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

e observe a saída. Observe que o arquivo `.config` ainda está vazio. A saída geralmente contém dois tipos de avisos. O primeiro começa com "WARNING" e pode ser ignorado no nosso caso. A segunda leitura:

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

Para cada uma dessas linhas, copie a parte `CONFIG_XXXX_XXXX` para `.config` no diretório e anexe `=m`, para que no final fique assim:

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

Após copiar todas as opções de configuração, execute `make localmodconfig` novamente para ter certeza de que você não tem nenhuma saída começando com "module". Depois de todos esses módulos específicos da máquina, restam mais alguns que também são necessários. `CONFIG_MODULES` é necessário para que você possa construir e carregar módulos separadamente e não ter tudo embutido no kernel. `CONFIG_BLK_DEV_SD` é necessário para leitura de discos rígidos. É possível que existam outros módulos dos quais você precisará.

Este post não pretende ser um guia para configurar seu próprio kernel, portanto, se você decidir construir um kernel personalizado, você terá que procurar outros guias para criar um kernel adequado às suas necessidades.

A segunda maneira de definir a configuração do kernel faz mais uso dos recursos do Guix e permite compartilhar segmentos de configuração entre diferentes kernels. Por exemplo, todas as máquinas que usam EFI para inicializar possuem vários sinalizadores de configuração EFI necessários. É provável que todos os kernels compartilhem uma lista de sistemas de arquivos para suporte. Ao usar variáveis, é mais fácil ver rapidamente quais recursos estão habilitados e garantir que você não tenha recursos em um kernel, mas ausentes em outro.

No entanto, não foi discutido o initrd do Guix e sua personalização. É provável que você precise modificar o initrd em uma máquina usando um kernel customizado, já que certos módulos que devem ser compilados podem não estar disponíveis para inclusão no initrd.

3.3 API de imagem do sistema Guix

Historicamente, o Sistema Guix é centrado em uma estrutura `operating-system`. Esta estrutura contém vários campos que vão desde o gerenciador de boot e a declaração do kernel até os serviços a serem instalados.

Dependendo da máquina de destino, que pode ir de uma máquina `x86_64` padrão a um pequeno computador de placa única ARM, como o Pine64, as restrições de imagem podem variar muito. Os fabricantes de hardware imporão diferentes formatos de imagem com vários tamanhos de partição e deslocamentos.

Para criar imagens adequadas para todas essas máquinas, é necessária uma nova abstração: esse é o objetivo do registro `image`. Este registro contém todas as informações

necessárias para ser transformada em uma imagem autônoma, que pode ser inicializada diretamente em qualquer máquina de destino.

```
(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)))
```

Este registro contém o sistema operacional a ser instanciado. O campo `format` define o tipo de imagem e pode ser `efi-raw`, `qcow2` ou `iso9660` por exemplo. No futuro, poderá ser estendido para `docker` ou outros tipos de imagem.

Um novo diretório nas fontes do Guix é dedicado à definição de imagens. Por enquanto existem quatro arquivos:

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

Vamos dar uma olhada em `pine64.scm`. Ele contém a variável `pine64-barebones-os` que é uma definição mínima de um sistema operacional dedicado à placa **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))))
```

Os campos `kernel` e `bootloader` apontam para pacotes dedicados a esta placa.

Logo abaixo, a variável `pine64-image-type` também está definida.

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

Ele está usando um registro do qual ainda não falamos, o registro `image-type`, definido desta forma:

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

O objetivo principal deste registro é associar um nome a um procedimento que transforma um isso é necessário, vamos dar uma olhada no comando que produz uma imagem de um arquivo de configuração `operating-system`:

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

Este comando espera uma configuração `operating-system` mas como devemos indicar que queremos uma imagem direcionada a uma placa Pine64? Precisamos fornecer uma informação extra, o `image-type`, passando o sinalizador `--image-type` ou `-t`, desta forma:

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

Este parâmetro `image-type` aponta para o `pine64-image-type` definido acima. Portanto, ao `operating-system` declarado em `my-os.scm` será aplicado o procedimento `(cut image-with-os arm64-disk-image <>)` para transformá-lo em um imagem.

A imagem resultante se parece com:

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

que é a agregação do `operating-system` definido em `my-os.scm` ao registro `arm64-disk-image`.

Mas chega de loucura do esquema. O que essa API de imagem traz para o usuário do Guix?

Pode-se executar:

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

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

e escrevendo um arquivo `operating-system` baseado em `pine64-barebones-os`, você pode personalizar sua imagem de acordo com suas preferências em um arquivo (`my-pine-os.scm`) como este :

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

execute:

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

ou,

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

para obter uma imagem que pode ser gravada diretamente em um disco rígido e inicializada.

Sem alterar nada em `my-hurd-os.scm`, chamando:

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

em vez disso, produzirá uma imagem Hurd QEMU.

3.4 Usando chaves de segurança

O uso de chaves de segurança pode melhorar sua segurança, fornecendo uma segunda fonte de autenticação que não pode ser facilmente roubada ou copiada, pelo menos para um adversário remoto (algo que você possui), para o segredo principal (uma senha - algo que você conhece). , reduzindo o risco de falsificação de identidade.

O exemplo de configuração detalhado abaixo mostra qual configuração mínima precisa ser feita em seu sistema Guix para permitir o uso de uma chave de segurança Yubico. Espera-se que a configuração também possa ser útil para outras chaves de segurança, com pequenos ajustes.

3.4.1 Configuração para uso como autenticador de dois fatores (2FA)

Para serem utilizáveis, as regras do udev do sistema devem ser estendidas com regras específicas de chave. O seguinte mostra como estender suas regras do udev com o arquivo de regras do udev `lib/udev/rules.d/70-u2f.rules` fornecido pelo pacote `libfido2` do `(gnu packages security- token)` e adicione seu usuário ao grupo ‘`"plugdev"`’ que ele usa:

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

Depois de reconfigurar seu sistema e fazer login novamente em sua sessão gráfica para que o novo grupo esteja em vigor para seu usuário, você pode verificar se sua chave pode ser usada iniciando:

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

e validar que a chave de segurança pode ser redefinida através do menu "Redefinir sua chave de segurança". Se funcionar, parabéns, sua chave de segurança está pronta para ser usada com aplicativos que suportam autenticação de dois fatores (2FA).

3.4.2 Desativando a geração de código OTP para um Yubikey

Se você usa uma chave de segurança Yubikey e fica irritado com os códigos OTP falsos que ela gera ao tocar inadvertidamente na chave (por exemplo, fazendo com que você se torne um spammer no canal '`#guix`' ao discutir sobre seu cliente de IRC favorito!), você pode desativá-lo através do seguinte comando `ykman`:

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

Alternativamente, você pode usar o comando `ykman-gui` fornecido pelo pacote `yubikey-manager-qt` e desativar totalmente o aplicativo '`OTP`' para a interface USB ou, a partir do pacote '`Applications -> Visualização OTP`', exclua a configuração do slot 1, que vem pré-configurada com o aplicativo Yubico OTP.

3.4.3 Exigindo que um Yubikey abra um banco de dados KeePassXC

O aplicativo gerenciador de senhas KeePassXC tem suporte para Yubikeys, mas requer a instalação de regras udev para seu sistema Guix e algumas configurações do aplicativo Yubico OTP na chave.

O arquivo de regras do udev necessário vem do pacote `yubikey-personalization` e pode ser instalado como:

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

Depois de reconfigurar seu sistema (e reconectar seu Yubikey), você desejará configurar o aplicativo de desafio/resposta OTP de seu Yubikey em seu slot 2, que é o que o KeePassXC usa. É fácil fazer isso por meio da ferramenta de configuração gráfica Yubikey Manager, que pode ser invocada com:

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

Primeiro, certifique-se de que '`OTP`' esteja habilitado na aba '`Interfaces`', depois navegue até '`Applications -> OTP`' e clique no botão '`Configure`' abaixo de '`Long Touch (Slot 2)`' seção. Selecione '`Challenge-response`', insira ou gere uma chave secreta e clique no botão '`Finish`'. Se você tiver um segundo Yubikey que gostaria de usar como backup, você deve configurá-lo da mesma forma, usando a chave secreta *same*.

Seu Yubikey agora deve ser detectado pelo KeePassXC. Ele pode ser adicionado a um banco de dados navegando até o menu '`Database -> Database Security...`' do KeePassXC e clicando no botão '`Adicionar proteção adicional...`' e em '`Add Challenge-Response`', selecionando a chave de segurança no menu suspenso e clicando no botão '`OK`' para concluir a configuração.

3.5 Trabalho mcron de DNS dinâmico

Se o seu ISP (Internet Service Provider) fornece apenas endereços IP dinâmicos, pode ser útil configurar um serviço DNS (Domain Name System) dinâmico (também conhecido como DDNS (Dynamic DNS)) para associar um nome de host estático para um endereço IP público, mas dinâmico. Existem vários serviços que podem ser usados para isso; no job mcron a seguir, DuckDNS (`https://duckdns.org`) é usado. Também deve funcionar com outros serviços DNS dinâmicos que oferecem uma interface semelhante para atualizar o endereço IP.

O Job mcron é fornecido abaixo, onde *DOMAIN* deve ser substituído pelo seu próprio prefixo de domínio, e o token fornecido pelo DuckDNS é associado a *DOMAIN* e adicionado ao arquivo `/etc/duckdns/`*DOMAIN* `arquivo .token`.

