7 minute read | 2021-04-02

The Case For Bash

`esac`

Friends, coders, I come not to praise bash, but to contextualize it.

Bash is oft-maligned: you hear it sneered at in sentences like "our continuous integration pipeline is just a pile of bash scripts." It is commonly regarded as unintelligible and thus unmaintainable. The pile of bash scripts are the toxic byproduct of an infrastructure team understaffed, pressed for time, unwilling or unable to do it "the right way" in a blessed programming language, such as Ruby, Python, or, uh, ... YAML?

> *some problems with bash:*
>
> *1. It's a weird & counterintuitive language*
>
> *2. you probably don't need to write it that often, so you don't practice*
>
> *3. the times you DO need to use it, it's often because something important (like a build) broke and it needs to be fixed RIGHT NOW*

-- Julia Evans, @b0rk, via tweet

Julia has a a great zine that will get you comfortable with Bash. I'm not going to try to do that in this post! Instead, I'm going to try to convince you why it's *worth* learning Bash. To do that, I'm going to talk about the problem space that Bash thrives in, and why other languages are a bad fit for that same space.

So, about problem spaces and languages. You might be familiar with the idea that any language with sufficient flow-control primitives can express the same algorithm that any other language of that class can express. In other words, given access to the same system APIs, the same approach to solving a problem could be expressed in Java, Ruby, or Bash. By comparison to art, if you had to render an image of the inky blackness of outer space, you could accomplish it with pen and ink or with a flat brush and paint. One approach will save you a lot of time (and wrist problems!)

This is to state by example: certain tools are better adapted to certain problems. You, as a programmer, are likely to run into a wide range of problems requiring a corsucating variety of problem-solving approaches in your career. The first, best tool that we can bring to bear on the problems before us is **language**, and thankfully we have a variety from which to choose.

Languages lend themselves to solving a subset of the problems you're likely to encounter. When selecting a language to fill a particular niche in your toolset, there are a number of considerations: performance, popularity, ubiquity, familiarity, capability, amongst others. Languages will overlap niches: C, C++, Java, Go, and Rust are good choices for problems where performance is a primary concern. Python, JavaScript, and Ruby are excellent multitools where performance isn't a problem. All are popular languages with active ecosystems. There is no universal correct answer, even amongst languages with overlapping capabilities, because every language has different values that suit different people.

“

> *As a programmer, I think it behooves us to think about not just the values that we hold, but the values of the people that use our software hold and, as a programmer, you should use tools that align with your values.*

-- "How Rust Views Tradeoffs", Steve Klabnik, @steveklabnik

I haven't talked about Bash yet.

So, what does Bash value? One might be tempted to say "ubiquity". Bash is available nearly everywhere these days; but ubiquity is a quality that is conferred from outside the language. Preserving ubiquity *can* be a value -- see JavaScript and backwards compatibility, or C and the PDP-11 -- but it's difficult to positively *attain* ubiquity.

Instead, Bash values a *very* particular fitness for purpose: the management and coordination of programs. This includes spawning processes, connecting processes via pipelines, and parallelizing processes. The purpose of Bash is *running other programs*. Consider the following:

```
tail -fq /var/log/nginx/access.log | awk '{print $7}' | sort | unic
```

"Follow the ngnix access log; print the 7th word (the path); sort those paths, count the number of unique instances, and then sort the results by occurrence count."

```
const { spawn } = require('child_process')

const tail = spawn('tail', ['-fq', '/var/log/nginx/access.log'])
const awk = spawn('awk', ['{print $7}'], {stdio: [tail.stdout, 'inh
const sort = spawn('sort', [], {stdio: [awk.stdout, 'inherit', 'inh
// ...
```

This is a contrived example, but there's a lot more *going on* in the Node example than there is in the Bash one. The Python and Ruby versions would be similar. The core data primitive of Bash is a *running process*. It has distinct syntax for dealing with the input and output of those processes, for capturing that output as a variable, and dealing with process exit conditions. The boilerplate you'd type in another programming language is the *default* in Bash.

It values running programs over assigning variables, over branching, over everything. If you don't believe me, consider that the common mode for `if` statements in Bash is `if [ <test> ]; then <stmt>; fi`. Now go run `less /bin/[`. That's right: `[` is a **program** being run by the `if` builtin.

Bash's choices don't make a lot of sense coming from other languages: `0` is "true" or "success", strings inside variables often need to be re-quoted on use, the loop and conditional logic is off-the-wall; but it makes sense in *context* of its values (and, well, history.)

It's true that a lot of what I've said about Bash is equally true of other shell scripting languages: Fish, Zsh, Powershell, and Nushell all value similar things. It's worth learning these languages! However, if you're dealing with Linux servers on a daily basis, it's definitely worth learning Bash. Even if you're not, it's worth learning just to understand what these other shell scripting languages are reacting to. Bash's `for` loops might be weird, but if you can familiarize yourself with them, you can use that knowledge everywhere from your personal laptop to a server running on AWS and back down to a shell running on your local router.

You (can) use Bash every time you open a terminal session. This knowledge compounds over time: you don't have to learn the entire language to be more efficient. You can layer in new concepts every few weeks and before you know it, it'll be muscle memory. Any time you're

presented with the need to write a one-off program to accomplish a small task, take the opporunity to puzzle out how to write it as a shell script. The difficulty of learning Bash is less in learning the language itself, and more in identifying what programs you should call to accomplish your goal. This is like any other language: finding the right API and the right way to call it forms the bulk of day-to-day programming effort. So, give it a whirl! Remember that there are alternatives to tricky APIs! If you do, you'll have a valuable tool that fills a niche most other popular programming languages struggle to address.