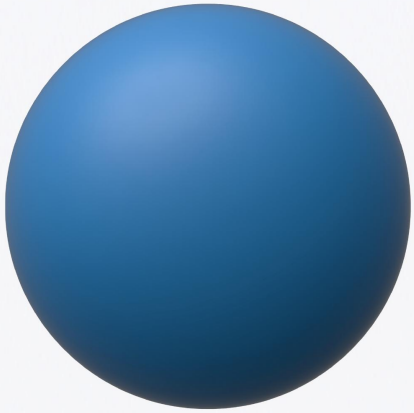
PragmaticBookshelf

simplicity

sustainable, humane, and effective software development



dave thomas

edited by
Susannah Davidson

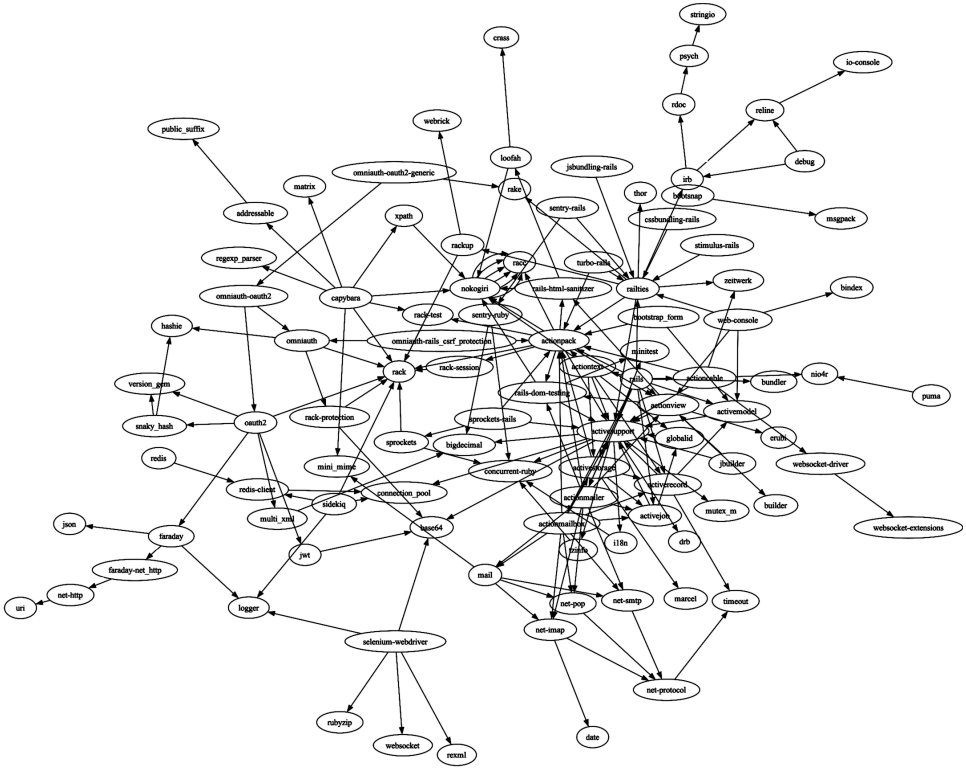
This extract shows the online version of this title, and may contain features (such as hyperlinks and colors) that are not available in the print version.

For more information, or to purchase a paperback or ebook copy, please visit https://www.pragprog.com.

Cut Back on Unhealthy Dependencies

It's late 2024. I just created a new Phoenix project using `mix phx.new my_project`. It had 43 dependencies, which came in at 15MB.

A new Rails project installs 83 gems, and these in turn load hundreds more. Here's a dependency graph of a basic Rails app.



`npx create-react-app my-app` downloaded 874 modules (about 350MB of code) for a basic React app. I had time to make a cup of tea.

I'm not claiming that React is bloated compared to the others. I'm saying that they *all* are carrying around a lot of baggage. And these are the numbers for baseline, empty projects. As you start adding code, you'll be adding dependencies of your own (and those dependencies will have their own dependencies, and so on).

