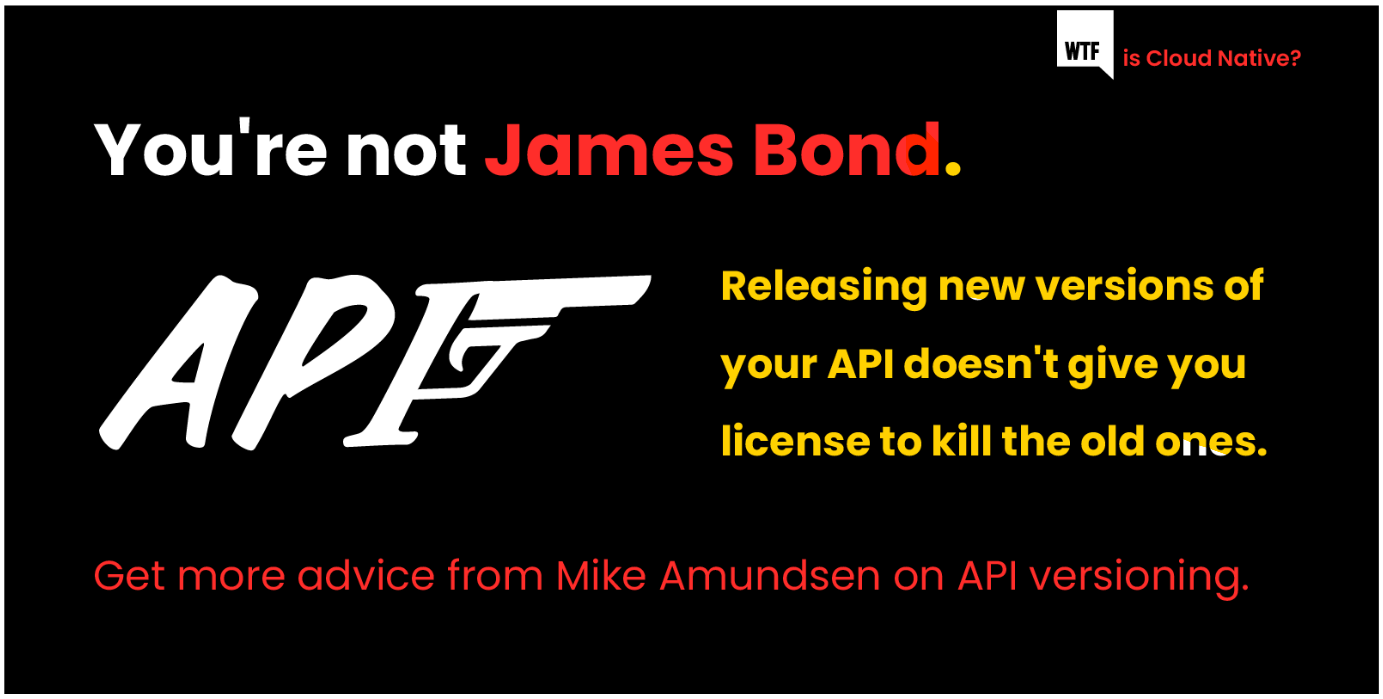
WTF is SRE Conference is coming in hot for 2022 | Register for updates today!

Container Solutions

share:


WTF is Cloud Native?
You're not James Bond.
API
Releasing new versions of your API doesn't give you license to kill the old ones.
Get more advice from Mike Amundsen on API versioning.

WTF IS CLOUD NATIVE

API Versioning: What Is It and Why Is It So Hard?

Mike Amundsen

April 1, 2021
13 minutes Read

If you want to start a debate among API technologists, just ask them to share their thoughts on "API versioning". It's a sure bet you'll uncover some strong feelings in short order. The term "API versioning" has become synonymous with "changing the API" and that is the first hurdle to sorting out a smart strategy for supporting continuous change for published APIs without creating needless problems for API consumers.

Most APIs don't need versioning; they need the ability to support compatible changes over time (not as flashy a term, right?). There are plenty of examples of technology supporting compatible changes like TCP/IP, HTTP, and HTML. Most programming languages do it, too. Successfully implementing compatible changes involves making a commitment to supporting change and sticking to that commitment over the long term. It's that simple—but it's not so easy.

The most effective change-compatible APIs can continue to add features without ever breaking already existing API consumers—even ones created years ago. The least effective API changes require all API producers and consumers to rewrite and re-release their code each time the API design is updated. The key to success is to design-in support for compatible changes from the very start. With that pillar in place, the process of implementing and releasing compatible editions of the API becomes more consistent, predictable, and resilient over time.

And that's what we all need, right?

Versioning is a 'middle finger to your customers'

One of the key challenges of maintaining APIs on the web is dealing with change over time. Put simply, once you release your API and people start using it, that interface represents a dependency. API consumers depend on the API to be stable and reliable going forward.

But all too often, server-side teams (API producers) ignore that dependency. Instead, they simply make changes with little regard to the cost of effort they are passing along to API consumers. As Roy Fielding (of REST fame) put it in a tweet in 2013:

"... a 'v1' is a middle finger to your API customers..."

That middle finger from API producers to API consumers, essentially means, "yeah, we changed our API, deal with it." The good news is the common practice is to just start a new version at a new URL—moving from `api.example.org/v1/customers` to `api.example.org/v2/customers`. At least that isolates the change to some new branch. But then it gets worse; These sever-side decisions usually come with a deadline attached (e.g. "We'll shut off the existing API version in 180 days").

