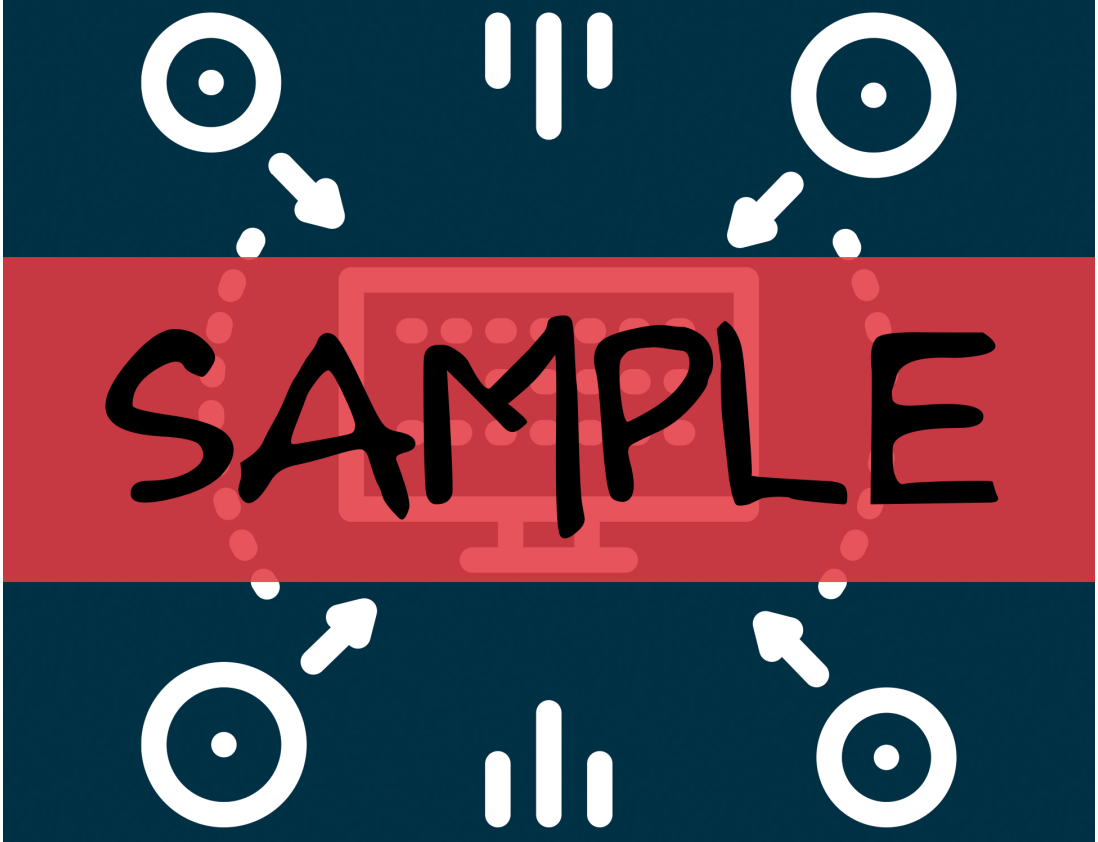
Sustainable Dev Environments

with

Docker and Bash

Build a Stable and Maintainable Place to Code



DAVID BRYANT COPELAND

Sustainable Dev Environments with Docker and Bash

Build a Stable and Maintainable Place to Code

David Bryant Copeland

Contents

Contents

Preface

Over the last four or five years, I've written several tech books, worked at a startup, and pursued many personal projects. Docker has allowed me to focus on the work and not on getting software re-installed for the 80th time just because Apple changed something.

I've refined how to use Docker for development, applying the same general principles to Ruby on Rails, NodeJS, CSS, and writing technical books. The consistency it brings, along with the value created by carefully writing the Bash scripts needed to glue it all together, has allowed me to focus, multi-task, and get things done.