```
(define duckdns-job
  ;; Update personal domain IP every 5 minutes.
  #~(job '(next-minute (range 0 60 5))
 #$(program-file
           "duckdns-update"
           (with-extensions (list guile-gnutls) ;required by (web client)
             #~(begin
                 (use-modules (ice-9 textual-ports)
                              (web client))
                 (let ((token (string-trim-both
                               (call-with-input-file "/etc/duckdns/DOMAIN.token"
                                 get-string-all)))
                       (query-template (string-append "https://www.duckdns.org/"
                                                      "update?domains=DOMAIN"
                                                      "&token=~a&ip=")))
                   (http-get (format #f query-template token))))))
         "duckdns-update"
         #:user "nobody"))
```

O Job então precisa ser adicionado à lista de trabalhos mcron do seu sistema, usando algo como:

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

3.6 Conectando-se à VPN Wireguard

Para se conectar a um servidor VPN Wireguard, você precisa que o módulo do kernel esteja carregado na memória e um pacote que forneça ferramentas de rede que o suportem (por exemplo, `wireguard-tools` ou `network-manager`).

Aqui está um exemplo de configuração para Linux-Libre versão menor que 5.6, onde o módulo está fora da árvore e precisa ser carregado manualmente — as seguintes revisões do kernel o possuem integrado e, portanto, não precisam de tal configuração:

```
(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)))
```

Após reconfigurar e reiniciar seu sistema, você pode usar as ferramentas Wireguard ou NetworkManager para conectar-se a um servidor VPN.

3.6.1 Usando ferramentas Wireguard

Para testar a configuração do Wireguard é conveniente usar `wg-quick`. Basta fornecer um arquivo de configuração `wg-quick up ./wg0.conf`; ou coloque esse arquivo em `/etc/wireguard` e execute `wg-quick up wg0`.

> **Nota:** Esteja avisado que o autor descreveu este comando como um: "[. . .] script bash muito rápido e sujo [. . .]".

3.6.2 Usando o NetworkManager

Graças ao suporte do NetworkManager para Wireguard, podemos conectar-nos à nossa VPN usando o comando `nmcli`. Até este ponto, este guia pressupõe que você esteja usando o serviço Network Manager fornecido por `%desktop-services`. Caso contrário, você precisará ajustar sua lista de serviços para carregar `network-manager-service-type` e reconfigurar seu sistema Guix.

Para importar sua configuração VPN, execute o comando "nmcli import":

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

Isso criará um arquivo de configuração em `/etc/NetworkManager/wg0.nmconnection`. Em seguida, conecte-se ao servidor Wireguard:

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

Por padrão, o NetworkManager se conectará automaticamente na inicialização do sistema. Para mudar esse comportamento você precisa editar sua configuração:

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

Para obter informações mais específicas sobre NetworkManager e wireguard consulte este artigo (`https://blogs.gnome.org/thaller/2019/03/15/wireguard-in-networkmanager/`).

3.7 Customizando um Gerenciador de Janelas

3.7.1 StumpWM

Você pode instalar o StumpWM com um sistema Guix adicionando pacotes `stumpwm` e opcionalmente `` `(,stumpwm "lib") `` a um arquivo de configuração do sistema, por exemplo, `/etc/config.scm`.

Um exemplo de configuração pode ser assim:

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

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

Por padrão, o StumpWM usa fontes X11, que podem ser pequenas ou pixeladas em seu sistema. Você pode corrigir isso instalando o módulo StumpWM contrib Lisp `sbcl-ttf-fonts`, adicionando-o aos pacotes do sistema Guix:

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

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

Então você precisa adicionar o seguinte código a um arquivo de configuração do StumpWM `~/.stumpwm.d/init.lisp`:

```
(require :ttf-fonts)
(setf xft:*font-dirs* '("/run/current-system/profile/share/fonts/"))
(setf clx-truetype:+font-cache-filename+ (concat (getenv "HOME")
                                                 "/.fonts/font-cache.sexp"))
(xft:cache-fonts)
(set-font (make-instance 'xft:font :family "DejaVu Sans Mono"
                                   :subfamily "Book" :size 11))
```

3.7.2 Bloqueio de sessão

Dependendo do seu ambiente, o bloqueio da tela da sua sessão pode ser incorporado ou pode ser algo que você mesmo precisa configurar. Se você usa um ambiente de área de trabalho como GNOME ou KDE, ele geralmente está integrado. Se você usa um gerenciador de janelas simples como StumpWM ou EXWM, talvez seja necessário configurá-lo você mesmo.

3.7.2.1 Xorg

Se você usa Xorg, você pode usar o utilitário xss-lock (`https://www.mankier.com/1/xss-lock`) para bloquear a tela da sua sessão. xss-lock é acionado pelo DPMS que, desde o Xorg 1.8, é detectado automaticamente e habilitado se o ACPI também estiver habilitado no tempo de execução do kernel.

Para usar o xss-lock, você pode simplesmente executá-lo e colocá-lo em segundo plano antes de iniciar o gerenciador de janelas, por exemplo. seu `~/.xsession`:

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

Neste exemplo, xss-lock usa `slock` para fazer o bloqueio real da tela quando determina que é apropriado, como quando você suspende seu dispositivo.

Para que o slock possa ser um bloqueador de tela para a sessão gráfica, ele precisa ser definido como setuid-root para poder autenticar usuários e precisa de um serviço PAM. Isso pode ser conseguido adicionando o seguinte serviço ao seu `config.scm`:

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

Se você bloquear sua tela manualmente, por exemplo, chamando slock diretamente quando quiser bloquear sua tela, mas não suspendê-la, é uma boa ideia notificar xss-lock sobre isso para que não ocorra confusão. Isso pode ser feito executando `xset s activate` imediatamente antes de executar o slock.

3.8 Executando Guix em um Servidor Linode

Para executar o Guix num servidor hospedado por Linode (`https://www.linode.com`), comece com um servidor Debian recomendado. Recomendamos usar a distribuição padrão como forma de inicializar o Guix. Crie suas chaves SSH.

```
ssh-keygen
```

Certifique-se de adicionar sua chave SSH para facilitar o login no servidor remoto. Isto é feito trivialmente através da interface gráfica do Linode para adicionar chaves SSH. Vá para o seu perfil e clique em adicionar chave SSH. Copie nele a saída de:

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

Desligue o Linode.

Na aba Armazenamento do Linode, redimensione o disco Debian para ser menor. Recomenda-se 30 GB de espaço livre. Em seguida, clique em "Adicionar um disco" e preencha o formulário com o seguinte:

- Label: "Guix"
- Filesystem: ext4
- Defina-o para o tamanho restante

Na aba Configurações, pressione “Editar” no perfil Debian padrão. Em "Bloquear atribuição de dispositivo", clique em "Adicionar um dispositivo". Deve ser `/dev/sdc` e você pode selecionar o disco "Guix". Salvar alterações.

Agora "Adicionar uma configuração", com o seguinte:

- Label: Guix
- Kernel:GRUB 2 (está na parte inferior! Esta etapa é **IMPORTANTE!**)
- Bloquear atribuição de dispositivo:
- `/dev/sda`: Guix

- `/dev/sdb`: swap
- Root device: `/dev/sda`
- Desligue todos os auxiliares de sistema de arquivos/inicialização

Agora ligue-o novamente, inicializando com a configuração do Debian. Quando estiver em execução, faça ssh para o seu servidor via `ssh root@<your-server-IP-here>`. (Você pode encontrar o endereço IP do seu servidor na seção Resumo do Linode.) Agora você pode executar as etapas "instalar o guix de veja Seção “Instalação de binários” em *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
```

Agora é hora de escrever uma configuração para o servidor. As principais informações estão abaixo. Salve o arquivo resultante como `guix-config.scm`.

