Dependency management

- Transitive dependencies and the commons

Transitive dependencies and the commons

In this article, I want to explain why I personally only work with programming languages anymore that allow conflicting transitive dependencies, and why this matters for the purpose of building a commons for software.

Types of dependency structures

There are a lot of considerations in designing dependency systems, but there are two axes I've found to be particularly relevant to the topic of a software commons: nested vs. flat dependencies, and system-global vs. project-local dependencies.

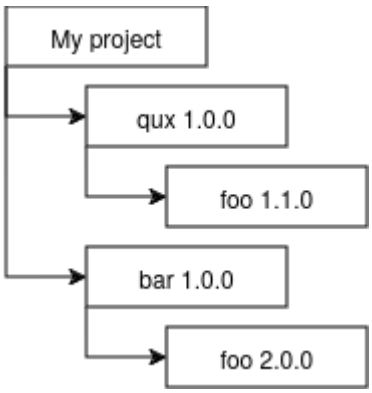
Nested vs. flat dependencies

There are roughly two ways to handle transitive dependencies - that is, dependencies of your dependencies:

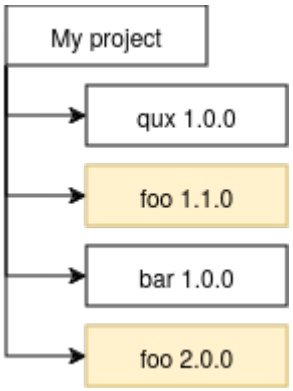
1. Either you make the whole dependency set a 'flat' one, where every dependency is a top-level one, or
2. You represent the dependency structure as a *tree*, where each dependency in your dependency set has *its own, isolated* dependency set internally.

I'm talking about the conceptual structure here, so it doesn't actually matter how these dependencies are stored on disk, but I'll illustrate the two forms below.

This is what a nested dependency tree in some project might look like:



And this would be the *equivalent* tree in a flat dependency structure:



This also immediately shows the primary limitation of a flat dependency set: you cannot have conflicting versions of a transitive dependency in your project! This is what most of the process of "dependency solving" is about - of all the theoretically possible versions of every dependency in your project, find the set that actually works together, ie. where every dependency matches the version constraints for *all* of its occurrences in the project.

Project-local vs. system-global dependencies

Another important, and related, design decision in a dependency system is whether dependencies are *isolated to the project*, or whether you have a single dependency set that's used system-wide. This is somewhat self-explanatory; if your dependencies are *project-local* then that means that they are stored within the project, but if they are *system-global* then there's a system-wide dependency set of some sort, and so the project gets its dependencies from the environment.

Some examples

Here are some examples of different combinations of these properties, and where you might find them:

- **System-global, flat:** Python (without virtual environment tools). There is a single system-wide collection of Python packages, that every piece of Python software on the system uses. Can be turned into "Project-local, flat" using `virtualenv`. Another example would be C-style libraries.
- **Project-local, flat:** PHP, with Composer. Packages are installed to a `vendor` directory in your project, but only one version of a given dependency can be installed at a time.
- **System-global, nested:** Nix. There is a system-wide package store called the "Nix store", where every unique variant of a package is stored under a deterministic hash, and every dependency within a piece of software is referenced by hash from that store explicitly.
- **Project-local, nested:** Node.js. Each project has its own `node_modules` folder, and each dependency has *its* own `node_modules` folder containing its transitive dependencies, and so on.

So why does any of this actually matter?

It might seem like all of this is just an implementation detail, and it's the problem of the package manager developers to deal with. But the choice of dependency model actually has a big impact on how people use the dependency system.

The cost of conflict

The problem at the center of all of this, is that *dependency conflicts are not free*. Every time you run into a dependency conflict, you have to stop what you are doing and resolve a conflict. Resolving it may require anything from a small change to a complete architectural overhaul of your code, like in the case where a new version of a critical dependency introduced a different design paradigm.