A great example is this book itself. The toolchain[1] I use to write books requires a lot of tools, including some weird ones, like Graphviz, LaTeX, pandoc, Ruby, NodeJS, and Mermaid. Getting them installed on my Mac is difficult, so I put them all into Docker. I took a three month break from working on this book and got a new computer in that time. The first time I started up the book's development environment on my new computer—which only had Docker installed—everything just worked.

Another part of it is that I took the time to write decent Bash scripts to automate aspects of Docker. Docker was clearly not designed for an everyday developer to be typing its commands, but this is what Bash is for!

Docker and Bash aren't the most beloved or easy-to-use tools, and I will be pointing out their various issues in usability and intuitiveness as we go. But they are ubiquitous and, despite some rough edges, they both work, and work reliably.

Acknowledgments

I'd like to thank the Docker team, since they made a useful tool and created a great ecosystem. I will be throwing a lot of shade their way in this book, but it just means that Docker is worth criticizing. I'm glad that Docker is something I can complain about, instead of something no one has ever heard of.

I'd also like to thank whoever has worked on Bash over the years, particular whoever has continually made it backward compatible. When I find a Bash answer on Stack Overflow, it always works. ChatGPT is even right more than half the time!

I'd also like to thank the many tech reviewers who gave me great feedback on early versions of this book: Jason Garber, Reid Gillette, Sam Livingston-Gray, and Bradley Schaefer.

[1] https://www.naildrivin5.com/blog/2023/02/03/toolchain-for-building-programming-books.html

1

Your Dev Environment Sucks

MacOS Bakersfield is our best version of macOS, yet.
We've replaced the long-deprecated version of Ruby with nothing. You'll have to build Ruby yourselves, and you'll get to use our exciting new compiler flags that we haven't bothered to document. We've also managed to make sure that Homebrew now requires your iTunes password with every install. Only Apple has the courage to make this update to your computer without your consent.
Oh, and one more thing: Your wiki is out of date.

—Craig Federighi

They say people don't quit their jobs—they quit their managers. I think developers often quit their dev environments. The hodgepodge of tools, libraries, scripts, and documentation required to actually build software is pretty important. So why does it seem to be broken all the time? In my almost 30 years writing professional software, I have not experienced a truly awesome dev environment. I bet you haven't, either. We're going to change that.

Dev environments suck because of a perfect storm of factors. Once you set one up, you rarely set it up again, so any instructions on how to do it are instantly obsolete. Hardware and software across even a small dev team is wildly inconsistent.

Operating system vendors and security teams force updates that break transitive dependencies. Any automation has to work in a degenerate environment where no other software has been installed, which makes it hard to write and change.

On top of all this, usually no one owns the dev environment. Each team typically has at most two experts: the developer that set it up initially and whoever was the most recent new hire.[1]

It doesn't have to be this way. We aren't woodworkers whose working environment is a physical space occupied by heavy machinery that can't be moved or replaced. A dev environment is a software system. We know this.

[1] And let's be honest, everyone else on the team is in a race to see if they can get promoted to management or hired at a competitor before IT refreshes their computer and they have to rebuild everything.

What You'll Learn

This book is going to show a way forward using Docker and Bash. That's right... Bash. But this isn't a blog post where I tell you what incantations to paste. You're going to learn how Docker works. You'll set up a dev environment by first running painfully verbose counter-intuitive Docker commands. This is how you'll learn what's going on. Then, you'll make it all better with well-written and nicely-crafted Bash scripts.

With just Docker and Bash you can automate just about any dev environment you can imagine[2].

Here is how this is going to go: We need a bit of theory to ground us in the problem we're solving and why the solution this book presents is the best, all things being equal. We'll learn what a dev environment is, and why the best implementation is automation of a virtualized environment. Our theory will end with an overview of Docker's terminology and behavior. You'll want to bookmark that chapter.