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

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


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

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

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

(packages (cons* 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)))
```

Substitua os seguintes campos na configuração acima:

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

A última linha no exemplo acima permite que você faça login no servidor como root e defina a senha root inicial (veja a nota no final desta receita sobre login root). Depois de fazer isso, você pode excluir essa linha da sua configuração e reconfigurar para evitar o login root.

Copie sua chave pública ssh (por exemplo: `~/.ssh/id_rsa.pub`) como `<seu-nome-de-usuário-aqui>_rsa.pub` e coloque `guix-config.scm` no mesmo diretório. Em um novo terminal execute estes comandos.

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

No seu primeiro terminal, monte o drive guix:

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

Devido à forma como configuramos a seção bootloader do guix-config.scm, apenas o arquivo de configuração grub será instalado. Então, precisamos copiar algumas das outras coisas do GRUB já instaladas no sistema Debian:

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

Agora inicialize a instalação do Guix:

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

Ok, desligue-o! Agora, no console Linode, selecione boot e selecione "Guix".

Depois de inicializar, você poderá fazer login via SSH! (A configuração do servidor terá mudado.) Você pode encontrar um erro como:

```
$ 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.
```

Exclua o arquivo `~/.ssh/known_hosts` ou exclua a linha incorreta começando com o endereço IP do seu servidor.

Certifique-se de definir sua senha e a senha do root.

```
ssh root@<endereço IP remoto>
passwd; para a senha root
passwd <nome de usuário>; para a senha do usuário
```

Talvez você não consiga executar os comandos acima neste momento. Se você tiver problemas para fazer login remotamente em sua caixa linode via SSH, então você ainda pode precisar definir sua senha root e de usuário inicialmente clicando na opção “Launch Console” em seu linode. Escolha “Glish” em vez de “Weblish”. Agora você deve conseguir fazer o ssh na máquina.

Viva! Neste ponto você pode desligar o servidor, excluir o disco Debian e redimensionar o Guix para o restante do tamanho. Parabéns!

A propósito, se você salvá-lo como uma imagem de disco neste momento, será fácil criar novas imagens Guix! Pode ser necessário reduzir o tamanho da imagem Guix para 6144 MB para salvá-la como uma imagem. Então você pode redimensioná-lo novamente para o tamanho máximo.

3.9 Running Guix on a Kimsufi Server

Para executar o Guix em um servidor hospedado por Kimsufi (`https://www.kimsufi.com/`), clique na guia netboot, selecione Rescue64-pro e reinicie.

A OVH enviar-lhe-á por e-mail as credenciais necessárias para efetuar o ssh num sistema Debian.

Agora você pode executar as etapas "instalar guix de veja Seção “Instalação de binários” em *GNU Guix*":

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

Particione as unidades e formate-as, primeiro interrompa a matriz raid:

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

Em seguida, limpe os discos e configure as partições, criaremos uma matriz RAID 1.

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

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

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

Crie a matriz:

```
mdadm --create /dev/md127 --level=1 --raid-disks=2 \
  --metadata=0.90 /dev/sda2 /dev/sdb2
```

Agora crie sistemas de arquivos nas partições relevantes, primeiro as partições de inicialização:

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

Então a partição raiz:

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

Inicialize as partições swap:

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

Monte a unidade guix:

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

Agora é hora de escrever um arquivo `os.scm` de declaração do sistema operacional; aqui está uma amostra:

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

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

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

  ;; Add a kernel module for RAID-1 (aka. "mirror").
  (initrd-modules (cons* "raid1"  %base-initrd-modules))

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

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

  (issue
   ;; Default contents for /etc/issue.
   "\
This is the GNU system at Kimsufi.  Welcome.\n")

  (file-systems (cons* (file-system
         (mount-point "/")
         (device "/dev/md127")
         (type "ext4")
```

```
          (dependencies mapped-devices))
        %base-file-systems))

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

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

  ;; Globally-installed packages.
  (packages (cons* tmux gnutls wireguard-tools %base-packages))
  (services
   (cons*
    (service static-networking-service-type
     (list (static-networking
    (addresses (list (network-address
      (device "enp3s0")
      (value "server-ip-address/24"))))
    (routes (list (network-route
   (destination "default")
   (gateway "server-gateway"))))
    (name-servers '("213.186.33.99")))))

    (service unattended-upgrade-service-type)

    (service openssh-service-type
     (openssh-configuration
      (openssh openssh-sans-x)
      (permit-root-login #f)
      (authorized-keys
       `(("guix" ,(plain-file "ssh-key-name.pub"
                              "ssh-public-key-content"))))))
    (modify-services %base-services
      (sysctl-service-type
       config =>
       (sysctl-configuration
(settings (append '(("net.ipv6.conf.all.autoconf" . "0")
    ("net.ipv6.conf.all.accept_ra" . "0"))
  %default-sysctl-settings)))))))
```

Não se esqueça de substituir as variáveis *server-ip-address*, *server-gateway*, *ssh-key-name* e *ssh-public-key-content* pelos seus próprias valores.

O gateway é o último IP utilizável em seu bloco, portanto, se você tiver um servidor com IP '`37.187.79.10`', seu gateway será '`37.187.79.254`'.

Transfira o arquivo de declaração do sistema operacional `os.scm` para o servidor por meio dos comandos `scp` ou `sftp`.

Agora só falta instalar o Guix com `guix system init` e reiniciar.

No entanto, primeiro precisamos configurar um chroot, porque a partição raiz do sistema de recuperação é montada em uma partição aufs e se você tentar instalar o Guix ele falhará na etapa de instalação do GRUB reclamando do caminho canônico de "aufs".

Instale os pacotes que serão usados no chroot:

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

Em seguida, execute o seguinte para criar os diretórios necessários para o chroot:

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

Copie o host resolv.conf no chroot:

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

Monte os dispositivos de bloco, a loja e seu banco de dados e a configuração atual do guix:

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

Faça chroot em /mnt e instale o sistema:

```
chroot /mnt/ /bin/bash

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

Por fim, na interface do usuário (IU) da web, altere '`netboot`' para '`boot to disk`' e reinicie (também na IU da web).

Aguarde alguns minutos e tente fazer ssh com `ssh guix@`*`endereço IP do servidor`*`> -i `*`caminho para sua chave ssh`*

Você deve ter um sistema Guix instalado e funcionando no Kimsufi; Parabéns!

3.10 Configurando uma montagem vinculada

Para vincular a montagem de um sistema de arquivos, é necessário primeiro configurar algumas definições antes da seção `operating-system` da definição do sistema. Neste exemplo, vincularemos a montagem de uma pasta de uma unidade de disco rígido a `/tmp`, para evitar desgaste no SSD primário, sem dedicar uma partição inteira para ser montada como `/tmp`.

Primeiro, a unidade de origem que hospeda a pasta que desejamos vincular a montagem deve ser definida, para que a montagem de ligação possa depender dela.

```
(define source-drive ;; "source-drive" can be named anything you want.
   (file-system
    (device (uuid "UUID goes here"))
    (mount-point "/path-to-spinning-disk-goes-here")
    (type "ext4"))) ;Make sure to set this to the appropriate type for your drive.
```

A pasta de origem também deve ser definida, para que o guix saiba que não é um dispositivo de bloco normal, mas uma pasta.

```
;; "%source-directory" pode receber qualquer nome de variável válido.
(define (%source-directory) "/caminho_para_ o_disco_vai_aqui/tmp")
```

Finalmente, dentro da definição `file-systems`, devemos adicionar a própria montagem.

```
(file-systems (cons*

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

               ;; Deve corresponder ao nome que você deu à unidade de origem
               ;; na definição anterior.
               source-drive

               (file-system
                ;; Make sure "source-directory" matches your earlier definition.
                (device (%source-directory))
                (mount-point "/tmp")
                ;; We are mounting a folder, not a partition, so this type needs to b
                (type "none")
                (flags '(bind-mount))
                ;; Ensure "source-drive" matches what you've named the variable for t
                (dependencies (list source-drive))
                )

                ...<outras unidades omitidas para maior clareza>...