Now you might think "huh, but I rarely run into that", and that is likely correct - but it's not because the problem doesn't happen. What tends to happen in mature language ecosystems, is that the whole ecosystem centers around a handful of large frameworks over time, where the maintainers do all this work of resolving conflicts preventatively; they coordinate with maintainers of other dependencies, for example, to make sure that these conflicts do not occur.

This has a large maintenance cost to maintainers, and indirectly also a cost to you - it means that that time is not spent on, for example, nice new features or usability improvements in the tools that you use. The cost is still there, it's just very difficult to see if you are not a maintainer of a large framework.

Frameworks and libraries

This also touches on another consequence of working with conflict-prone dependency systems: they incentivize centralization of the ecosystem around a handful of major *frameworks*, that are usually quite opinionated about how to use them. In a vacuum, small and single-responsibility libraries would be the optimal structure of an ecosystem, but that is simply not a sustainable model when your transitive dependencies can conflict; every dependency you add would superlinearly increase the chance of running into a conflict.

These frameworks are usually acceptable if you work on common systems, that solve common problems; there have been many people before you building similar things, and so the framework will likely have been designed to account for it. But it's a deathly barrier for unconventional or innovative projects, which do not fit into that mold; they are severely disadvantaged because in a framework-heavy ecosystem, every package comes with a large set of assumptions about what you'll be doing with it, and they're usually not going to be the right ones. Leaving you to either not use packages at all, or spend half your time working around them.

Consequences for the commons

A more abstract way in which this problem occurs, is in its impact to the commons. The idea of a 'software commons' is simple; a large, public, shared, freely accessible and usable collection of software that anyone can build upon according to their needs and contribute to according to their ability, resulting in a low-friction way to collaborate on software at a large scale. Some of the

idealized consequences of such a commons would be that every problem only needs to be solved exactly once, and it will forever be reliably solved for everyone, and we can all collectively move on to solving other problems.

This is a laudable goal, but it too is harmed by conflict-prone dependency systems. For this goal to be achievable, there must be some sort of distribution format for 'software' that is universally usable, assumption-free, and isolated from the rest of the environment, so that it is guaranteed to fit into any project that has the problem it is designed to solve. But a flat or even system-global dependency model cannot do that - in such a model, it is possible for one piece of software to make it impossible to use another, after all this is what a dependency conflict is.

In other words, to achieve a true software commons on a *technical* level (the social requirements are for another article), we need a nested, project-local dependency mechanism - or at least a mechanism that can approximate or simulate those properties in some way.

So why are dependency systems so conflict-prone?

So given all of that, the answer would seem obvious, right? Just build nested, project-local dependency systems! And that does indeed solve these issues, but it brings some problems of its own.

Duplication

One of the most obvious problems, but also one of the easiest to solve, is that of duplication. If there are two uses of the same dependency in different parts of the dependency tree, you ideally want those to use the same copy to save space and resources, and indeed this is exactly the typical justification for a flat dependency space. This also applies to compilation; you'd usually want to avoid compiling more than one copy of the same library.

But there is a better way, and it is implemented today by systems like npm: a nested dependency tree which opportunistically moves dependencies to the top level when they are conflict-free. This way, they are only stored in a nested structure *on disk* when it is necessary to preserve the guarantees of a conflict-free dependency system, ie. when otherwise a dependency conflict would occur. This could be considered a hybrid form between nested and flattened dependencies, and is pretty close to an optimal representation.

The duplication problem exists in another form, and its solution is *the* optimal representation: duplication between projects. Two pieces of independent end-user software might use the same version of the same dependency, and you would probably want to reuse a single copy for all the same reasons as above. This is typically used as a justification for system-global dependency systems.

But here again, there is a better option, and this time it is truly an optimal representation: a single shared system-global store of packages, identified by some sort of unique identifier, with the software pointing to specific copies within that store. This optimally deduplicates everything, but still allows conflicting implementations to exist. This exists today in Nix (where each store entry is hashed and referenced by path from the dependent) and pnpm (an alternative Node.js package manager where the store is keyed by *version* and symlinks and hardlinks are used to access it in an npm-compatible manner).