We'll then use Docker and Docker Compose to make a generalized dev environment for an example app. In addition to seeing Docker in action, we'll learn the right way to choose base images for use and the right way to install software in the images you will build. It's marginally more difficult than using Stack Overflow, but will produce far superior-and sustainable—results.

After that, we'll wrap up all those Docker commands with some general-purpose Bash scripts. But these will be no ordinary Bash scripts. They will provide *help on the command line*. They will be easy to use. They will even use some features of good software design so that someone other than the author can make sense of them.

What You Need

To use the tools we'll cover, you need a computer capable of running Docker. That means any recent Linux box, Apple computer, or a Windows-based PC that has Windows Subsystem for Linux 2 (WSL1). All of these platforms will contain Bash.

For completeness, I'm running Docker Desktop 4.20.1 and GNU Bash 5.1.4. Anything close to these versions will be fine.

[2]Except iPhone apps. For those you are stuck with Xcode, the App Store, and a 27% markup on an untestable payment processing experience you must maintain on behalf of customers you cannot contact. I'm very sorry if this is your life, and I hope your app is a hit.

A Note on Windows

If you are using a Windows PC, you really should use WSL2, which is free and easy to install. Like…so easy I couldn't believe it. Once you do that, and then install Docker, you *must* follow the post-install instructions[a] on Docker's website.

Your WSL setup will have you running Linux as a normal, non-root user, and by default, that type of user can't use Docker. The instructions at the link work - I did this using a Windows PC having not touched Windows at all in probably 10 years.

For the most part, everything works on Windows the same was as it does on Mac or Linux. On the few occasions where something is off, I've noted it, but generally what WSL2 buys you is compatibility with the rest of the computing landscape.

[a] https://docs.docker.com/engine/install/linux-postinstall/

You also don't need to be the most experienced developer. Some experience writing code in any language will be sufficient. The Docker stuff isn't really code. If you've never done Bash before, don't worry, that will all be explained, including some esoteric command line stuff that is notoriously hard to search for.

Typographical Conventions

As a book that must work on lots of devices *and* paper, the code listings have to fit in a certain space. I've done my best to make sure the code is accurate and doesn't flow off the page.

There are some command-line invocations, and they look like so:

```
> ls -xF /
bin@   boot/  dev/  etc/  home/  lib@  media/  mnt/  opt/  p. . .
sbin@  srv/   sys/  tmp/  usr/   var/
```

Sometimes, the output is lengthy and irrelevant, so in those cases, it's omitted, and would look like so:

```
> ls /etc
«lots of output»
```

More frequently, a command-line invocation will be long, and not fit on the page. Fortunately, the UNIX command-line environment allows you to stretch a command over several lines by ending a line in a backslash. For example:

```
> ls -l ~/projects
```

Can be written as:

```
> ls \
  -l \
  ~/projects
```

For code, when creating a file, the text should state clearly to create a file with a given name in a given location. For example, I might ask you to create the file `amazing.js` in the current directory:

```
/* amazing.js */

document.addEventListener("DOMContentLoaded", () => {
  console.log("This is the only JS in this book")
})
```

For editing code, I'll try to show enough context to know where to make edits, along with arrows that indicate the changed code:

```
/* amazing.js */

  document.addEventListener("DOMContentLoaded", () => {
→   console.log("I lied, this is also JS")
→   console.log("And sorry in advance, there is gonna be YAML")
  })
```

Sometimes lines of output are too long for the page but also not relevant. Those will be truncated and I assure you that there is no important information being lost.

One last thing that I cannot seem to fix is that copy and paste out of the PDF is often somewhat strange. I would recommend you type things out. It's better for learning anyway. If you don't want to type things out, the sample code is available from the book's website at `https://devbox.computer/sample_code.zip`.

If you have any problems getting things to work, try the sample code from the website first.

OK, let's get into it. What exactly *is* a dev environment?

