Major Version Numbers are Not Sacred

23 May 2022 - Positano, Italy

Ten years ago, I sat down and wrote the first version of what became the Semantic Versioning (SemVer) spec. I was tired of everyone using version numbers in whatever way they wanted and knew we could do better if we all agreed on what each part of a version number meant and how those numbers should change as the project’s code changed.

It worked.

Today, SemVer is one of the world’s most well-known and heavily adopted versioning schemes. Several prominent package ecosystems (like Node’s npm and Rust’s Cargo) built-in SemVer’s concepts from day one. npm alone has more than 1.8 million packages, nearly all of which are versioned with SemVer.

While it’s unrealistic to expect every package to perfectly adhere to SemVer all the time (mistakes do happen), I’ve seen in practice that the vast majority of package developers are quite careful about version number changes and try to follow SemVer in good faith. This can be hard, though, especially when it comes to breaking changes.

In the time before Semantic Versioning, major version number changes often signified large shifts in the architecture, approach, or broad capabilities of the underlying software. In that world, the minor versions would contain breaking changes, especially for products with large API surface areas. Bumping the major version often corresponded to a marketing push to communicate those improvements to the world. This produced a natural outcome: increasing the major version number was a big deal.

Today, in SemVer ecosystems, it’s **still** a big deal. And I think that’s a problem.

Our collective hesitance to bump the major version of a package is so strong that we sometimes concoct elaborate justifications as to why a breaking change can be included in a minor version release.

“Not many people are using that feature yet; it’ll be fine.”

Or,

“It was clearly intended to be an experimental feature, so we should be able to retroactively mark it as such and then break it in minor, right? Right? RIGHT????”.

Or plainly,

“There isn’t enough to warrant a major version bump; we’re not going to release a major version for some tiny breaking change.”

Either SemVer means something, or it doesn’t. I choose to believe that it does. I’ve seen the benefits to huge package ecosystems and upgrade processes. If we want to live in a world where SemVer improves our collective experience as much as possible, it means we need to believe something new:

Major version numbers are not sacred.

In practice, this means two things:

1. When you need to release a breaking change—**any** breaking change—you do it in a major version. Period. No excuses.
2. You need to find a different way to communicate more significant project-wide changes and improvements. Major version numbers can’t do double duty as both breaking-change indicator **and** marketing hook.

The first one should be easy. You just have to internalize that major version numbers are not sacred, you’re not going to run out of them, and it’s your duty as a responsible release manager to always indicate that a breaking change is included within.

Of course, just because numbers are free doesn't mean it's wise to break your API on every release. Each breaking change means manual code modifications for some users of your package. Break the API too frequently, and you'll risk fatiguing your users by forcing them to dig through your release notes (you do have release notes, right?) and working through the breaking changes to see if any of them are relevant and need action.

The second one is harder. There have been various proposals over the years, a common one being that SemVer should add a 4th part, perhaps called "epoch," that is the property of the marketing department. So you'd see `version 2.3.1.0` instead of just `version 3.1.0`, and the `2` indicates that this is the 2nd epochal version corresponding to some marketing event or a larger shift in the library. It's not a horrible idea, but it does increase the visual complexity of the version number quite a bit and would be challenging to get adopted throughout the various SemVer package ecosystems.

Another idea (and one that doesn't require SemVer itself to change) is to use a code name or marketing name to associate with a range of major versions. When the name changes, **that** is the marketing event. For instance, Ubuntu is famous for its wacky animal code names like Trusty Tahr and Bionic Beaver. With a bit of extra discipline, it's not hard to imagine starting with the letter A and naming each "epoch" with a name starting with the next letter in the alphabet, giving users some sense of how many big iterations the software has been through.

These points are not simply an academic exercise for me. In fact, this post is in part an explanation for why we're about to release RedwoodJS 2.0 only a bit over a month after releasing 1.0.

Weird, right? See how ingrained it has become to think that major versions should only happen once a year?

So why do we need a 2.0 so soon?

1. Being a full-stack web framework, RedwoodJS has an extensive API surface area. There is simply a lot of functionality being worked on.
2. We need to move quickly and iterate to stay relevant. The web framework space is a constantly moving target, and success will require rapidly improving capabilities and developer experience.
3. Not all changes can maintain backward compatibility without incurring unacceptable tech debt or complexity increases in the codebase. There are real development, maintenance, and velocity costs associated with supporting both the old and new ways of doing something. We need to be able to make the call to move forward with new ideas without having to incur these costs or wait many months until a marketing-approved major release can be rolled. The specific breaking change that is prompting 2.0 is this set of changes to the Baremetal deploy strategy.
4. We are unwilling to compromise the trust of our users by finding some excuse for shipping breaking changes in a minor release. If you know we're serious about marking breaking changes with a major release, you can spend less time verifying that our minor releases are safe upgrades.

Ok, even if all that's true, how are we going to reduce the impact of releasing breaking changes so often (perhaps monthly)?

1. Every minor or major release gets a serious write-up of everything that changed and how to adapt your code to any breaking changes. Even during the pre-1.0 days of RedwoodJS, our team was obsessive about top-notch, comprehensive release notes, and we've gotten very good at making them as useful as possible. See the RedwoodJS v1.0.0 release notes for an example of what I mean.
2. On top of the release notes, whenever possible, we provide executable code mods to automatically upgrade your code to work with the new release! This is a non-trivial undertaking, but by taking the pain out of upgrades, we believe we can do more breaking changes with less impact, leading to a better framework and improved DX more quickly. The Go programming language famously provided code mods to reduce the impact of their frequent breaking changes during the initial development of the language, giving them free license to make sweeping changes without having to support every syntactical mistake they ever made.
3. I actually prefer more frequent major versions with smaller sets of breaking changes. There is a real danger in saving up all your significant changes and releasing them in one whopping major version once a year. In the early days of GitHub, when we would need to upgrade Ruby on Rails to the latest major version, it would often take **weeks or months** to accomplish, even with a dedicated developer on the task. Once, because of how extensive the changes needed to be, the upgrade branch started to deviate so much from the main branch that it became too unwieldy to merge, and we had to throw it away and start over with a different approach. Spreading upgrade work out throughout the year would have saved us a lot of time and anguish.

In an effort to drink my own philosophical champagne, I am excited to very soon deliver to you **RedwoodJS 2.0**, part of the **Arapaho Forest** epoch. Yep, you guessed it, each epochal version of Redwood will be named after a US national forest, and when we announce a new epochal version (that starts with a B), you can be sure you'll know about it. In the meantime, you'll see a number of major versions of Redwood (all with the Arapaho Forest epoch name) without a ton of fanfare surrounding the releases.

I want to live in a world where every breaking change comes gift-wrapped in a major release. I want the promise of SemVer to be fully realized. I think we can get there by rejecting the tyranny of sacred major version numbers. If you feel the same, I hope you'll join me in embracing this philosophy.

Discuss this post on Hacker News

Related Posts

- 07 Apr 2022 » Introducing the Redwood Startup Fund
- 04 Apr 2022 » Announcing RedwoodJS 1.0 and $1M Funding
- 15 Jun 2020 » Committing $250k this Year to Racial Justice Efforts

Tom Preston-Werner
Cofounder of GitHub, Chatterbug
tom@mojombo.com

github.com/mojombo
twitter.com/mojombo