Swift for Linux Documentation

Ryan Gonzalez

May 22, 2018

Contents:

Swift for Linux is a project aiming at making the experience of using the Swift programming language from Linux easier.

CHAPTER 1

What's wrong?

A lot! Apple only provides Swift binaries that run well on Ubuntu, and users of other distros will have a fun time trying to get them to work. It's hard to share Swift applications you've built, and the majority of the ecosystem, being Mac-and-iOS-focused, tends to create libraries that don't work under Linux. This guide is trying to be a central point for people trying to get a great Swift experience on their Linux systems.

If you have anything you'd like to add, or if you notice a problem in this guide, feel free to submit an issue.

Also, if you need help, try stopping by the unofficial Swift Discord.

1.1 Installing Swift

Unfortunately, the official Swift binaries only work on Ubuntu, and they're standalone binaries independent of any package manager. Here are steps on getting it to work on various distros/package tools.

1.1.1 Flatpak

The easiest way to get started using Swift is via the Flatpak repo. Make sure you've followed the official Flatpak setup instructions.

First, make sure you have the Freedesktop SDK installed:

```
$ flatpak --user install flathub org.freedesktop.Sdk
```

Add the Flatpak remote:

```
$ flatpak --user remote-add swift https://swift-flatpak.refi64.com/swift.flatpakrepo
# Or, if you prefer to add it manually:
curl -sSL https://gpg.refi64.com/swift-flatpak | flatpak --user remote-add --gpg-
↪import=- swift http://swift-flatpak.refi64.com
```

Then, install the SDK extension and live SDK:

```
$ flatpak --user install swift org.freedesktop.Sdk.Extension.swift4 org.freedesktop.
↪Sdk.Extension.swift4.live
```

In order to run the Swift compiler via the Flatpak, you'll need to use this command:

```
$ flatpak run -d org.freedesktop.Sdk.Extension.swift4.live swift ...
```

To shorten this, define an alias:

```
$ alias swiftpak='flatpak run -d org.freedesktop.Sdk.Extension.swift4.live swift'
```

Now, you can just use *swiftpak*:

```
$ swiftpak
```

1.1.2 Ubuntu

You can use the official Swift binaries. If you want to be able to upgrade Swift via apt, then try out the Vapor PPA:

```
$ eval "$(curl -sL https://apt.vapor.sh)"
```

or:

```
curl -L https://repo.vapor.codes/apt/keyring.gpg | sudo apt-key add -
echo "deb https://repo.vapor.codes/apt $(lsb_release -sc) main" | sudo tee /etc/apt/
↪sources.list.d/vapor.list
sudo apt update
sudo apt install swift
```

1.1.3 Fedora

An RPM is available here, though it requires a build from source. The Ubuntu binaries should more-or-less work if you patch them as mentioned in *Using the Ubuntu Binaries*.

1.1.4 Arch Linux

You can install Swift on Arch Linux using either the AUR swift package or the swift-bin package. (The former builds from source, whereas the latter uses patched versions of the Ubuntu binaries.)

1.1.5 Others

For other distros, see *Extra Credit: Swift on Other Distros*.

If you've gotten Swift working on your favorite distros, feel free to create an issue to mention yours!

1.2 Navigating the Ecosystem

Now that Swift is installed, there's still a major problem: the ecosystem. Many Swift libraries are designed with macOS and/or iOS in mind, meaning they rely (either intentionally or accidentally) on Cocoa APIs and Darwin C library bindings.

1.2.1 Pure Swift

The best indicator that a library will work on Linux is if it mentions *pure Swift* somewhere in the name. This term basically means that it likely won't depend on any Cococa APIs, at minimum.

In addition, the presence of a `Package.swift` file, used by the Swift package manager, means that it doesn't require something like CocoaPods to build.

Note that when I say *pure Swift*, I'm not referring to this GitHub organization. Although they *create* pure Swift libraries, that's about it.

1.2.2 Libraries

Here are some awesome libraries that work on Linux:

Web

- Vapor: A popular backend web framework. The developers also maintain the PPA mentioned in *Installing Swift*.
- Kitura: A web framework by IBM. Note that, although they have Linux support, the documentation is very macOS-centric. It tends to assume you *develop* on macOS and *deploy* to Linux.

CLI

- Console: A fantastic CLI library that's part of the Vapor project.
- Progress.swift: Progress bars.

GUI

- SwiftGtk: GTK+ bindings. These seem to be rather complete and are auto-generated.
- Qlift: Qt bindings. Not sure how complete these are.
- Cacao: A UIKit implementation that works on macOS *and* Linux. No commits since late 2017, and there are open bugs related to building it.

