# Podman in Theory and Practice

Wednesday, June 21st, 2023

## Introduction

This inaugural blogpost of Goose and Quill is intended as a storehouse of my own learning on Podman[1], especially centered in its operation on Guix System, a futuristic flavor of Linux with user freedom at its heart. It should be of use *to you*. By the time you've reached the end you will have set up Podman rootlessly, created a local Kubernetes cluster using Podman Desktop and deployed a simple web service to the cluster without any assumed knowledge of the internals of Kubernetes.

## Why Podman?

Podman is enormously *useful*. Rejected by many as a Docker clone, Podman possesses a toolset broader than its cousin.[2]

Podman is a portmanteau for "POD MANager". The unit of abstraction Podman operates on is the pod, a *collection* of software containers that work together to perform their function. In Docker, the fundamental unit is the container. Because Podman works at the level of the pod, just like its bigger cousin, Kubernetes, it's able to serve as an ad-hoc container orchestrator.

This level of shared abstraction enables powerful workflows that start at the level of a container specification, the Containerfile or Dockerfile, and end with a generated manifest that runs on any enterprise Kubernetes cluster. You could use *just Podman* and your favorite programming language and be well-equipped to deploy (almost) everywhere, anywhere.

1. Podman was developed by Daniel Walsh and his team at Red Hat in 2017. Walsh writes in Podman in Action, 2023 (https://livebook.manning.com/book/podman-in-action/front-matter/), of his aim to, "create a tool that ran the same containerized applications in the same manner but with more security and requiring fewer privileges."
2. Fun fact: Guix System, like Podman, began as a reaction. Guix is a portmanteau of "Guile" and "Nix" (https://web.archive.org/web/20230910205858if_/https://guix.gnu.org/en/blog/2022/10-years-of-stories-behind-guix/).

## Why rootless?

Running containers rootlessly is both practically powerful and secure. It's practically powerful because of the way containers achieve file and network separation from their host, using namespaces, a Linux kernel feature. When a namespace is created without elevated privileges, the user's user ID (UID) and group ID (GUID) are mapped inside the container. Any files shared across this boundary maintain consistent permissions. The result is an entire class of annoying container problems that just don't apply—Docker Compose veterans understand the pain of setting the right UID and GID when using bind mounts.

If you're sharing a workstation with others (more commonly, a server), users can run their own rootless containers isolated from others'. Podman has even introduced "Podmansh" in Podman 4.6, which extends this to its logical conclusion: every user logs in to their own rootless container.

Rootless containers are also secure from *container escapes* and file mount mishaps that allow a determined attacker to mount your entire drive and toast it[3].

## How-to set up Podman for rootless mode on Guix System

Because Docker, by default, runs root, it can do anything it wants, which makes its perceived ease of use very high. Rootless Podman is going to take some vim and vigor. After installing Podman with `guix install podman`, there's just two pre-requisites to bootstrap Podman and they're copy-paste jobs.

### Reserve user and group IDs for Podman to map into a namespace

Rootless containers use user namespaces[4].

> User namespaces isolate security-related identifiers and attributes, in particular, user IDs and group IDs, the root directory, keys, and capabilities. A process's user and group IDs can be different inside and outside a user namespace. In particular, a process can have a normal unprivileged user ID outside a user namespace while at the same time having a user ID of 0 inside the namespace; in other words, the process has full privileges for

3. https://www.devseccon.com/blog/whats-so-great-about-rootless-containers-secadvent-day-24
4. https://man7.org/linux/man-pages/man7/user_namespaces.7.html

operations inside the user namespace, but is unprivileged for operations outside the namespace.

We “trick” our containers into believing they have all the privileges of a rootful environment by assigning a range of user and group IDs our rootless container will map inside their boundary. In the code sample below, which you’ll need to copy into your own Guix System configuration (here `system.scm`), we map 65,536 subuids[5] to our user. Rootless containers will map, from host to container: 100000 to 1, 100001 to 2, and so on. 1000 to 0 is a special default mapping we don’t need to define.

Change the username string, “worldofgeese” to your own user.

system.scm

```
(define username "worldofgeese")

(operating-system
  (services
   (cons*
 ...
       (simple-service 'etc-subuid etc-service-type
                  (list `("subuid" ,(plain-file "subuid" (string-append "root:0:65536\n" username
                        ":100000:65536\n")))))
       (simple-service 'etc-subgid etc-service-type
                  (list `("subgid" ,(plain-file "subgid" (string-append "root:0:65536\n" username
                        ":100000:65536\n")))))
 ...
)))
```

## Set container image trust policy and activate your changes

Next, set your container image trust policy, which by default prevents the user from pulling from any and all remote registries.

system.scm

```
(simple-service 'etc-container-policy etc-service-type
              (list `("containers/policy.json", (plain-file "policy.json" "{\"default\": [{\"
                   type\": \"insecureAcceptAnything\"}]}"))))
```

One more nicety before we write, build and run our first image: setting a fast storage driver. Podman on Guix uses vfs by default and you should absolutely not subject yourself to it because it is dog slow. You can check for yourself if you’re using vfs with `podman info | grep graphDriverName`, which should return `graphDriverName:`

5. Subuids or subordinate UIDs authorize a user to delegate user IDs into child namespaces.

`vfs`. I won't go into virtual filesystems here.[6] `overlayfs` has been in the kernel since 5.11 so that's what we'll be using here. Below where you put your container image trust policy add the following:

```scheme
system.scm
(simple-service 'etc-storage-driver etc-service-type
                (list `("containers/storage.conf", (plain-file "storage.conf" "[storage]\ndriver = \"overlay\""))))
```

Activate your changes to `system.scm`. I keep mine in ` /.config/guix/system.scm` so I activate a new *Guix generation*[7] with my changes by invoking `sudo -E guix system reconfigure  /.config/guix/system.scm`.