               ))
```

3.11 Obtendo substitutos pelo Tor

O daemon Guix pode usar um proxy HTTP para obter substitutos, aqui estamos configurando-o para obtê-los via Tor.

> **Aviso:** *Nem todo* o tráfego do daemon Guix passará pelo Tor! Somente HTTP/HTTPS será encaminhado ao proxy; As conexões FTP, o protocolo Git, SSH, etc. ainda passarão pela rede aberta. Novamente, esta configuração não é infalível, pois parte do seu tráfego não será roteado pelo Tor. Use-o por sua conta e risco.
>
> Observe também que o procedimento descrito aqui se aplica apenas à substituição de pacotes. Ao atualizar sua distribuição guix com `guix pull`, você

ainda precisará usar `torsocks` se quiser rotear a conexão para os servidores de repositório git do guix através do Tor.

O servidor substituto do Guix está disponível como um serviço Onion, se você quiser usá-lo para obter seus substitutos através do Tor, configure seu sistema da seguinte forma:

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

(operating-system
  ...
  (services
    (cons
      (service tor-service-type
               (tor-configuration
                 (config-file (plain-file "tor-config"
                                          "HTTPTunnelPort 127.0.0.1:9250"))))
      (modify-services %base-services
        (guix-service-type
          config => (guix-configuration
                      (inherit config)
                      ;; ci.guix.gnu.org's Onion service
                      (substitute-urls "\
https://4zwzi66wwdaalbhgnix55ea3ab4pvvw66ll2ow53kjub6se4q2bclcyd.onion")
                      (http-proxy "http://localhost:9250")))))))
```

Isso manterá um processo tor em execução que fornece um túnel HTTP CONNECT que será usado por `guix-daemon`. O daemon pode usar outros protocolos além do HTTP(S) para obter recursos remotos. A solicitação usando esses protocolos não passará pelo Tor, pois estamos apenas configurando um túnel HTTP aqui. Observe que `substitutes-urls` está usando HTTPS e não HTTP ou não funcionará, isso é uma limitação do túnel do Tor; você pode querer usar `privoxy` para evitar tais limitações.

Se você não deseja sempre obter substitutos através do Tor, mas usá-lo apenas algumas vezes, pule o `guix-configuration`. Quando você deseja obter um substituto da execução do túnel Tor:

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

3.12 Configurando NGINX com Lua

O NGINX pode ser estendido com scripts Lua.

Guix fornece serviço NGINX com capacidade de carregar módulos Lua e pacotes Lua específicos, e responder a solicitações avaliando scripts Lua.

O exemplo a seguir demonstra a definição do sistema com configuração para avaliar o script Lua `index.lua` na solicitação HTTP para o endpoint `http://localhost/hello`:

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

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

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

ngx.say(stdout)
```

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

3.13 Servidor de música com áudio Bluetooth

MPD, the Music Player Daemon, is a flexible server-side application for playing music. Client programs on different machines on the network — a mobile phone, a laptop, a desktop workstation — can connect to it to control the playback of audio files from your local music collection. MPD decodes the audio files and plays them back on one or many outputs.

By default MPD will play to the default audio device. In the example below we make things a little more interesting by setting up a headless music server. There will be no graphical user interface, no Pulseaudio daemon, and no local audio output. Instead we will

configure MPD with two outputs: a bluetooth speaker and a web server to serve audio streams to any streaming media player.

Bluetooth is often rather frustrating to set up. You will have to pair your Bluetooth device and make sure that the device is automatically connected as soon as it powers on. The Bluetooth system service returned by the `bluetooth-service` procedure provides the infrastructure needed to set this up.

Reconfigure your system with at least the following services and packages:

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

Start the `bluetooth` service and then use `bluetoothctl` to scan for Bluetooth devices. Try to identify your Bluetooth speaker and pick out its device ID from the resulting list of devices that is indubitably dominated by a baffling smorgasbord of your neighbors' home automation gizmos. This only needs to be done once:

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

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

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

[bluetooth]# default-agent
[bluetooth]# Default agent request successful

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

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

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

```
Pairing successful

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

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

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

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

Congratulations, you can now automatically connect to your Bluetooth speaker!

It is now time to configure ALSA to use the *bluealsa* Bluetooth module, so that you can define an ALSA pcm device corresponding to your Bluetooth speaker. For a headless server using *bluealsa* with a fixed Bluetooth device is likely simpler than configuring Pulseaudio and its stream switching behavior. We configure ALSA by crafting a custom `alsa-configuration` for the `alsa-service-type`. The configuration will declare a `pcm` type `bluealsa` from the `bluealsa` module provided by the `bluez-alsa` package, and then define a `pcm` device of that type for your Bluetooth speaker.

All that is left then is to make MPD send audio data to this ALSA device. We also add a secondary MPD output that makes the currently played audio files available as a stream through a web server on port 8080. When enabled a device on the network could listen to the audio stream by connecting any capable media player to the HTTP server on port 8080, independent of the status of the Bluetooth speaker.

What follows is the outline of an `operating-system` declaration that should accomplish the above-mentioned tasks:

```
(use-modules (gnu))
(use-service-modules audio dbus sound #;... etc)
(use-package-modules audio linux #;... etc)
(operating-system
  ;; ...
  (packages (cons* bluez bluez-alsa
                   %base-packages))
  (services
   ;; ...
   (service mpd-service-type
            (mpd-configuration
```

```
                (user "your-username")
                (music-dir "/path/to/your/music")
                (address "192.168.178.20")
                (outputs (list (mpd-output
                                (type "alsa")
                                (name "MPD")
                                (extra-options
                                 ;; Use the same name as in the ALSA
                                 ;; configuration below.
                                 '((device . "pcm.btspeaker"))))
                               (mpd-output
                                (type "httpd")
                                (name "streaming")
                                (enabled? #false)
                                (always-on? #true)
                                (tags? #true)
                                (mixer-type 'null)
                                (extra-options
                                 '((encoder . "vorbis")
                                   (port    . "8080")
                                   (bind-to-address . "192.168.178.20")
                                   (max-clients . "0") ;no limit
                                   (quality . "5.0")
                                   (format  . "44100:16:1")))))))
   (dbus-service #:services (list bluez-alsa))
   (bluetooth-service #:auto-enable? #t)
   (service alsa-service-type
            (alsa-configuration
             (pulseaudio? #false) ;we don't need it
             (extra-options
              #~(string-append "\
# Declare Bluetooth audio device type \"bluealsa\" from bluealsa module
pcm_type.bluealsa {
    lib \""
#$(file-append bluez-alsa "/lib/alsa-lib/libasound_module_pcm_bluealsa.so") "\"
}

# Declare control device type \"bluealsa\" from the same module
ctl_type.bluealsa {
    lib \""
#$(file-append bluez-alsa "/lib/alsa-lib/libasound_module_ctl_bluealsa.so") "\"
}

# Define the actual Bluetooth audio device.
pcm.btspeaker {
    type bluealsa
    device \"AA:BB:CC:A4:AA:CD\" # unique device identifier
```

```
        profile \"a2dp\"
    }

    # Define an associated controller.
    ctl.btspeaker {
        type bluealsa
    }
    "))))))
```

Enjoy the music with the MPD client of your choice or a media player capable of streaming via HTTP!

4 Contêineres

The kernel Linux provides a number of shared facilities that are available to processes in the system. These facilities include a shared view on the file system, other processes, network devices, user and group identities, and a few others. Since Linux 3.19 a user can choose to *unshare* some of these shared facilities for selected processes, providing them (and their child processes) with a different view on the system.

A process with an unshared `mount` namespace, for example, has its own view on the file system — it will only be able to see directories that have been explicitly bound in its mount namespace. A process with its own `proc` namespace will consider itself to be the only process running on the system, running as PID 1.

Guix uses these kernel features to provide fully isolated environments and even complete Guix System containers, lightweight virtual machines that share the host system's kernel. This feature comes in especially handy when using Guix on a foreign distribution to prevent interference from foreign libraries or configuration files that are available system-wide.

4.1 Contêineres Guix

The easiest way to get started is to use `guix shell` with the `--container` option. Veja Seção "Invoking guix shell" em *GNU Guix Reference Manual* for a reference of valid options.

The following snippet spawns a minimal shell process with most namespaces unshared from the system. The current working directory is visible to the process, but anything else on the file system is unavailable. This extreme isolation can be very useful when you want to rule out any sort of interference from environment variables, globally installed libraries, or configuration files.

```
guix shell --container
```

It is a bleak environment, barren, desolate. You will find that not even the GNU coreutils are available here, so to explore this deserted wasteland you need to use built-in shell commands. Even the usually gigantic `/gnu/store` directory is reduced to a faint shadow of itself.

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

There isn't much you can do in an environment like this other than exiting it. You can use `^D` or `exit` to terminate this limited shell environment.

You can make other directories available inside of the container environment; use `--expose=DIRECTORY` to bind-mount the given directory as a read-only location inside the container, or use `--share=DIRECTORY` to make the location writable. With an additional mapping argument after the directory name you can control the name of the directory

inside the container. In the following example we map `/etc` on the host system to `/the/host/etc` inside a container in which the GNU coreutils are installed.

```
$ guix shell --container --share=/etc=/the/host/etc coreutils
$ ls /the/host/etc
```

Similarly, you can prevent the current working directory from being mapped into the container with the `--no-cwd` option. Another good idea is to create a dedicated directory that will serve as the container's home directory, and spawn the container shell from that directory.