Parsing

- Kanna: HTML parsing.
- SwiftSoup: HTML parsing.
- Yams: YAML parsing.

Miscellaneous

- Regex: Regular expressions.

Submitting

Feel free to submit more libraries!

1.3 Distributing Swift Applications

You've created your Swift application. Now, how do you get it to your users?

The easiest way is by using the Swift *Flatpak*. If you haven't installed it yet, now's the time to do so.

1.3.1 Creating a Flatpak

First off, make sure you already know the basics of creating Flatpaks; see the official developer guide for more information.

Let's use a simple *Hello, world!* project as our example. Create a `Package.swift`:

```
// swift-tools-version:4.0

import PackageDescription

let package = Package(
    name: "example",
    dependencies: [
    ],
    targets: [
        .target(
            name: "example",
            dependencies: []),
    ]
)
```

and `Sources/example/main.swift`:

```
print("Hello, world!")
```

Now, let's assume the full application ID will be *org.mysite.Hello* (Flatpak uses reverse domain name notation for application IDs). Create `org.mysite.Hello.json` containing the following:

```
{
  "app-id": "org.mysite.Hello",
  "runtime": "org.freedesktop.Platform",
  "runtime-version": "1.6",
  "sdk": "org.freedesktop.Sdk",
  "sdk-extensions": [
    "org.freedesktop.Sdk.Extension.swift4"
  ],
  "command": "/app/bin/example",
  "modules": [
    {
      "name": "sdk",
      "buildsystem": "simple",
      "sources": [
        {
          "type": "git",
          "path": "https://github.com/myuser/myrepo.git",
          "tag": "HEAD"
        },
        {
          "type": "script",
```

(continues on next page)

(continued from previous page)

```
          "commands": [
            ". /usr/lib/sdk/swift4/enable.sh",
            "swift build -c release",
            "install -Dm 755 .build/release/example /app/bin/example",
            "/usr/lib/sdk/swift4/set-runtime.sh /app/lib /app/bin example"
          ],
          "dest-filename": "build-example.sh"
        }
      ],
      "build-commands": [
        "./build-example.sh"
      ]
    }
  ]
}
```

Here's a breakdown of the interesting parts of this file:

- `sdk-extensions`: **This is where the Swift SDK extension is used.** The SDK extension "extends" the previously chosen SDK with the Swift compiler and libraries.
- `sources`: This is where the sources are chosen. The first is just the Git repository of our application, but the second is far more interesting and will be explained below.
- `build-commands` calls into `./build-example.sh` to build our code.

`build-example.sh` does the following:

- Sources `enable.sh` to enable use of the Swift SDK extension.
- Builds the application.
- Installs it to `/app/bin/example` via the `install` command. Note that Flatpak requires your application to be installed to the `/app` prefix.
- Calls a script called `set-runtime.sh`. This script will copy the Swift runtime libraries to the application directory, and it will set the dynamic linker of your application binaries to the library directory. The arguments are as follows:
  - The first is the directory to store the Swift runtime libraries and patched dynamic linker (see *Version Warnings* for information on why that is necessary).
  - The second is the directory where your application binaries are stored.
  - Any other arguments passed are assumed to be paths to binaries, relative to the second argument (the application directory). These binaries will all have their dynamic linker set to the patched one that doesn't emit version warnings.

1.4 Extra Credit: Swift on Other Distros

Both Swift's sources and official binaries tend to assume a Ubuntu-like system. This is a guide on how to make Swift run well on other distros, too.

1.4.1 Using the Ubuntu Binaries

Shared Libraries

Swift's binaries depend on the following shared libraries that may not be easily available:

- ICU 55. Many distros come with newer versions. If that is the case for you, you'll have to either find a package providing ICU 55 (AUR has icu55, which is what I use in swift-bin), use precompiled binaries (ICU provides some for RHEL6), or build ICU 55 yourself.
- libcurl.so.3. **The 3 here is important!** When libcurl underwent an API changes, many distros updated the version from libcurl 3 to libcurl 4, even though the ABI remained unchanged (source). As Swift was built on Ubuntu distros that still run libcurl 3, running it on other distros may result is issues regarding libcurl.so.3 being missing. *In most cases*, you can just run `sed -i 's/libcurl.so.3/libcurl.so.4'` on all the binaries in the Swift distribution. (You may still encounter version information warnings, described below in *ref*:Version Warnings:.)
- A similar issue exists with Swift requiring libedit.so.2, which in most other distros is (properly) named libedit.so.0. Again, a `sed` call should do the trick.
- Swift requires ncurses5, but many distros now carry version 6. However, they also usually will carry a package provided ncurses5 binaries; for instance, Fedora carries `ncurses-compat-libs`, and Arch has `ncurses5-compat-libs`.