2

The Four Types of Dev Environments

Pai Mei brought Kiddo to the standing desk. On the desk was a brand new computer. Kiddo tried to install the latest version of NodeJS following Pai Mei's written instructions. When it failed, she turned to her master. Pai Mei only frowned.
Kiddo, unsure how to proceed, said "Master, how am I to install software onto my computer?"
*Pai Mei, enraged, smashed her knuckles with the end of his staff. "*Your *computer?"*
Cradling her bruised hand, Kiddo cried: "Master, is it not mine?!"
Pai Mei, still enraged, again smashed Kiddo's hand with his staff. After a single stroke of his long, white beard, he simply grumbled and glared at Kiddo.
Kiddo's eyes dropped as she looked back at the computer. Her gnarled fingers could barely enter the search terms into the operating system vendor's support site. In that moment, she was enlightened.

—Kill Bill, Vol 3

While there are actually an *infinite* number of dev environments, any given setup can be categorized along two axes, to give us four broad categories. This chapter explains why one of these is superior to the others.

First, a dev environment is categorized based on the instructions for setting it up: documentation on one side and automation on the other. I'd bet most environments you've used were heavy on documentation and light on automation.

The second axis relates to how the environment is run. Is it native—running everything directly on the developer's workstation—or virtual—running in the cloud or a virtual environment?

What we're building in this book is the best of the four options: automated and virtualized. Let's talk through why automation is better than documentation and virtualization is better than native.

2.1 The Best Documentation is an Executable Script

Documentation is cheap to produce, especially if it isn't maintained or well-written. It's often better than nothing, so most teams start their journey to a sustainable dev environment with a Markdown file or a wiki that outlines what you need to do to get set up. This does not scale. At all.

It's extremely hard to write good documentation. It's harder when what you are documenting is complex, which happens when your software installation system has to accommodate several package managers, operating systems, hardware architectures, and Pat, who insists on building everything from source.

Of course, even if you could achieve this, how is this documentation maintained? For dev environments, the new hires are usually charged with updating it when they find it doesn't work. Over time, the steps become so convoluted that not even the most conscientious person can follow them.

Automation solves this. Automation shows exactly what has to happen because it makes it happen. Automation either works or doesn't. Even though automation feels expensive to produce, it saves time the more it's used.

Automation has two further advantages. First, developers already possess the skills to produce it, whereas they may or may not be good at writing. Second, automation can be tested. A script that sets up a working dev environment can be used to setup an environment for continuous integration, thus ensuring that the team is aware of issues quickly, and can fix them just as quickly.

2.2 Our Computers are Increasingly Not Under Our Control

If you happen to be a Ruby developer who uses a Mac, you've no-doubt experienced the yearly problem when macOS releases an update and you can no longer install Ruby. Macs have long-since stopped shipping with a reasonable version of Ruby, and you certainly can't install gems (Ruby's form of third-party libraries) without breaking something.

This is not something unique to Apple. Every OS vendor, in their quest for stability, will take great strides to prevent changes to what is considered the "system software". If some script depends on a particular version of Perl, and you change that version, you could break the operating system.

Of course, it's not just the operating system vendor. Many companies have IT and security teams tasked with preventing security incidents. A critical tool in doing so is to force operating system and software updates to the employees. These teams aren't always capable or incentivized to work with developers to ensure such updates won't impact their ability to work. Even if they did, at the end of the day, security updates are going to be more important.

The reality is that we don't really own all the software on our computers, and that we can't easily understand how the various libraries and tools that come with it are affected by the libraries and tools we need to do our work.

Virtualization solves this. As long as your computer can run the virtualization software, you can run a virtual machine configured exactly how you like, and it

won't change out from under you. *And* your entire team can use that exact same version, even though said team might be using a myriad of different computers and operating systems.

Virtualization *does* come with potentially worse performance than running natively, but this is a worthwhile trade-off (and the performance gap is always shrinking). I would be willing to bet that the time spent waiting for slightly slower tests is far outweighed by the time saved not wrestling with some arcane compiler flags every time something changes in your OS.