On a foreign system a container environment can be used to compile software that cannot possibly be linked with system libraries or with the system's compiler toolchain. A common use-case in a research context is to install packages from within an R session. Outside of a container environment there is a good chance that the foreign compiler toolchain and incompatible system libraries are found first, resulting in incompatible binaries that cannot be used by R. In a container shell this problem disappears, as system libraries and executables simply aren't available due to the unshared `mount` namespace.

Let's take a comprehensive manifest providing a comfortable development environment for use with R:

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

        ;; base packages
        "bash-minimal"
        "glibc-locales"
        "nss-certs"

        ;; Common command line tools lest the container is too empty.
        "coreutils"
        "grep"
        "which"
        "wget"
        "sed"

        ;; R markdown tools
        "pandoc"

        ;; Toolchain and common libraries for "install.packages"
        "gcc-toolchain@10"
        "gfortran-toolchain"
        "gawk"
        "tar"
        "gzip"
        "unzip"
        "make"
        "cmake"
        "pkg-config"
        "cairo"
```

```
        "libxt"
        "openssl"
        "curl"
        "zlib"))
```

Let's use this to run R inside a container environment. For convenience we share the `net` namespace to use the host system's network interfaces. Now we can build R packages from source the traditional way without having to worry about ABI mismatch or incompatibilities.

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

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

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

Using container shells is fun, but they can become a little cumbersome when you want to go beyond just a single interactive process. Some tasks become a lot easier when they sit on the rock solid foundation of a proper Guix System and its rich set of system services. The next section shows you how to launch a complete Guix System inside of a container.

4.2 Contêineres do Sistema Guix

The Guix System provides a wide array of interconnected system services that are configured declaratively to form a dependable stateless GNU System foundation for whatever tasks you throw at it. Even when using Guix on a foreign distribution you can benefit from the design of Guix System by running a system instance as a container. Using the same kernel features of unshared namespaces mentioned in the previous section, the resulting Guix System instance is isolated from the host system and only shares file system locations that you explicitly declare.

A Guix System container differs from the shell process created by `guix shell --container` in a number of important ways. While in a container shell the containerized process is a Bash shell process, a Guix System container runs the Shepherd as PID 1. In a system container all system services (veja Seção "Services" em *GNU Guix Reference Manual*) are set up just as they would be on a Guix System in a virtual machine or on

bare metal—this includes daemons managed by the GNU Shepherd (veja Seção “Shepherd Services” em *GNU Guix Reference Manual*) as well as other kinds of extensions to the operating system (veja Seção “Service Composition” em *GNU Guix Reference Manual*).

The perceived increase in complexity of running a Guix System container is easily justified when dealing with more complex applications that have higher or just more rigid requirements on their execution contexts—configuration files, dedicated user accounts, directories for caches or log files, etc. In Guix System the demands of this kind of software are satisfied through the deployment of system services.

4.2.1 Um banco de dados de contêineres

A good example might be a PostgreSQL database server. Much of the complexity of setting up such a database server is encapsulated in this deceptively short service declaration:

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

A complete operating system declaration for use with a Guix System container would look something like this:

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

(operating-system
  (host-name "container")
  (timezone "Europe/Berlin")
  (file-systems (cons (file-system
                        (device (file-system-label "does-not-matter"))
                        (mount-point "/")
                        (type "ext4"))
                      %base-file-systems))
  (bootloader (bootloader-configuration
               (bootloader grub-bootloader)
               (targets '("/dev/sdX"))))
  (services
   (cons* (service postgresql-service-type
                  (postgresql-configuration
                   (postgresql postgresql-14)
                   (config-file
                    (postgresql-config-file
                     (log-destination "stderr")
                     (hba-file
                      (plain-file "pg_hba.conf"
                                  "\
local all all trust
host all all 10.0.0.1/32  trust"))
                     (extra-config
                      '(("listen_addresses" "*")
```

```
                ("log_directory"    "/var/log/postgresql")))))))
        (service postgresql-role-service-type
                 (postgresql-role-configuration
                  (roles
                   (list (postgresql-role
                          (name "test")
                          (create-database? #t))))))
        %base-services)))
```

With `postgresql-role-service-type` we define a role "test" and create a matching database, so that we can test right away without any further manual setup. The `postgresql-config-file` settings allow a client from IP address 10.0.0.1 to connect without requiring authentication—a bad idea in production systems, but convenient for this example.

Let's build a script that will launch an instance of this Guix System as a container. Write the `operating-system` declaration above to a file `os.scm` and then use `guix system container` to build the launcher. (veja Seção "Invoking guix system" em *GNU Guix Reference Manual*).

```
$ guix system container os.scm
The following derivations will be built:
  /gnu/store/...-run-container.drv
  ...
building /gnu/store/...-run-container.drv...
/gnu/store/...-run-container
```

Now that we have a launcher script we can run it to spawn the new system with a running PostgreSQL service. Note that due to some as yet unresolved limitations we need to run the launcher as the root user, for example with `sudo`.

```
$ sudo /gnu/store/...-run-container
system container is running as PID 5983
...
```

Background the process with `Ctrl-z` followed by `bg`. Note the process ID in the output; we will need it to connect to the container later. You know what? Let's try attaching to the container right now. We will use `nsenter`, a tool provided by the `util-linux` package:

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

The PostgreSQL service is running in the container!

4.2.2 Rede em contêineres

What good is a Guix System running a PostgreSQL database service as a container when we can only talk to it with processes originating in the container? It would be much better if we could talk to the database over the network.

The easiest way to do this is to create a pair of connected virtual Ethernet devices (known as `veth`). We move one of the devices (`ceth-test`) into the `net` namespace of the container and leave the other end (`veth-test`) of the connection on the host system.

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

# Attach the new net namespace "guix-test" to the container PID.
sudo ip netns attach $ns $pid

# Create the pair of devices
sudo ip link add $host type veth peer name $client

# Move the client device into the container's net namespace
sudo ip link set $client netns $ns
```

Then we configure the host side:

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

. . . and then we configure the client side:

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

At this point the host can reach the container at IP address 10.0.0.2, and the container can reach the host at IP 10.0.0.1. This is all we need to talk to the database server inside the container from the host system on the outside.

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

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

Now that we're done with this little demonstration let's clean up:

```
sudo kill $pid
```

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

5 Máquinas Virtuais

Guix can produce disk images (veja Seção “Invoking guix system” em *GNU Guix Reference Manual*) that can be used with virtual machines solutions such as virt-manager, GNOME Boxes or the more bare QEMU, among others.

This chapter aims to provide hands-on, practical examples that relates to the usage and configuration of virtual machines on a Guix System.

5.1 Ponte de rede para QEMU

By default, QEMU uses a so-called “user mode” host network back-end, which is convenient as it does not require any configuration. Unfortunately, it is also quite limited. In this mode, the guest VM (virtual machine) can access the network the same way the host would, but it cannot be reached from the host. Additionally, since the QEMU user networking mode relies on ICMP, ICMP-based networking tools such as `ping` do *not* work in this mode. Thus, it is often desirable to configure a network bridge, which enables the guest to fully participate in the network. This is necessary, for example, when the guest is to be used as a server.

5.1.1 Creating a network bridge interface

There are many ways to create a network bridge. The following command shows how to use NetworkManager and its `nmcli` command line interface (CLI) tool, which should already be available if your operating system declaration is based on one of the desktop templates:

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

To have this bridge be part of your network, you must associate your network bridge with the Ethernet interface used to connect with the network. Assuming your interface is named ‘`enp2s0`’, the following command can be used to do so:

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

Importante: Only Ethernet interfaces can be added to a bridge. For wireless interfaces, consider the routed network approach detailed in Veja Seção 5.2 [Roteamento de rede para libvirt], Página 58.

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

5.1.2 Configuring the QEMU bridge helper script

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