Version Warnings

After making any necessary changes as detailed above, you may also encounter these dreaded warnings:

```
usr/bin/swift: /usr/lib/libtinfo.so.5: no version information available (required by␣
↪usr/bin/swift)
```

The reason for these is that some Linux distros encode *version information* into their shared libraries (Ubuntu and Fedora do this), whereas others do not (rolling release distros such as Arch fall into this category). As the Swift binaries where built on Ubuntu, they expect your shared libraries to have version information, and on a distro where they don't, you'll get the above warnings.

Unfortunately, this is compounded by two facts:

- These warnings are emitted by your *dynamic linker*, which is the program that runs ELF binaries. They are toggled by a verbosity option which, oddly enough, is hardcoded into the glibc source code. There is no way to turn them off.
- Swift tends to assume that *any* output to standard error by helper tools means that the tool failed. This makes it virtually impossible to build Swift code, since the compiler will see that the version warnings were printed to standard error and abort the compilation process.

The only solution is to created a patched copy of the dynamic linker that does *not* emit these warnings. I created qldv to accomplish this task. (I also described the process a bit here, though qldv now uses a different algorithm that is compatible with more distros.)

Of course, there's still the task of making the Swift binaries use the patched linker. Luckily, patchelf handles this job quite nicely.

The TL;DR of this whole thing is that **you'll need to use qldv, combined with patchelf, in order to silence the version information warnings**. Here is an example:

```
# Create a patched copy of the dynamic linker, and save it to /usr/lib/swift/ld.so.
# (qldv -find is to locate the dynamic linker; it's usually somewhere like
/lib/ld-linux-x86-64.so.2.)
$ qldv `qldv -find` /usr/lib/swift/ld.so
# Patch all the Swift binaries to use this new linker.
# (Except for liblldb-intel-mpxtable.so, which doesn't need to be patched and
# doesn't work with patchelf.)
$ find usr/bin -type f -not -name liblldb-intel-mpxtable.so -exec patchelf --set-
↪interpreter /usr/lib/swift/ld.so {} \;
```

However, there's still one more issue: any binaries you compile will still emit version warnings. The solution to this is to trick Swift into running patchelf on any binaries it builds. The easiest way I've found to do this is to replace clang++ (which Swift uses to link binaries) with a shell script that calls the *real* clang++ and then runs patchelf on the result.

I've toyed with various methods to do this (using bubblewrap) was an interesting one, but the best option I've found is to do this:

- Install Swift to a prefix *other* than /usr. This is because Swift first tries to locate clang++ in the same directory as the other Swift binaries, and only *then* does it refer to your PATH. From here on out, I'll assume you picked /usr/lib/swift, and that you had saved the patched ld.so to the same directory.
- Symlink all the Swift binaries from /usr/lib/swift/bin to /usr/bin.
- Create a shell script /usr/lib/swift/bin/clang++ containing the following:

  ```
  #!/bin/bash
  /usr/bin/clang++ "$@" && patchelf --set-interpreter /usr/lib/swift/ld.so "${@: -1}
  ↪"
  ```

  This just calls the *real* clang++ and then calls patchelf on the result (the last argument passed by Swift is always the output file).

Now, Swift will call the fake, shell script clang++, which calls the real clang++ and patches the output binaries.

Include Directories

You may still exhibit some issues regarding missing headers. This is simply because Swift looks for system headers in /usr/include/x86_64-linux-gnu, but on some distros, they're all in /usr/include. To fix this, again refer to trusty sed:

```
$ sed -i 's|x86_64-linux-gnu/||' usr/lib/swift/linux/x86_64/glibc.modulemap
$ sed -i 's|x86_64-linux-gnu/||' usr/lib/swift_static/linux/static-stdlib-args.lnk
```

1.4.2 Building from Source

Building from source is far easier, surprisingly enough, but it also makes upgrades take far longer. However, you'll still probably have to patch the sources a bit.

One thing you may have to change are references of /usr/bin/python to /usr/bin/python2, since many distros now set /usr/bin/python to point to Python 3, which isn't backwards-compatible with Python 2. You can see this being done in the AUR swift package.

Outside of that change, other ones are far more distro-specific. For a great starting point, check out the changes needed to build Swift on Fedora.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search