Unholy

AstraLuma

Dec 09, 2023

CONTENTS

Cooperates with Docker Compose, injects an nvim-based container into it, and connects neovide to it.

You can call it what you'd like, but I'm going to call this union Unholy.

CHAPTER

ONE

WHAT?

Unholy is a tool to create and manage Docker-based development environments–it's a dev container implementation.

Unholy performs all operations over the Docker API; no side-band channels or open ports are used. Both local and remote daemons are supported.

- Uses Neovim and Neovide as the editing UI
- Connects the development environment to its Docker daemon, so container and compose operations work
- Your SSH agent is forwarded, so SSH operations (including Git-based ones) work
- Your Git config and some SSH data is copied (notably not your private keys; you should use an ssh agent for that)
- Multiple daemons are supported through Docker Contexts

CHAPTER

TWO

CONTENTS

2.1 Installation Guidelines

This is not going to give specific instructions, because I have not tested Unholy on many machines yet.

2.1.1 Prerequisites

First and formost, you need a Docker daemon somewhere.

Additionally, you need the following software installed locally:

- Git
- socat
- Docker CLI
- Neovide
- Python (3.10 or later)

Note that while you do not need a local Docker daemon running, your Docker client must already be connected to the daemon and functional. (Unholy supports Docker Contexts and it's suggested to use them for managing daemon connections.)

2.1.2 Installing Unholy

Unholy can be installed through Python. pipx is suggested.

With pipx, installation is:

```
$ pipx install unholy
```

2.1.3 Initial Configuration

Besides the aforementioned Docker client credentials, no initial configuration of Unholy is required.

2.2 Quickstart

Ok, so you have Unholy, what do you do with it?

2.2.1 Make a Project

An Unholy-native project will contain both an *Unholyfile* and a Compose file.

If we follow the Try Docker Compose guide, we'll end up with a compose file like this:

Listing 1: compose.yaml

```
services:
  web:
    build: .
    ports:
      - "8000:5000"
  redis:
    image: "redis:alpine"
```

Additionally, the absolute minimum Unholyfile is this:

Listing 2: Unholyfile

```

```

You can actually skip creating an Unholyfile, and it'll be equivalent.

2.2.2 Spawn your Environment

With the basic files out of the way, you can ask unholy to create an environment:

```
$ unholy new --name demo git@github.com/YOU/YOURPROJECT.git
```

A lot of text will go by as Unholy creates the workspace, clones the repo, starts compose services, and creates & configures your development environment.

This will create a vanilla Debian environment with pretty minimal utilities. (Really, just enough for Unholy to work and a few utilities for humans.)

2.2.3 Access Your Environment

Ok, so you've got this shiny development environment sitting on a computer somewhere. It doesn't do you any good if you can't access it.

In order to open Neovide and a shell, use these:

```
$ unholy neovide demo
$ unholy shell demo
root@demo:/workspace#
```

These two commands do related things:

- `unholy neovide` opens Neovide and connects it to neovim running in the development environment
- `unholy shell` drops you into bash inside the development environment

2.2.4 Customizing You Environment

Ok, so one of the cool things about containers is recreatable artifacts and environments, and Unholy is no exception.

In summary, the Unholyfile is a script with some TOML headmatter. The framework would look something like:

Listing 3: Unholyfile

```
---
---
#!/bin/sh
set -e
```

Pick an Image

Since this is a Python project, let's start with a Python environment.

Listing 4: Unholyfile

```
---
[dev]
image = "python:3"
---
#!/bin/sh
set -e
```

Note: The container image must be Debian-based.

Add Some Dependencies

Most projects have some dependencies that need to happen: test runners, git tooling, etc. Let's install them.

Listing 5: Unholyfile

```
---
[dev]
image = "python:3"
---
#!/bin/sh
set -e
pip install -r requirements.txt
pip install pytest
```

Note: Since this is a dedicated, single-purpose environment, you do not need to use a Python virtual environment or similar.

2.2.5 Recreate the Environment

Since we've changed the Unholyfile (in particular, we've changed the base image), we need to recreate the environment:

```
$ unholy remake demo
```

This will recreate the development environment without touching your workspace files. Note that any open shell or Neovim sessions will be uncerimoniously closed, so make sure that your work is saved.

2.3 How-To

Some quick recipes.