I hate dependencies—every last one of them.

Every dependency I use gives control of a part of my future to a tree of third parties, people I probably don't know, can't control, and in reality can't even trust. There are probably tens of thousands of people who've contributed to the 874 modules that React uses out of the box. It only takes one of them to break my app.

Idea 1 **Every dependency gives away control of part of your app**

In March 2016, a prolific developer of JavaScript libraries suddenly removed all of them from the NPM repository. Among these modules was one named `left-pad` which, not surprisingly, added padding characters to the start of a string until it reached a given length.

The `left-pad` library was used by many thousands of other libraries, and these in turn by even more libraries. These second-order libraries included mainstays of the JavaScript world such as Babel and React.

One developer's decision rendered a significant proportion of JavaScript apps unbuildable.

In March 2024, a global warning (CVE-2024-3094) was issued after malicious code was found embedded in the XZ compression library. This was particularly concerning because the library was used by the OpenSSH daemon code, potentially compromising what was assumed to be a secure data transport.

When you add a dependency to your code, you're inviting a bunch of developers into your project, giving them access to your runtime. And these developers don't have to be malicious to cause problems: they can change APIs, deprecate features, or accidentally add security vulnerabilities.

My Personal Dependency Hell

In 2023 I came back to the Pragmatic Bookshelf after a seven-year break only to discover that the software that was the backbone of our business had effectively been unmaintained. It was still running Rails 3 (Rails is currently at version 8).

No problem, I thought, I'll migrate it up to the latest Rails, one release at a time.

Except Rails 3 uses a library that used a library that… was totally different on modern machines. I couldn't even build the Rails 3 gems. I tried just

updating blindly, only to discover that Rails had so many breaking changes between all these releases that I couldn't even get the app to start.

In the end, the only thing I could do was create a Docker container running an eight-year–old version of Linux that still had the old libraries. After a frustrating week, I finally had the most complex (and fragile) development environment I've ever used. I'm still managing the fallout as we migrate away.

Simplifying Dependencies

Remember the left-pad fiasco? Here's the entire code from the library:

```
module.exports = leftpad;
function leftpad (str, len, ch) {
  str = String(str);
  var i = -1;
  if (!ch && ch !== 0) ch = ' ';
  len = len - str.length;
  while (++i < len) {
    str = ch + str;
  }
  return str;
}
```

It's 11 lines of fairly ugly code; the line `str = ch + str` gives me garbage-collection nightmares.

Somewhere out there are developers who thought "I need to left pad this string," and who then took the time to search for a library, add it to their build, and work out how to import and use it. It probably took them three times as long as it would have done to write it themselves.

Today, `padstart` is built into JavaScript. Even if it weren't, a decent implementation could be just 3 lines of code:

```
function leftpad (str, len, ch=" ") {
    str = String(str);
    len = len - str.length;
    return (len > 0) ? ch.repeat(len) + str : str;
}
```

Idea 2 **Eleven lines of code should not be a dependency**

I know: this is an extreme case. But it's still useful to think about why folks chose to find a library for something so trivial. They probably were under time constraints, although in this case finding the library probably took longer than writing the code.

They may also have fallen into the *someone-else's-problem* trap. Delegating code to the author of a library means that they're not responsible for any problems.

Obviously I need to add dependencies to my project: I'm not going to write an SSH library from scratch; that would be ridiculous. But there's a trade-off to be made: adding a dependency today may solve a problem today. At the same time, it adds complexity for future-me in terms of risk and maintenance.

Isolating Dependencies

In the past, I've been bitten by dependencies that I use extensively through an app changing their APIs, forcing me to update code across multiple source files.

Now, if possible, whenever I use a dependency widely in an app, I wrap it in a simple function, and use that function in the rest of the code. That way, if the API changes, I might be able to get away with updating just the wrapper function.

