Monolithic vs. modular - what's the difference?

> This article was originally published at https://gist.github.com/joepie91/7f03a733a3a72d2396d6.

When you're developing in Node.js, you're likely to run into these terms - "monolithic" and "modular". They're usually used to describe the different types of frameworks and libraries; not just HTTP frameworks, but modules in general.

At a glance

- **Monolithic:** "Batteries-included" and typically tightly coupled, it tries to include all the stuff that's needed for common usecases. An example of a monolithic web framework would be Sails.js.
- **Modular:** "Minimal" and loosely coupled. Only includes the bare minimum of functionality and structure, and the rest is a plugin. Fundamentally, it generally only has a single 'responsibility'. An example of a modular web framework would be Express.

Coupled?

In software development, the terms "tightly coupled" and "loosely coupled" are used to indicate how much components rely on each other; or more specifically, how many assumptions they make about each other. This directly translates to how easy it is to replace and change them.

- **Tightly coupled:** Highly cohesive code, where every part of the code makes assumptions about every other part of the code.
- **Loosely coupled:** Very "separated" code, where every part of the code communicates with other parts through more-or-less standardized and neutral interfaces.

While tight coupling can sometimes result in slightly more performant code and very occasionally makes it easier to build a 'mental model', loosely coupled code is much easier to understand and maintain - as the inner workings of a component are separated from its interface or API, you can make many more assumptions about how it behaves.

Loosely coupled code is often centered around 'events' and data - a component 'emits' changes that occur, with data attached to them, and other components may optionally 'listen' to those events and do something with it. However, the emitting component has no idea who (if anybody!)

is listening, and cannot make assumptions about what the data is going to be used for.

What this means in practice, is that loosely coupled (and modular!) code rarely needs to be changed - once it is written, has a well-defined set of events and methods, and is free of bugs, it no longer needs to change. If an application wants to start using the data differently, it doesn't require changes in the component; the data is still of the same format, and the application can simply process it differently.

This is only one example, of course - loose coupling is more of a practice than a pattern. The exact implementation depends on your usecase. A quick checklist to determine how loosely coupled your code is:

- ☐ **Does your component rely on external state?** This is an absolute no-no. Your component cannot rely on any state outside of the component itself. It may not make any assumptions about the application *whatsoever*. Don't even rely on configuration files or other filesystem files - all such data must be passed in by the application explicitly, always. What isn't in the component itself, doesn't exist.
- ☐ **How many assumptions does it make about how the result will be used?** Loosely coupled code shouldn't care about how its output will be used, whether it's a return value or an event. The output just needs to be consistent, documented, and neutral.
- ☐ **How many custom 'types' are used?** Loosely coupled code should generally only accept objects that are defined on a language or runtime level, and in common use. Arrays and A+ promises are fine, for example - a proprietary representation of an ongoing task is not.
- ☐ **If you *need* a custom type, how simple is it?** If absolutely needed, your custom object type should be as plain as possible - just a plain Javascript object, optimally. It should be well-documented, and not duplicate an existing implementation to represent this kind of data. Ideally, it should be defined in a separate project, just for documenting the type; that way, others can implement it as well.

In this section, I've used the terms "component" and "application", but these are interchangeable with "callee"/"caller", and "provider"/"consumer". The principles remain the same.

The trade-offs

At first, a monolithic framework might look easier - after all, it already includes everything you think you're going to need. In the long run, however, you're likely to run into situations where the framework just doesn't *quite* work how you want it to, and you have to spend time trying to work around it. This problem gets worse if your usecase is more unusual - because the framework developers didn't keep in mind your usecase - but it's a risk that always exists to some degree.

Initially, a modular framework might look harder - you have to figure out what components to use for yourself. That's a one-time cost, however; the majority of modules are reusable across projects, so after your first project you'll have a good idea of what to start with. The remaining usecase-

specific modules would've been just as much of a problem in a monolithic framework, where they likely wouldn't have existed to begin with.

Another consideration is the possibility to 'swap out' components. What if there's a bug in the framework that you're unable (or not allowed) to fix? When building your application modularly, you can simply get rid of the offending component and replace it with a different one; this usually doesn't take more than a few minutes, because components are typically small and only do one thing.

In a monolithic framework, this is more problematic - the component is an inherent part of the framework, and replacing it may be impossible or extremely hard, depending on how many assumptions the framework makes. You will almost certainly end up implementing a workaround of some sort, which can take *hours*; you need to understand the framework's codebase, the component you're using, and the exact reason why it's failing. Then you need to write code that works around it, sometimes even having to 'monkey-patch' framework methods.

Relatedly, you may find out halfway through the project that the framework doesn't support your usecase as well as you thought it would. Now you have to either *replace the entire framework*, or build hacks upon hacks to make it 'work' somehow; well enough to convince your boss or client, anyway. The higher cost for on-boarding new developers (as they have to learn an entire framework, not just the bits you're interested in *right now*), only compounds this problem - now they *also* have to learn why all those workarounds exist.

In summary, the tradeoffs look like this:

- **Monolithic:** Slightly faster to get started with, but less control over its workings, more chance of the framework not supporting your usecase, and higher long-term maintenance cost due to the inevitable need for workarounds.
- **Modular:** Takes slightly longer to get started on your first project, but total control over its workings, practically every usecase is supported, and long-term maintenance is cheaper.

The "it's just a prototype!" argument

When explaining this to people, a common justification for picking a monolithic framework is that "it's just a prototype!", or "it's just an MVP!", with the implication that it can be changed later. In reality, it usually can't.

Try explaining to your boss that you want to throw out the working(!) code you have, and rewrite everything from the ground up in a different, more maintainable framework. The best response that you're likely to get, is your boss questioning why you didn't use that framework to begin with - but more likely, the answer is "no", and you're going to be stuck with your hard-to-maintain monolithic codebase for the rest of the project or your employment, whichever terminates first.

Again, the cost of a modular codebase is a one-time cost. After your first project, you already know where to find most modules you need, and building on a modular framework will not be more expensive than building on a monolithic one. Don't fall into the "prototype trap", and do it right

from day one. You're likely to be stuck with it for the rest of your employment.

Revision #1
Created 2024-12-11 01:25:24 UTC by joepie91
Updated 2024-12-11 18:18:00 UTC by joepie91