2.3.1 How To Share a Workspace with Compose

When using Docker Compose, it's very common to share your project with the service containers through a bind mount, such as:

Listing 6: compose.yaml

```yaml
services:
  web:
    build: .
    ports:
      - "8000:5000"
    volumes:
      - .:/code
  redis:
    image: "redis:alpine"
```

However, this tries to bind a directory from the docker daemon host, not inside the development environment. So how does one share code with services?

Explicitly Name Your Workspace

While this isn't strictly necessary, since you're going to refer to your workspace volume by name, I suggest explicitly naming it.

Listing 7: Unholyfile

```
---
[dev]
# ...
volume = "workspace"
# ...
---
# ...
```

Warning: If anyone has an Unholy workspace, and you change this value from `"workspace"`, everyone must recreate their entire Unholy project. `unholy remake` will not save you.

Tell Compose

Ok, the actually important part:

1. Tell Compose about your workspace volume
2. Mount it into your services

Like so:

Listing 8: compose.yaml

```yaml
services:
  web:
    build: .
    ports:
      - "8000:5000"
    volumes:
      - workspace:/code
  redis:
    image: "redis:alpine"

volumes:
  workspace:
```

Warning: `docker compose down --volumes` will now attempt to delete your workspace. It'll fail (probably), but it's going to try.

Recreate Your Services

Ask Compose to recreate your services:

```
$ docker compose up -d
```

All Done!

Now things like code auto-reload should work as expected.

2.3.2 How To Use Unholy With a Project Without an Unholyfile

For reasons, you will probably need to use Unholy with a project that doesn't have an Unholy file.

Create Your Unholy Project

As per normal, call `unholy new` with your repo:

```
$ unholy new git@git.host/you/repo.git
```

In the output, you might be able to spot a line:

> Unholyfile not found in project. Continuing with defaults

Edit Your Unholyfile

As discussed in *Configuration*, some data is kept locally on your workstation. You can make your Unholyfile adjustments in there instead in the workspace.

This file is typically `~/.config/unholy/PROJECT.Unholyfile`.

You can put all settings and scripting in there just like an Unholyfile in the workspace.

Recreate Your Environment

As mentioned in the quickstart, you'll probably want to do this every time you edit your Unholyfile.

```
$ unholy remake PROJECT
```

2.4 Guides

These are some long-form discussions about aspects of Unholy.

2.4.1 Some Details

Ok, so what does Unholy actually do?

The Resources

Throughout Unholy, we refer to several resources:

- **Workspace**: A Docker volume that's used to keep your git repo and other project materials, mounted as `/workspace`.
- **Development Environment**: A Docker container where all commands are run.
- **Unholyfiles**, **local config**, etc: The complete collection of data that Unholy uses to manage your project. (See *Configuration*)
- **Unholy Project**: The entire collection of the above.

For example, `unholy remake` deletes and recreates the Development Environment but doesn't touch your Workspace.

Development Environment Creation

Unholy doesn't use Dockerfiles or create/cache images, or anything like that, for a few reasons:

- It's expected that most of the time that you're recreating your development environment, it's because you changed an Unholyfile
- Config comes from many places, including a few user-specific ones, so caching would have to be user-specific, too
- The Dockerfile syntax is not particularly helpful in this use case

So Development Environment creation is like so:

1. Pull the base image (every time, no use building from a stale base)

2. Copy some data from the user, like git config and ssh known hosts 2. Run each Unholyfile's script

Forwarding and Piping

All connections in and out of the development environment are through Docker exec and stdio forwarding. That's all.

Neovide is pointed to something like `docker exec devenv nvim`.

SSH agent forwarding is two copies of socat chained together with Docker.

If you find this horrifying, first I'm sorry I inflicted this knowledge on you, and second the name of the tool is Unholy.

Bootstrap

Occasionally, when the development environment is unavailable, Unholy will spawn a bootstrap container and use that for operations. It should be deleted automatically when Unholy is done with it.

2.4.2 The Unholyfile

The only configuration used by Unholy is The Unholyfile, so let's discuss it.

Format

Listing 9: Unholyfile

```
---
# This is headmatter. It is TOML.
key = "value"

[section]
key = "value"
---
#!/usr/bin/sh
# This is the script
```

An Unholyfile is a script with TOML headmatter.

Both the script and the headmatter are optional.

Delimiters are lines with nothing but dashes, and at least 3 of them.

If there is headmatter, it must be preceeded by a delimiter. That delimiter must by on the first line.

If there is both headmatter and a script, they must be separated by a delimiter. If there is headmatter but no script, the trailing delimiter is optional.