The Dependency Decision Chain

Before adding a dependency to a project, I consider:

Do I need the functionality?

Younger Dave often added a new dependency after I read about some cool new feature online. I did it because it looked interesting, not because my project absolutely needed it.

Now I try to have the discipline not to experiment like that; if my project will work without it, I won't add it. Remember that "less is more." (Of course, there's nothing to stop me from playing around with it in my spare time….)

Is it easier just to code it?

Increasingly, I find myself not using dependencies if all I need from them is a single function. If I can write it myself, I do. If the library is open source and the license permits, I might copy the function (with attribution) into my own code, where I can examine it and also update it to fit the project's needs better.

Am I buying a jungle?

Joe Armstrong said about dependencies "You wanted a banana but what you got was a gorilla holding the banana and the entire jungle." Sometimes the functionality I want comes with a lot of baggage. Maybe I just need to escape

HTML strings but the library I found contains an entire HTML parsing and creation system, along with a dozen external dependencies for things like parsing, indentation, and so on.

All that extra cruft is a liability. I may not be using it, but it still contributes to the code that I might be fighting with in the future.

Is this vital to my project?

If so, I'd better make sure I have a local copy of it. There's nothing quite like that sinking feeling of getting a 404 when you try to fetch something your project needs.

Am I locked to a particular version?

I always use a dependency manager for the libraries I use. I also use one for the tools. (Right now I use asdf to manage tools for me; I know, I know, the cool kids use Nix…). Both of these allow me to specify the versions of everything I use when creating a project.

That's fine for a while, but eventually there'll come a day when I try to add one more dependency, and it relies on a newer version of some dependency I already have. I update that dependency, but then break the dependency requirements for 5 more libraries. Sometimes the most innocuous addition can end up causing an update that ripples through half my dependencies. And, when I make those updates, I'm hoping that my tests will cover the obscure corner cases that might break.

During development, I got into the habit of updating my dependencies daily or weekly, banking on the idea that smaller steps will be easier to handle. But this doesn't help down the road, after I've moved on. When I come back to a project after only a year or so I assume I'm going to be battling with dependencies for a while before I can start being productive.

And because dependencies depend on each other too, each additional dependency will cost you a more than linear amount of time down the road.

Am I secure?

Many library repositories monitor for reported security vulnerabilities in the libraries they host. When you install a JavaScript library using NPM, for example, it will perform a quick audit of all your packages, and not just the ones you're installing.

```
$ npm add jest
npm warn deprecated glob@7.2.3: Glob versions prior to v9 are no longer supported

added 275 packages, and audited 310 packages in 13s
```

```
33 vulnerabilities (6 moderate, 19 high, 8 critical)
Run `npm audit` for details.
```

In this example, the deprecation warning is caused by adding Jest, but the 33 vulnerabilities were all in existing packages.

During development, I see these reports whenever I refresh my dependencies. After that time, I get warnings from GitHub if it notices a security issue in my code base.

Idea 3 Make each dependency a deliberate choice

Here's what I currently do:

- I don't use a whole dependency for just a single function.
- I don't use a dependency for something I could simply write.
- I try to keep dependencies updated during development.
- I spend a little time checking the provenance of dependencies before adding them.

Here's what I should get better at doing:

- Investigate services which monitor the security status of dependencies.
- Get more consistent about wrapping dependencies to isolate them from the rest of my code.
- Set aside a day a quarter to fetch and build old projects, updating their dependencies if possible.

Your checklists will be different. But having a your own deliberate decision-making process will help you to weight the benefits of each dependency you add.

Investigate

Open up your current project and have a look at all the top-level dependencies (the ones you and your team have declared).

- Do you know what each does?
- Do you know how your code is using each? Is that use trivial? Can you remove the dependency and simply implement the functionality you were using?

Now look at the list of all the dependencies (not just the ones you explicitly included). These will typically be in something like a .lock file.

- How many are there?
- Does that number scare you?
- Are you going to do something about it?