```
(setuid-programs
 (cons (file-append qemu "/libexec/qemu-bridge-helper")
       %setuid-programs))
```

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

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

5.1.3 Invoking QEMU with the right command line options

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

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

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

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

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

5.1.4 Networking issues caused by Docker

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

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

5.2 Roteamento de rede para libvirt

If the machine hosting your virtual machines is connected wirelessly to the network, you won't be able to use a true network bridge as explained in the preceding section (veja Seção 5.1 [Ponte de rede para QEMU], Página 57). In this case, the next best option is to use a *virtual* bridge with static routing and to configure a libvirt-powered virtual machine to use it (via the `virt-manager` GUI for example). This is similar to the default mode of operation of QEMU/libvirt, except that instead of using NAT (Network Address Translation), it relies on static routes to join the VM (virtual machine) IP address to the LAN (local area network). This provides two-way connectivity to and from the virtual machine, which is needed for exposing services hosted on the virtual machine.

5.2.1 Creating a virtual network bridge

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

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

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

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

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

5.2.2 Configuring the static routes for your virtual bridge

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

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

'`Interface`'
: lan

'`Target`'
: 192.168.2.0

'`IPv4-Netmask`'
: 255.255.255.0

'`IPv4-Gateway`'
: *server-ip*

'`Route type`'
: unicast

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

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

6 Gerenciamento avançado de pacotes

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

veja Seção "Package Management" em *GNU Guix Reference Manual* for a complete reference.

6.1 Perfis Guix na Prática

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

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

> **Nota:** This section is an opinionated guide on the use of multiple profiles. It predates `guix shell` and its fast profile cache (veja Seção "Invoking guix shell" em *GNU Guix Reference Manual*).
>
> In many cases, you may find that using `guix shell` to set up the environment you need, when you need it, is less work that maintaining a dedicated profile. Your call!

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

Multiple profiles have many benefits:

- Clean semantic separation of the various packages a user needs for different contexts.
- Multiple profiles can be made available into the environment either on login or within a dedicated shell.
- Profiles can be loaded on demand. For instance, the user can use multiple shells, each of them running different profiles.
- Isolation: Programs from one profile will not use programs from the other, and the user can even install different versions of the same programs to the two profiles without conflict.
- Deduplication: Profiles share dependencies that happens to be the exact same. This makes multiple profiles storage-efficient.
- Reproducible: when used with declarative manifests, a profile can be fully specified by the Guix commit that was active when it was set up. This means that the exact same profile can be set up anywhere and anytime (`https://guix.gnu.org/blog/2018/multi-dimensional-transactions-and-rollbacks-oh-my/`), with just the commit information. See the section on Seção 6.1.5 [Perfis reproduzíveis], Página 65.

- Easier upgrades and maintenance: Multiple profiles make it easy to keep package listings at hand and make upgrades completely frictionless.

Concretely, here follows some typical profiles:

- The dependencies of a project you are working on.
- Your favourite programming language libraries.
- Laptop-specific programs (like '`powertop`') that you don't need on a desktop.
- TEXlive (this one can be really useful when you need to install just one package for this one document you've just received over email).
- Games.

Let's dive in the set up!

6.1.1 Configuração básica com manifestos

A Guix profile can be set up *via* a *manifest*. A manifest is a snippet of Scheme code that specifies the set of packages you want to have in your profile; it looks like this:

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

Veja Seção "Writing Manifests" em *GNU Guix Reference Manual*, for more information about the syntax.

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

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

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

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

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

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

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

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

```
for i in $GUIX_EXTRA_PROFILES/*; do
  profile=$i/$(basename "$i")
```

```
  if [ -f "$profile"/etc/profile ]; then
    GUIX_PROFILE="$profile"
    . "$GUIX_PROFILE"/etc/profile
  fi
  unset profile
done
```

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

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

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

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

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

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

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

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

To upgrade a profile, simply install the manifest again:

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

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

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

Each profile has its own generations:

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

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

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

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

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

6.1.2 Pacotes necessários

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

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

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

- Either export the variable manually, e.g.

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

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

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

6.1.3 Perfil padrão

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

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

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

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

without having to specify the path to a profile.

6.1.4 Os benefícios dos manifestos

Manifests let you *declare* the set of packages you'd like to have in a profile (veja Seção "Writing Manifests" em *GNU Guix Reference Manual*). They are a convenient way to keep your package lists around and, say, to synchronize them across multiple machines using a version control system.

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

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

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

- When a profile is set up from a manifest, the manifest itself is self-sufficient to keep a "package listing" around and reinstall the profile later or on a different system. For ad-hoc profiles, we would need to generate a manifest specification manually and maintain the package versions for the packages that don't use the default version.

- `guix package --upgrade` always tries to update the packages that have propagated inputs, even if there is nothing to do. Guix manifests remove this problem.
- When partially upgrading a profile, conflicts may arise (due to diverging dependencies between the updated and the non-updated packages) and they can be annoying to resolve manually. Manifests remove this problem altogether since all packages are always upgraded at once.
- As mentioned above, manifests allow for reproducible profiles, while the imperative `guix install`, `guix upgrade`, etc. do not, since they produce different profiles every time even when they hold the same packages. See the related discussion on the matter (`https://issues.guix.gnu.org/issue/33285`).
- Manifest specifications are usable by other '`guix`' commands. For example, you can run `guix weather -m manifest.scm` to see how many substitutes are available, which can help you decide whether you want to try upgrading today or wait a while. Another example: you can run `guix pack -m manifest.scm` to create a pack containing all the packages in the manifest (and their transitive references).
- Finally, manifests have a Scheme representation, the '`<manifest>`' record type. They can be manipulated in Scheme and passed to the various Guix APIs (`https://en.wikipedia.org/wiki/Api`).

It's important to understand that while manifests can be used to declare profiles, they are not strictly equivalent: profiles have the side effect that they "pin" packages in the store, which prevents them from being garbage-collected (veja Seção "Invoking guix gc" em *GNU Guix Reference Manual*) and ensures that they will still be available at any point in the future. The `guix shell` command also protects recently-used profiles from garbage collection; profiles that have not been used for a while may be garbage-collected though, along with the packages they refer to.

To be 100% sure that a given profile will never be collected, install the manifest to a profile and use `GUIX_PROFILE=/the/profile; . "$GUIX_PROFILE"/etc/profile` as explained above: this guarantees that our hacking environment will be available at all times.

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

6.1.5 Perfis reproduzíveis

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

- a manifest (veja Seção "Writing Manifests" em *GNU Guix Reference Manual*);
- a Guix channel specification (veja Seção "Replicating Guix" em *GNU Guix Reference Manual*).

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

You can output the Guix channel specification with '`guix describe --format=channels`' (veja Seção "Invoking guix describe" em *GNU Guix Reference Manual*). Save this to a file, say '`channel-specs.scm`'.

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

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

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

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

mkdir -p "$GUIX_EXTRA_PROFILES/my-project"
"$GUIX_EXTRA"/my-project/guix/bin/guix package \
 --manifest=/path/to/guix-my-project-manifest.scm \
 --profile="$GUIX_EXTRA_PROFILES"/my-project/my-project
```

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

7 Desenvolvimento de software X

Guix is a handy tool for developers; `guix shell`, in particular, gives a standalone development environment for your package, no matter what language(s) it's written in (veja Seção "Invoking guix shell" em *GNU Guix Reference Manual*). To benefit from it, you have to initially write a package definition and have it either in Guix proper, or in a channel, or directly in your project's source tree as a `guix.scm` file. This last option is appealing: all developers have to do to get set up is clone the project's repository and run `guix shell`, with no arguments.

Development needs go beyond development environments though. How can developers perform continuous integration of their code in Guix build environments? How can they deliver their code straight to adventurous users? This chapter describes a set of files developers can add to their repository to set up Guix-based development environments, continuous integration, and continuous delivery—all at once[1].

7.1 Começando

How do we go about "Guixifying" a repository? The first step, as we've seen, will be to add a `guix.scm` at the root of the repository in question. We'll take Guile (`https://www.gnu.org/software/guile`) as an example in this chapter: it's written in Scheme (mostly) and C, and has a number of dependencies—a C compilation tool chain, C libraries, Autoconf and its friends, LaTeX, and so on. The resulting `guix.scm` looks like the usual package definition (veja Seção "Defining Packages" em *GNU Guix Reference Manual*), just without the `define-public` bit:

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

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

[1] This chapter is adapted from a blog post (`https://guix.gnu.org/en/blog/2023/from-development-environments-to-continuous-integrationthe-ultimate-guide-to-software-development-with-guix/`) published in June 2023 on the Guix web site.

```
              (gnu packages tex)
              (gnu packages texinfo)
              (gnu packages version-control))

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

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

  (native-search-paths
   (list (search-path-specification
          (variable "GUILE_LOAD_PATH")
          (files '("share/guile/site/3.0")))
         (search-path-specification
          (variable "GUILE_LOAD_COMPILED_PATH")
          (files '("lib/guile/3.0/site-ccache")))))
  (synopsis "Scheme implementation intended especially for extensions")
  (description
   "Guile is the GNU Ubiquitous Intelligent Language for Extensions,
and it's actually a full-blown Scheme implementation!")
```

```
  (home-page "https://www.gnu.org/software/guile/")
  (license license:lgpl3+))
```

Quite a bit of boilerplate, but now someone who'd like to hack on Guile now only needs to run:

```
guix shell
```

That gives them a shell containing all the dependencies of Guile: those listed above, but also *implicit dependencies* such as the GCC tool chain, GNU Make, sed, grep, and so on. Veja Seção "Invoking guix shell" em *GNU Guix Reference Manual*, for more info on `guix shell`.