2.3 Automation and Virtualization Lead to Sustainability

An automated dev environment, based on virtualized operating systems, provides a solid foundation for building just about any app. The automation is never out of date, and the operating system can be kept stable.

Eschewing virtualization requires automating the set up of a developer laptop. While this is better than a documentation-based approach, it's still highly complex. The automation must account for all operating systems and hardware.

When automating developer workstation setup, the team must either maintain that system themselves or rely on a third party. Whatever preconceived notions you may have about Docker, I can assure you that it's simpler to have Docker install software than to write a script that must work across many different OSes and hardware profiles.

As for third party solutions, they have to get installed themselves *and* the team must understand how they work to debug or enhance them. This turns out to be more difficult than learning a commonly-used tool like Docker. We'll talk about this in "Tech Companies Should Not Own Your Dev Environment" on page **??**.

On the other side, using a virtualized environment with documented instructions can be helpful, but you still fall victim to the trappings of documentation. Your docs might be simpler, since they can address the virtualized environment only, but they will still fall out of date.

We're going to use Docker for virtualization, and a combination of Docker and Bash for automation. The reasons have to do with a hidden, third axis: how easy is it to understand the abstractions on which your dev environment is built?

2.4 You Must Always Understand One Level Below the Current Abstraction

The best abstractions are borne from repeated applications of a technology for a well-defined use-case. Writing assembly language gets tiresome, so C was invented. Even though assembly can do far more than C, for most common use-cases, C is much faster and easier to use. It's a great abstraction.

If you learn C and not assembly, you will eventually hit a limit. You won't know exactly what problem C was created to solve and, eventually, there will be a problem that your knowledge of C alone cannot solve. You will need to learn a bit of assembly.

Your dev environment is the same way. Whatever mechanism you use to manage it, it is ultimately an abstraction on top of other technologies that are being orchestrated to manage your environment. When something goes wrong—either due to a bug or an unforeseen use case—you'll need to pop the hood and see what's under there.

Thus, you need to understand—or be able to get an understanding of—whatever your dev environment is built on, as shown in "Understanding Abstractions" below.