An empty file is valid. A file with no delimiters is interpreted as all script with no headmatter.

A script may start with a shbang (`#!`). If it does, it must be on the line immediately following the delimiter (if present).

Schema

As documentation and information, here is the base Unholyfile (see *Configuration*) included in Unholy:

Note: TOML does not have a null/nil/None/etc, it only has missing keys. Commented out lines are values that default to null (aka missing).

```
---
#: git url
# repository = ""

#: Docker context to use
# context = ""


[dev]
#: The base image for the dev container
image = "docker.io/library/debian:latest"

#: The name of the volume containing the source
```

(continues on next page)

(continued from previous page)

```
volume = "workspace"


[compose]
#: Compose file to use
file = "compose.yaml"

#: The name of the compose project
# project = ""
---
#!/bin/sh
set -e
apt-get update

# Install some basics
apt-get install -y sudo git curl gpg socat man less

# Add Docker's official GPG key:
apt-get install -y ca-certificates curl gnupg rsync
install -m 0755 -d /etc/apt/keyrings
curl -fsSL https://download.docker.com/linux/debian/gpg | gpg --dearmor -o /etc/apt/
↪keyrings/docker.gpg
chmod a+r /etc/apt/keyrings/docker.gpg

# Add the repository to Apt sources:
echo \
  "deb [arch="$(dpkg --print-architecture)" signed-by=/etc/apt/keyrings/docker.gpg]␣
↪https://download.docker.com/linux/debian \
  "$(. /etc/os-release && echo "$VERSION_CODENAME")" stable" > /etc/apt/sources.list.d/
↪docker.list
apt-get update

apt-get install -y docker-ce-cli docker-compose-plugin


# Download neovim tarball
curl -L https://github.com/neovim/neovim/releases/download/stable/nvim-linux64.tar.gz \
| tar -xz -C /tmp
rsync -a /tmp/nvim-linux64/* /usr/
rm -rf /tmp/nvim-linux64
```

2.4.3 Configuration

When Unholy creates or manipulates projects, it uses *Unholyfiles* from several sources (in order from back to front in the stack):

1. `core.Unholyfile` from Unholy itself
2. Some dynamically generated values
3. `~/.config/unholy/Unholyfile` (or the XDG user configuration directory)
4. `~/.config/unholy/<project>.Unholyfile`, substituting the project name (again, XDG)

5. The `Unholyfile` from the project (either pulled from the workspace or from git)

Settings from lower in the list take prescedence over those higher.

`<project>.Uholyfile` is automatically created and updated by `unholy new`. If Unholy needs to update it, it will do so with minimal disturbance of changes by humans. That is, you can safely edit any Unholyfile and know that Unholy won't wontonly discard comments, spacing, etc.

Each Unholyfile is also used for environment configuration. They are applied in the order above–so `core.Unholyfile` goes first, and the repo's goes last.

2.5 Reference

2.5.1 unholy

An amalgamation of docker compose and neovim

```
unholy [OPTIONS] COMMAND [ARGS]...
```

Commands

ls

List projects in the local config.

neovide

Open neovim/neovide inside the devenv

new

Create a new project from a git repo

remake

Recreate the devenv.

shell

Open a shell inside the devenv

2.5.2 unholy new

unholy new

Create a new project from a git repo

```
unholy new [OPTIONS] REPOSITORY
```

Options

--name <name>
Project name (default: guess from repository URL)

-o, --remote, --origin <remote>
Name of the remote (default: origin)

-b, --branch <branch>
Name of the branch to check out (default: remote's HEAD)

-c, --context <context>
Name of the docker context to use (default: unset)

Arguments

REPOSITORY
Required argument

2.5.3 unholy remake

unholy remake

Recreate the devenv.

```
unholy remake [OPTIONS] NAME
```

Arguments

NAME
Required argument

2.5.4 unholy neovide

unholy neovide

Open neovim/neovide inside the devenv

```
unholy neovide [OPTIONS] NAME
```

Arguments

NAME
Required argument

2.5.5 unholy shell

unholy shell

Open a shell inside the devenv

```
unholy shell [OPTIONS] NAME
```

Arguments

NAME

Required argument

2.5.6 unholy ls

unholy ls

List projects in the local config.

```
unholy ls [OPTIONS]
```

CHAPTER

THREE

INDICES AND TABLES

- genindex

INDEX

Symbols

N

R

U