The chef's recommendation: Our suggestion is to create development environments like this:

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

... or, for short:

```
guix shell -CP
```

That gives a shell in an isolated container, and all the dependencies show up in `$HOME/.guix-profile`, which plays well with caches such as `config.cache` (veja Seção "Cache Files" em *Autoconf*) and absolute file names recorded in generated `Makefile`s and the likes. The fact that the shell runs in a container brings peace of mind: nothing but the current directory and Guile's dependencies is visible inside the container; nothing from the system can possibly interfere with your development.

7.2 Level 1: Building with Guix

Now that we have a package definition (veja Seção 7.1 [Começando], Página 67), why not also take advantage of it so we can build Guile with Guix? We had left the `source` field empty, because `guix shell` above only cares about the *inputs* of our package—so it can set up the development environment—not about the package itself.

To build the package with Guix, we'll need to fill out the `source` field, along these lines:

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

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

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

Here's what we changed compared to the previous section:

1. We added `(guix git-download)` to our set of imported modules, so we can use its `git-predicate` procedure.
2. We defined `vcs-file?` as a procedure that returns true when passed a file that is under version control. For good measure, we add a fallback case for when we're not in a Git checkout: always return true.
3. We set `source` to a `local-file` (`https://guix.gnu.org/manual/devel/en/html_node/G_002dExpressions.html#index-local_002dfile`)—a recursive copy of the current directory (`"."`), limited to files under version control (the `#:select?` bit).

From there on, our `guix.scm` file serves a second purpose: it lets us build the software with Guix. The whole point of building with Guix is that it's a "clean" build—you can be sure nothing from your working tree or system interferes with the build result—and it lets you test a variety of things. First, you can do a plain native build:

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

But you can also build for another system (possibly after setting up veja Seção "Daemon Offload Setup" em *GNU Guix Reference Manual* or veja Seção "Virtualization Services" em *GNU Guix Reference Manual*):

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

. . . or cross-compile:

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

You can also use *package transformations* to test package variants (veja Seção "Package Transformation Options" em *GNU Guix Reference Manual*):

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

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

Handy!

7.3 Level 2: The Repository as a Channel

We now have a Git repository containing (among other things) a package definition (veja Seção 7.2 [Construindo com Guix], Página 69). Can't we turn it into a *channel* (veja Seção "Channels" em *GNU Guix Reference Manual*)? After all, channels are designed to ship package definitions to users, and that's exactly what we're doing with our `guix.scm`.

Turns out we can indeed turn it into a channel, but with one caveat: we must create a separate directory for the `.scm` file(s) of our channel so that `guix pull` doesn't load unrelated `.scm` files when someone pulls the channel—and in Guile, there are lots of them! So we'll start like this, keeping a top-level `guix.scm` symlink for the sake of `guix shell`:

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

To make it usable as part of a channel, we need to turn our `guix.scm` file into a *package module* (veja Seção "Package Modules" em *GNU Guix Reference Manual*): we do that by changing the `use-modules` form at the top to a `define-module` form. We also need to actually *export* a package variable, with `define-public`, while still returning the package value at the end of the file so we can still use `guix shell` and `guix build -f guix.scm`. The end result looks like this (not repeating things that haven't changed):

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

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

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

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

We need one last thing: a `.guix-channel` file (`https://guix.gnu.org/manual/devel/en/html_node/Package-Modules-in-a-Sub_002ddirectory.html`) so Guix knows where to look for package modules in our repository:

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

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

To recap, we now have these files:

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

And that's it: we have a channel! (We could do better and support *channel authentication* (`https://guix.gnu.org/manual/devel/en/html_node/Specifying-Channel-Authorizations.html`) so users know they're pulling genuine

code. We'll spare you the details here but it's worth considering!) Users can pull from this channel by adding it to `~/.config/guix/channels.scm` (`https://guix.gnu.org/manual/devel/en/html_node/Specifying-Additional-Channels.html`), along these lines:

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

After running `guix pull`, we can see the new package:

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

That's how, as a developer, you get your software delivered directly into the hands of users! No intermediaries, yet no loss of transparency and provenance tracking.

With that in place, it also becomes trivial for anyone to create Docker images, Deb/RPM packages, or a plain tarball with `guix pack` (veja Seção "Invoking guix pack" em *GNU Guix Reference Manual*):

```
# How about a Docker image of our Guile snapshot?
guix pack -f docker -S /bin=bin guile@3.0.99-git

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

7.4 Bonus: Package Variants

We now have an actual channel, but it contains only one package (veja Seção 7.3 [O Repositório como um Canal], Página 70). While we're at it, we can define *package variants*

(veja Seção "Defining Package Variants" em *GNU Guix Reference Manual*) in our `guile-package.scm` file, variants that we want to be able to test as Guile developers—similar to what we did above with transformation options. We can add them like so:

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

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

(define-public guile
  ...)

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

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

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


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

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

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

7.5 Level 3: Setting Up Continuous Integration

The channel we defined above (veja Seção 7.3 [O Repositório como um Canal], Página 70) becomes even more interesting once we set up *continuous integration* (`https://en.wikipedia.org/wiki/Continuous_integration`) (CI). There are several ways to do that.

You can use one of the mainstream continuous integration tools, such as GitLab-CI. To do that, you need to make sure you run jobs in a Docker image or virtual machine that has

Guix installed. If we were to do that in the case of Guile, we'd have a job that runs a shell command like this one:

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

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

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

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

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

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

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

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

7.6 Bonus: Build manifest

The Cuirass spec above is convenient: it builds every package in our channel, which includes a few variants (veja Seção 7.5 [Configurando Integração Contínua], Página 73). However, this might be insufficiently expressive in some cases: one might want specific cross-compilation jobs, transformations, Docker images, RPM/Deb packages, or even system tests.

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

```
;; This is ‘.guix/manifest.scm’.

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

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

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

                '("x86_64-linux" "i686-linux"
                  "aarch64-linux" "armhf-linux"
                  "powerpc64le-linux"))
           (map (lambda (guile)
                  (package->manifest-entry* guile "x86_64-linux"))
                (cons (package
                        (inherit (package-with-c-toolchain
                                  guile
                                  `(("clang-toolchain"
                                     ,(specification->package
                                       "clang-toolchain")))))
                        (name "guile-clang"))
                      (list guile-without-threads
                            guile-without-networking
                            guile-debug
                            guile-strict-typing))))))