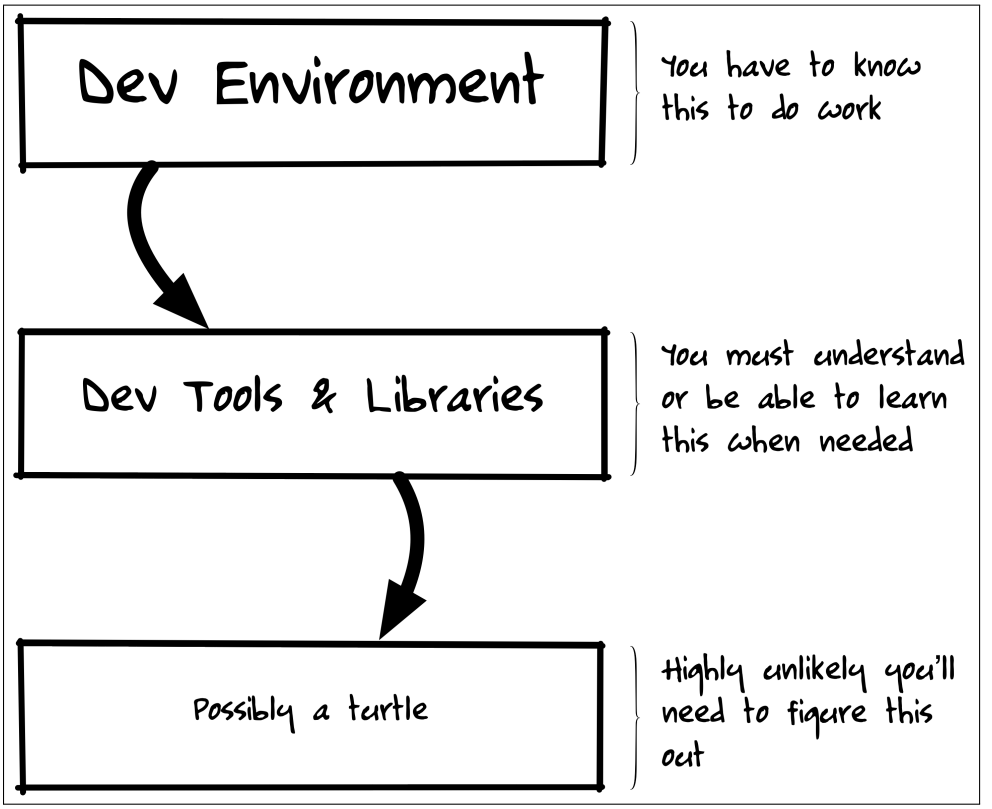


Figure 2.1: Understanding Abstractions

What *this* means is that your dev environment should use technologies that you either do, or can, understand. Applying this to your team, this also implies that there is more value in using commonly-understood, battle-tested technologies than in using something that might tick off more features but is more esoteric or less likely to continue to exist past its next round of funding.

Docker—despite being VC-funded itself—is prolific. A *lot* of people understand it, and it's not going anywhere. There are even competing products that can build Docker images and run Docker containers.

And Bash... well... Bash will outlive us all. The only better investment in your career than learning Bash is learning SQL and if there were a way to automate all this with SQL, I'd *definitely* be considering it.

To that end, we'll now start our journey to learn this stuff. It's not going to take too long. We'll start with Docker, which, despite some warts and a few design flaws, is the best tool for the job of virtualizing our dev environment.

A

The Docker Lexicon for Regular People

"Forget about it" is like if you agree with someone, you know, like Perl was great before version 5, forget about it. But then, if you disagree, like functional programming is better than imperative? Forget about it! You know? But then, it's also like if something's the greatest thing in the world, like Ruby on Rails 3.0, forget about it. But it's also like saying go to hell, too. Like, you know, like "Hey Patty, your cohort in that A/B test tanked our conversion rate" and Patty says "Forget about it!". Sometimes, it just means forget about it.

—Donald "Joe" Brasco "Pistone", former Fog Creek Staff Engineer

Table A.1: Docker Nouns

Docker	English
Container	Virtual Computer
Container Registry	Hosted Images
Dockerfile	Image Build Script
Host	Your Computer
Image	Disk Image for a Virtual Computer
Port	Network Port on a Virtual Computer
Repository	Remote location for different versions of the same image
Volume	Disk

Table A.2: Docker Verbs

Docker	English
build (an image)	Create an image from a Dockerfile
create (a container)	Order a computer from Amazon and put a hard drive in it when it shows up
expose (a port)	Allow another container to access a port in this container
mount (a volume)	Make a disk available

Docker	English
pull (an image)	Fetch an image from a remote registry and store it for later use
publish (a port)	Map a container's port to your computer's port on `localhost`
push (an image)	Send an image to a remote registry
start (a container)	Turn on a computer
stop (a container)	Shut down a computer

B

Bash and Command-Line Quick Reference

Ready are you? What know you of ready? Through eight rounds of investment have I trained UNIX hackers. My own counsel I will keep on who is to be trained. A UNIX hacker must have the deepest commitment, the most serious mind. React? Hmph. Mobile Apps? Heh. A UNIX hacker craves not these things.

—OH at the bar of Pied Piper, Palace Hotel, San Francisco

B.1 Concepts

Environment, or "UNIX Environment" A set of keys and values that a process can access when it executes. The UNIX Environment is a common way to pass complex information from one process to the next. Each key/value pair is called an *Environment Variable*. Both the key and the value are treated as strings.