Nominal types

Unfortunately, there is also a more difficult problem - it affects only a subset of languages, and explains why Node.js *does* have nested dependencies while a lot of other new systems do not. That problem is the nominal typing problem.

If you have a system with nominal typing, then that means that types are not identified by *what they are shaped like (as in structural typing), but *by what they are called*, or more accurately by their *identity*. In a typical nominal typing system, if you define the same thing under the same name twice, but in different files, they are different types.

This poses an obvious problem for a nested dependency system: if you can have two different copies of a dependency in your tree, that means you can also have two different types that are *supposed* to be identical! This would cause a lot of issues - for example, say that a value in a dependency is generated by a transitive dependency, and consumed by a different dependency that uses a different version of that same transitive dependency... the value generated by one copy would be rejected by the other, for not being the same type.

This is what can happen, for example, in Rust - Cargo will nominally let you have conflicting versions, but as soon as you try to exchange values between those copies in your code, you'll encounter a type mismatch.

There are some theoretical language-level solutions to this problem, for example in the form of *type adapters* - specification on how one copy of a type may be converted to another copy of a type. But this is a non-trivial thing to account for in a language design, and to this date I have not seen any major languages that have such a mechanism. Which means that nominally typed languages are, generally, stuck with flat dependencies.

(If you're wondering how this problem is overcome *without* nominal typing: the answer is that you're mostly just relying on the internal structure of types not changing in a breaking way between versions, or at least not without also changing the attribute names or, in a structurally typed system, the internal types. That *sounds* unreliable, but in practice it is very rare to run into situations where this goes wrong, to the point that it's barely worth worrying about.)

But even if this problem were overcome, there's another one.

Backwards compatibility

Dependencies are, almost always, something that is deeply integrated into a language. Whether through the design of the import syntax, or the compiler's lookup rules, or anything else, there's usually *something* in the design of a language that severely constrains the possibilities for package management. Nested dependencies can work for Node.js because CommonJS accounted for the needs of a nested dependency system from the start, and it is virtually impossible to retrofit it into most existing systems.

For the same reason that a software commons is a possible concept, dependencies are also subject to the network effect - they are a social endeavour, an exercise in interoperation and labour delegation, and that means that there is an immense ecosystem cost associated with breaking dependency interoperability - just ask anyone who has had to try fitting an ES Module into a CommonJS project, for example, or anyone who has gone through the Python 2 to 3 transition. This makes changing the dependency system a very unappealing move.

So in practice, a lot of languages simply aren't able to adopt a nested dependency system, because it would break everything they have today. For the most part, only new languages can adopt nested dependencies, and most new languages are going to be borrowing ideas from existing languages, which... have flat dependencies. Among other things, I'm hoping that this article might serve as an inspiration to choose differently.

My personal view

Now, to get back to why I, personally, don't want to work with conflict-prone languages anymore, which has to do with the 'software commons' point mentioned earlier. I have many motivations behind the projects I work on, but one of them is the desire to build on a software commons using FOSS; to build reliable implementations for solving problems that are generically usable, ergonomic, and just as useful in 20 years (or more) as they are today.

I do not think that this is achievable in a conflict-prone language. Even with the best possible API design that needs no changes, you would still need to periodically update dependencies to make sure that your dependency's transitive dependencies remain compatible with widely-used frameworks and tools. This makes it impossible to write 'forever libraries' that are written once and then, eventually, after real-world testing and improvements, done forever. The maintenance cost alone would become unsustainable.

The problem is made worse by conflict-prone dependency systems' preference for monolithic frameworks, as those necessarily are opinionated and make assumptions about the usecase; which, unlike a singular solution to a singular problem, is not something that will stand the test of time - needs change, and as such, so do common usecases. Therefore, 'forever libraries' cannot take the shape that a conflict-prone dependency system encourages.

In short, a conflict-prone dependency system simply throws up too many barriers to credibly and sustainably build a long-term software commons, and that means that whatever work I do in the context of such a system, does not contribute towards my actual goals. In practice this means that I am mostly stuck with Javascript today, and I am hoping to see more languages adopt a conflict-free dependency system in the future.