(define cross-builds
```

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

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

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

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

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

7.7 Empacotando

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

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

These days, repositories are commonly peppered with dot files for various tools: `.envrc`, `.gitlab-ci.yml`, `.github/workflows`, `Dockerfile`, `.buildpacks`, `Aptfile`, `requirements.txt`, and whatnot. It may sound like we're proposing a bunch of *additional* files, but in fact those files are expressive enough to *supersede* most or all of those listed above.

With a couple of files, we get support for:

- development environments (`guix shell`);

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

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

8 Gerenciamento de ambientes

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

8.1 Ambiente Guix via direnv

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

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

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

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

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

    # Unset 'GUIX_PACKAGE_PATH'.
    export GUIX_PACKAGE_PATH=""

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

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

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

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

# Predefine configure flags.
configure()
{
    ./configure
}
export_function configure

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

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

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

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

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

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

9 Instalando Guix em um Cluster

Guix is appealing to scientists and HPC (high-performance computing) practitioners: it makes it easy to deploy potentially complex software stacks, and it lets you do so in a reproducible fashion—you can redeploy the exact same software on different machines and at different points in time.

In this chapter we look at how a cluster sysadmin can install Guix for system-wide use, such that it can be used on all the cluster nodes, and discuss the various tradeoffs[1].

> **Nota:** Here we assume that the cluster is running a GNU/Linux distro other than Guix System and that we are going to install Guix on top of it.

9.1 Configurando um nó principal

The recommended approach is to set up one *head node* running `guix-daemon` and exporting `/gnu/store` over NFS to compute nodes.

Remember that `guix-daemon` is responsible for spawning build processes and downloads on behalf of clients (veja Seção "Invoking guix-daemon" em *GNU Guix Reference Manual*), and more generally accessing `/gnu/store`, which contains all the package binaries built by all the users (veja Seção "The Store" em *GNU Guix Reference Manual*). "Client" here refers to all the Guix commands that users see, such as `guix install`. On a cluster, these commands may be running on the compute nodes and we'll want them to talk to the head node's `guix-daemon` instance.

To begin with, the head node can be installed following the usual binary installation instructions (veja Seção "Binary Installation" em *GNU Guix Reference Manual*). Thanks to the installation script, this should be quick. Once installation is complete, we need to make some adjustments.

Since we want `guix-daemon` to be reachable not just from the head node but also from the compute nodes, we need to arrange so that it listens for connections over TCP/IP. To do that, we'll edit the systemd startup file for `guix-daemon`, `/etc/systemd/system/guix-daemon.service`, and add a `--listen` argument to the `ExecStart` line so that it looks something like this:

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

For these changes to take effect, the service needs to be restarted:

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

> **Nota:** The `--listen=0.0.0.0` bit means that `guix-daemon` will process *all* incoming TCP connections on port 44146 (veja Seção "Invoking guix-daemon" em *GNU Guix Reference Manual*). This is usually fine in a cluster setup where the head node is reachable exclusively from the cluster's local area network—you don't want that to be exposed to the Internet!

[1] This chapter is adapted from a blog post published on the Guix-HPC web site in 2017 (`https://hpc.guix.info/blog/2017/11/installing-guix-on-a-cluster/`).

The next step is to define our NFS exports in `/etc/exports` (`https://linux.die.net/man/5/exports`) by adding something along these lines:

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

The `/gnu/store` directory can be exported read-only since only `guix-daemon` on the master node will ever modify it. `/var/guix` contains *user profiles* as managed by `guix package`; thus, to allow users to install packages with `guix package`, this must be read-write.

Users can create as many profiles as they like in addition to the default profile, `~/.guix-profile`. For instance, `guix package -p ~/dev/python-dev -i python` installs Python in a profile reachable from the `~/dev/python-dev` symlink. To make sure that this profile is protected from garbage collection—i.e., that Python will not be removed from `/gnu/store` while this profile exists—, *home directories should be mounted on the head node* as well so that `guix-daemon` knows about these non-standard profiles and avoids collecting software they refer to.

It may be a good idea to periodically remove unused bits from `/gnu/store` by running `guix gc` (veja Seção "Invoking guix gc" em *GNU Guix Reference Manual*). This can be done by adding a crontab entry on the head node:

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

... with something like this:

```
# Every day at 5AM, run the garbage collector to make sure
# at least 10 GB are free on /gnu/store.
0 5 * * 1  /usr/local/bin/guix gc -F10G
```

We're done with the head node! Let's look at compute nodes now.

9.2 Configurando nós de computação

First of all, we need compute nodes to mount those NFS directories that the head node exports. This can be done by adding the following lines to `/etc/fstab` (`https://linux.die.net/man/5/fstab`):

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

... where *head-node* is the name or IP address of your head node. From there on, assuming the mount points exist, you should be able to mount each of these on the compute nodes.

Next, we need to provide a default `guix` command that users can run when they first connect to the cluster (eventually they will invoke `guix pull`, which will provide them with their "own" `guix` command). Similar to what the binary installation script did on the head node, we'll store that in `/usr/local/bin`:

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

We then need to tell `guix` to talk to the daemon running on our master node, by adding these lines to `/etc/profile`:

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

To avoid warnings and make sure `guix` uses the right locale, we need to tell it to use locale data provided by Guix (veja Seção "Application Setup" em *GNU Guix Reference Manual*):

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

# Here we must use a valid locale name.  Try "ls $GUIX_LOCPATH/*"
# to see what names can be used.
LC_ALL=fr_FR.utf8
export LC_ALL
```

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

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

Last but not least, Guix provides command-line completion notably for Bash and zsh. In `/etc/bashrc`, consider adding this line:

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

Voilà!

You can check that everything's in place by logging in on a compute node and running:

```
guix install hello
```

The daemon on the head node should download pre-built binaries on your behalf and unpack them in `/gnu/store`, and `guix install` should create `~/.guix-profile` containing the `~/.guix-profile/bin/hello` command.

9.3 Network Access

Guix requires network access to download source code and pre-built binaries. The good news is that only the head node needs that since compute nodes simply delegate to it.

It is customary for cluster nodes to have access at best to a *white list* of hosts. Our head node needs at least `ci.guix.gnu.org` in this white list since this is where it gets pre-built binaries from by default, for all the packages that are in Guix proper.

Incidentally, `ci.guix.gnu.org` also serves as a *content-addressed mirror* of the source code of those packages. Consequently, it is sufficient to have *only* `ci.guix.gnu.org` in that white list.

Software packages maintained in a separate repository such as one of the various HPC channels (`https://hpc.guix.info/channels`) are of course unavailable from `ci.guix.gnu.org`. For these packages, you may want to extend the white list such that source and pre-built binaries (assuming this-party servers provide binaries for these packages) can be downloaded. As a last resort, users can always download source on their workstation and add it to the cluster's `/gnu/store`, like this:

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

The above command downloads `starpu-1.2.3.tar.gz` *and* sends it to the cluster's `guix-daemon` instance over SSH.

Air-gapped clusters require more work. At the moment, our suggestion would be to download all the necessary source code on a workstation running Guix. For instance, using the `--sources` option of `guix build` (veja Seção "Invoking guix build" em *GNU Guix Reference Manual*), the example below downloads all the source code the `openmpi` package depends on:

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

...

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

(In case you're wondering, that's more than 320 MiB of *compressed* source code.)

We can then make a big archive containing all of this (veja Seção "Invoking guix archive" em *GNU Guix Reference Manual*):

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

. . . and we can eventually transfer that archive to the cluster on removable storage and unpack it there:

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

This process has to be repeated every time new source code needs to be brought to the cluster.

As we write this, the research institutes involved in Guix-HPC do not have air-gapped clusters though. If you have experience with such setups, we would like to hear feedback and suggestions.

9.4 Uso do Disco

A common concern of sysadmins' is whether this is all going to eat a lot of disk space. If anything, if something is going to exhaust disk space, it's going to be scientific data sets rather than compiled software—that's our experience with almost ten years of Guix usage on HPC clusters. Nevertheless, it's worth taking a look at how Guix contributes to disk usage.

First, having several versions or variants of a given package in `/gnu/store` does not necessarily cost much, because `guix-daemon` implements deduplication of identical files, and package variants are likely to have a number of common files.

As mentioned above, we recommend having a cron job to run `guix gc` periodically, which removes *unused* software from `/gnu/store`. However, there's always a possibility that users will keep lots of software in their profiles, or lots of old generations of their profiles, which is "live" and cannot be deleted from the viewpoint of `guix gc`.

The solution to this is for users to regularly remove old generations of their profile. For instance, the following command removes generations that are more than two-month old:

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

Likewise, it's a good idea to invite users to regularly upgrade their profile, which can reduce the number of variants of a given piece of software stored in `/gnu/store`:

```
guix pull
guix upgrade
```

As a last resort, it is always possible for sysadmins to do some of this on behalf of their users. Nevertheless, one of the strengths of Guix is the freedom and control users get on their software environment, so we strongly recommend leaving users in control.

9.5 Security Considerations

On an HPC cluster, Guix is typically used to manage scientific software. Security-critical software such as the operating system kernel and system services such as `sshd` and the batch scheduler remain under control of sysadmins.

The Guix project has a good track record delivering security updates in a timely fashion (veja Seção "Security Updates" em *GNU Guix Reference Manual*). To get security updates, users have to run `guix pull && guix upgrade`.

Because Guix uniquely identifies software variants, it is easy to see if a vulnerable piece of software is in use. For instance, to check whether the glibc 2.25 variant without the mitigation patch against "Stack Clash (`https://www.qualys.com/2017/06/19/stack-clash/stack-clash.txt`)", one can check whether user profiles refer to it at all:

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

This will report whether profiles exist that refer to this specific glibc variant.

10 Agradecimentos

Guix is based on the Nix package manager (`https://nixos.org/nix/`), which was designed and implemented by Eelco Dolstra, with contributions from other people (see the `nix/AUTHORS` file in Guix.) Nix pioneered functional package management, and promoted unprecedented features, such as transactional package upgrades and rollbacks, per-user profiles, and referentially transparent build processes. Without this work, Guix would not exist.

As distribuições de software baseadas em Nix, Nixpkgs e NixOS, também foram uma inspiração para o Guix.

O GNU Guix em si é um trabalho coletivo com contribuições de várias pessoas. Veja o arquivo `AUTHORS` no Guix para obter mais informações sobre essas pessoas legais. O arquivo `THANKS` lista as pessoas que ajudaram a relatar erros, cuidar da infraestrutura, fornecer ilustrações e temas, fazer sugestões e muito mais – obrigado!

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

Apêndice A Licença de Documentação Livre GNU

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.

Índice de conceitos