Exit Code or Exit Status a value, usually from 0 to 127, that represents the outcome of a command. 0 means the command succeeded. Any other value means it failed. Some commands document their exit codes so you can inspect the value to find out more details about why it failed. Used by Bash when running commands in a pipeline, using `&&`, or `||`.

File Descriptor An integer that UNIX interprets as an input or output stream.

Input Stream An abstraction that allows reading input. It is typically read in a loop until the stream is closed, which allows any amount of data to be read, since it doesn't have to fit into memory at the same time.

Output Stream An abstraction that allows sending output somewhere. It is typically used until whoever is writing to it closes it. This allows a potentially unlimited amount of data to be written.

Path A set of directories that will be searched for when you attempt to run a command. The path is a string where each directory is delimited by a colon. The directories are searched in order. You can inspect the path by examining the environment variable `PATH`, e.g. `echo $PATH`.

Process Something running on the computer, like your app, `ls`, `bash`, or a web server. A process is the lowest-level abstraction that you will deal with when using Bash or the command-line.

Shell An interpreter to interact with the operating system, typically initiated by a user logging into the system and executing commands.

Standard Error An output stream where command-line apps are expected to write error messages. This is file descriptor #2

Standard Input An input stream a command-line app can use to read input from the user or another command. This descriptor is #0.

Standard Output An output stream where command-line apps are expected to send whatever output they exist to output. Should not have error messages. This is file descriptor #1.

B.2 Symbol Soup

`/dev/null` A special file that discards everything sent to it. Used to hide output from scripts.

`>` e.g. `ls > file.txt` Redirect the *standard output* of a command into a file, overwriting that file.

`>>` e.g. `ls >> file.txt` Append the *standard output* of a command to the end of a file, creating it if needed.

`2>` e.g. `grep gem Gemfile 2> /dev/null` Redirect the *standard error* of a command into a file.

`2>&1` e.g. `grep gem Gemfile > /dev/null 2>&1` Combine the *standard error* of a command into its *standard output* and send them to wherever the > sent them (to `/dev/null` in this example)

`|` e.g. `echo ls | bash` Send the *standard output* of a command into the *standard input* of another command, thus creating a pipeline. Note that to cause the entire pipeline to fail if *any* command fails, you must `set -o pipefail` before running the pipelined commands.

`||` e.g. as in `bin/setup || echo 'failed'` If the command on the left of the `||` fails, run the command on the right. Otherwise, don't do that.

`&&` e.g. `bin/setup && echo 'OK'` If the command on the left of the `&&` fails, don't run the command on the right. Otherwise, run that command.

`;` e.g. `bin/setup ; echo 'No idea'` Separates commands, running each in order, regardless of the exit code of any command.

`&` e.g. `bin/run &` Runs a command in the background. When you do this, the shell will print a number inside brackets like so: `[3]`. You can then kill the process by running `kill %«number»` (e.g. `kill %3` in this example), or you can bring it back to the foreground via `fg %«number»` (e.g. `fg %3` in this example).

#! e.g. `#!/bin/bash` at the start of script. On the first line of a script, tells the operating system what program to use to interpret the remainder of the script.

**** As seen in many code examples here, this allows you to split long lines onto multiple lines in the terminal.

${VAR_NAME} Access the value of a variable named `VAR_NAME` which could be in the UNIX environment or set as part of the script, *or* set on the command line.

${0} The name of the script currently being executed.

${1}, ${2}, ... The arguments given to the script or function, with `${1}` being the first argument, `${2}` being the second and so on.

'${VARNAME}' or '$VARNAME' When used in a Bash script, the use of single quotes prevents `VARNAME` from being expanded into its value. The literal value inside the single quotes is used.

"${VARNAME}" or "$VARNAME" When used in a Bash script, the use of double quotes ensures that when `VARNAME`'s value is expanded, it is quoted and thus will not be misinterpreted as multiple arguments.

"${@}" A special form that expands to the Bash command's arguments list, with *each argument* properly quoted so that any command to which this is passed will interpret it the same way the Bash command did.