If you've used Podman before, you'll need to run podman system reset after to enable your new storage driver. Check again with `podman info | grep graphDriverName`. It should now read `graphDriverName: overlay`.

## How to package "Hello, World" in Guile on Guix

That's all we need to run Podman rootlessly! Now we'll create a simple container using the Hello HTTP server example from the official website of of the Guile Scheme language[8].

Create a file, any file, that ends in `.scm` and inside paste this code:

```scheme
my-hello-http.scm
;;; Hello HTTP server
(use-modules (web server))

(define (my-handler request request-body)
  (values '((content-type . (text/plain)))
        "Hello World!"))

(run-server my-handler)
```

If Guile isn't already installed, install it with `guix install guile`. In the example above where we've saved the code to `my-hello-http.scm`, you can run it directly with

6. Docker has a good intro to storage drivers (https://web.archive.org/web/20230927081131if_/https://docs.docker.com/storage/storagedriver/select-storage-driver/) as well as a page on each if you're curious.
7. A Guix generation is a collection of symbolic links that points to a specific Guix configuration in time. This gives Guix its power to roll back non-destructively to previous sytem versions without fuss. Try that with your Windows system!
8. https://www.gnu.org/software/guile/

`guile my-hello-http.scm`, open a web browser and visit http://localhost:8080[9] where you'll see, printed, "Hello, World!".

This code isn't yet portable: it's still a script that requires a user to know to install Guile first, download the code and run it in a file ending in `.scm`. And it won't run in Podman, which expects a container, not a script. In the next section, we'll write a self-contained Guix package definition that generates a container image using Guix. This is a break from what you may be used to, which is using Podman or Docker to build an image from a Containerfile or Dockerfile.

## Create a container image with Guix and run it with Podman

To package our "Hello, World!" example, we're going to need to write more Guile. This time, in Guix's own extension language to Guile.[10] It's Guile-ception!

The following code *package definition* puts the HTTP server code we ran earlier into a function `generate-server-code`, then uses Guix's special "G-expressions"[11] in the build step. Finally, we generate a manifest that tells Guix the contents of our package. Replace the code in `my-hello-http.scm` with the following:

my-hello-http.scm

```
(define-module (my-hello-http)
  #:use-module (guix packages)
  #:use-module (guix build-system trivial)
  #:use-module (gnu packages guile)
  #:use-module (guix licenses)
  #:use-module (guix gexp))

(define (generate-server-code guile-path)
  (string-append "#!" guile-path " -s
  !#
  ;;; Hello HTTP server
  (use-modules (web server))

  (define (my-handler request request-body)
    (values '((content-type . (text/plain)))
          \"Hello World!\"))

  (run-server my-handler)"))
```

9. http://localhost:8080/
10. Teaching Guix's Guile syntax is out of scope of this post. For an intro, visit the Guix website for a three-part tutorial (https://web.archive.org/web/20230601125937if_/https://guix.gnu.org/en/blog/2023/dissecting-guix-part-3-g-expressions/).
11. G-expressions use special notation (# , #$) to evaluate Guix package expressions inside the build environment.

```

(define server-code
 #~(generate-server-code #$guile-3.0))

(define-public my-hello-http
 (package
  (name "my-hello-http")
  (version "0.1")
  (source #f)
  (build-system trivial-build-system)
  (arguments
   (list #:builder
       #~(begin
           (let* ((bin-dir (mkdir-p (string-append #$output "/bin")))
                  (script-file (string-append bin-dir "/my-hello-http")))
             (with-output-to-file script-file
               (lambda () (display ,server-code)))
             (chmod script-file #o755)))))
  (native-inputs (list guile-3.0))

  (synopsis "My Hello HTTP server")
  (description "This package contains a simple HTTP server.")
  (home-page "https://www.gnu.org/software/guile/")
  (license gpl3+)))

(specifications->manifest (list "my-hello-http"))
```

Now we need to tell Guix where to find our package. We do that by adding the package path to the `GUIX_PACKAGE_PATH` environment variable. On your command line, enter `export GUIX_PACKAGE_PATH=$GUIX_PACKAGE_PATH: /path/to/package`. As an example, if `my-hello-http.scm` file is in the `/home/worldofgeese/testing` folder, I'd enter `export GUIX_PACKAGE_PATH=$GUIX_PACKAGE_PATH: /testing`.

To test that Guix can find your package, run `guix show my-hello-http`, which should print:

```
name: my-hello-http
version: 0.1
outputs:
+ out: everything
systems: x86_64-linux i686-linux
dependencies: guile@3.0.9
location: my-hello-http.scm:10:2
homepage: http://example.com
license: GPL 3+
synopsis: My Hello HTTP server
description: This package contains a simple HTTP server.
```

Now, from `/home/$USER` enter `guix pack -f docker -m testing/my-hello-http.scm`.

Voilà! A container image is produced in the Guix store[12]. We can load this image directly

12. https://guix.gnu.org/manual/en/html_node/The-Store.html

into Podman like so:

```
> podman load < /gnu/store/235f92alcfr7hfjbs8a0snnnrxz3ill1-my-hello-http-docker-pack.tar.gz
WARN[0000] "/" is not a shared mount, this could cause issues or missing mounts with rootless
     containers
Getting image source signatures
Copying blob 304960ad3eb5 done
Copying config b1a55ba007 done
Writing manifest to image destination
Storing signatures
Loaded image: localhost/my-hello-http:latest
```

Then run it with `podman run localhost/my-hello-http`.

Now if you visit http://localhost:8080[13] you'll see, again, "Hello, World!".

13. http://localhost:8080/