Rethinking your shebang

brianstorti.github.io/rethinking-your-shebang

April 22, 2015

Are you using `#!/bin/{bash,zsh,sh}` in your shebang? Most scripts that I have to deal with are, and it sucks. Let me show you why using `#!/usr/bin/env {bash,zsh,sh}` is better, most of the time. If you are already using `#!/usr/bin/env`, learn why you shouldn't just use it blindly.

Why is `/bin/bash` bad?

First, it assumes that `bash` (or whatever you are using) is installed in that specific location, in every system it's going to run. Although this is the case in most systems, there are exceptions. In OpenBSD, for example, `bash` is an optional package and is located at `/usr/local/bin/bash`.

But even if I'm using a system where `bash` is installed at `/bin/bash`, I might be using a different `bash` version, installed in a different location.

```
$ bash --version
version 4.3.33

$ /bin/bash --version
version 3.2.51
```

What's happening here is that I used `homebrew` to install a newer version of `bash`, and all these `homebrew` packages are installed at `/usr/local/bin/`. If this script uses a `bash` 4 feature, it will fail just because it's pointing directly to `bash` 3.

Using the `$PATH` instead

So, how does `/usr/bin/env` solve this problem? Well, instead of hard-coding a location, you are telling your script to look for `bash` in the system's `$PATH`. That means it doesn't matter where `bash` is, or how many different versions you have installed, as long as it's in your `$PATH`, it will be found. This will solve both problems that I showed before.

```
$ export PATH=/bin/:$PATH
$ /usr/bin/env bash --version
version 3.2.51

$ export PATH=/usr/local/bin:$PATH
$ /usr/bin/env bash --version
version 4.3.33
```

Security concerns

As we are now using the `$PATH` to find what we want to execute, there are some (minor, I'd say) security concerns that need to be taken into consideration when we are dealing with a multi-user environment.
Let's say I create a malicious script at `/home/brianstorti/evil/bash` and somehow trick you to add this directory to your path (e.g. `export PATH=/home/brianstorti/evil/:$PATH`). Now every time that `env` looks for `bash` in your `$PATH`, it will actually find my evil script.

And a last portability concern

Also, ironically, there is a portability issue that you have to keep in mind when using `/usr/bin/env`.
In some systems the shebang line processing will accept just one interpreter and one argument. This means that if, for instance, you want to run a `Ruby` script in warning mode, using `#!/usr/bin/env ruby -w` might fail.
`env` is the interpreter and `ruby` is the argument, so it tries to parse `-w` as a file name.

Summarizing

- In most cases, using `/usr/bin/env bash` will be better than `/bin/bash`;
- If you are running in a multi-user environment and security is a big concern, forget about `/usr/bin/env` (or anything that uses the `$PATH`, actually);
- If you need an extra argument to your interpreter and you care about portability, `/usr/bin/env` may also give you some headaches.

Interested in learning Kubernetes?

I just published a new book called Kubernetes in Practice, you can use the discount code **blog** to get 10% off.

Get fresh articles in your inbox

If you liked this article, you might want to subscribe. If you don't like what you get, unsubscribe with one click.