Either way, API consumers are stuck holding the bag. I've even seen companies create a new version of their API just to add new features. Now, if a consumer wants to use the new feature, they need to stop their current work and commit time and budget to a new sprint just to resolve any conflicts with the new version of the API they consume. And sometimes these API consumer teams don't want or need the new features anyway. All wasted expense.

Add to all this the recent move to faster releases and you can only imagine what the downstream cost of APIs can turn into. I've seen more than one API program spend its way right out of business by churning its way right down the drain.

Middle finger indeed.

If they can do it, you can, too.

Safely adding non-breaking changes to running systems is nothing new, of course. It is, in fact, the very bedrock of how Internet standards work. Alan Kay, regarded by many as the "father of personal computers," has noted that, over the decades, every line of code that built the TCP/IP stack has been replaced. Yet TCP/IP continues to work just fine throughout all these changes.

Another great example of this kind of supporting change over time without breaking is embedded into the HTTP protocol itself. There's an entire design effort built into the HTTP UPGRADE command that includes dedicated status codes (101 Switching Protocol and 426 Upgrade Required) along with a well-defined list of interactions to handle changing from one protocol to another.

Finally, lest you think that this level of change support is limited to low-level protocol elements, the HTML specification was built on the principle of "IgnoreUnknownTags". Things were moving so quickly in the early days of HTTP and HTML that the only safe way to continue to move the specification forward was to establish the rule that any browser that encountered an unrecognised element or element attribute should just act as if that tag was not there. This made it possible to rapidly update the functionality of HTML without adding to the "break-ability" of HTML client apps (browsers).

(Note: See Bert Bos' 2001 talk on "W3C, XML and standardization" for some insight into the early days of HTTP/HTML design.)

And all these examples can be traced back to a fundamental rule that guided the creation of most of the Internet standards themselves. The "Robustness Principle", coined by Jon Postel in 1980:

> *"Be conservative in what you do, be liberal in what you accept from others."*

So, we certainly have the ability to manage change over time without breaking. But why does it seem so difficult to implement and support? I think this comes from a lack of shared principles within a company's IT community. Essentially, we're too lazy to do the work.

But it doesn't have to be that way.

The Hippocratic Oath of APIs

The first rule of properly managing change over time with APIs is to not commit breaking changes to production. The reason is simple: Once you release your API and people start using it, that interface is a promise. And breaking that promise can mean losing customers. As Amazon's Werner Vogels puts it: "APIs are forever." In a 2016 blog post he said,

> *"[At Amazon] we knew that designing APIs was a very important task as we'd only have one chance to get it right."*

This is essentially a kind of "Hippocratic Oath" for API designers and developers: "First, do no harm."

Companies that rely on APIs to keep going know how important this promise can be. Especially companies that earn revenue from their APIs. When you are getting paid every time someone successfully uses your API, you start to make non-breaking changes a priority.

The 3 Rules of Backward Compatibility

There are some simple implementation rules you can use to reduce the likelihood of introducing a breaking change into your API ecosystem. Making these part of your API design and review practice is essential if you want to be successful at managing APIs over time.

Rule #1: Don't take anything away.

The first rule is that, once you release an API, you can't take things away from it. You can't take away a promised URL, an input parameter, or an output data point. Even if you *tell* developers in your documentation that some element of your API might disappear or change, you still can't do it. Because, well, developers.

The truth of this first rule is explained in what has become known as Hyrum's Law or "The Law of Implicit Dependencies":

> *"With a sufficient number of users of an API, it does not matter what you promise in the contract: all observable behaviors of your system will be depended on by somebody."*
> ***— Hyrum Wright, staff software engineer at Google.***

And that goes for enumerated values, too. If you have a field named `status` and document that it has five possible values, you can't later release an update to the API where `status` has only four possible values.

Rule #2: Don't change the meaning of anything.

The second rule states you can't change the meaning of an existing element of your API. If you promised that the `count` parameter in a filter statement meant "user count", you can't later change that same parameter to mean "page count".

Changing the meaning of things has the same effect as taking it away and replacing it with something else. That's a breaking change and an abdication of your Hippocratic Oath of the API.

Rule #3: All additions are optional.

The third rule is that any new features (URLs, input parameters, and output parameters) must be treated as optional. If your `addCustomer` operation took four required arguments in the initial release, you can't add a new, fifth, required argument. Again, that's essentially taking away the old `addCustomer` method and replacing it with a non-backward-compatible version of `addCustomer`.

Fine, but what if you are required to make a breaking change?

Stick a fork in it, you're done!

There are certainly cases where you have no choice but to introduce a breaking change. Maybe your team messed up and the API is leaking personally identifiable information or some regulations have changed and you are no longer allowed to accept or share data, etc. When that happens, it's time to get out your forks. Because a new "version" of an API is really just a fatal fork (in software parlance).

Forking is a pretty well-known term in software engineering. It dates back to online discussions via Usenet in the early 1980s and refers to creating a branch off the main trunk of a project. The assumption was that, one you branch off your fork, you are not expecting to merge back to the main trunk again. While we use main, branch, and merge commonly in software engineering today, APIs tend to "stay forked" over time and rarely merge back together again.

If you know you need to introduce a breaking change, fork your API and release it in a new URL space. Call that a "version" if you like. You're still not James Bond though, so be aware that releasing new versions does not give you license to kill the old ones. Instead, you need to run them for a time side-by-side. That avoids the whole "middle finger" scenario we covered earlier.

With side-by-side APIs in place, API consumers can continue to use their current API edition until they are ready to upgrade at some future point—on their own schedule, not yours.

A company that knows very well how to fork APIs for fun and profit is Salesforce. They have a habit of releasing a new edition of their API three times a year. Their Spring 2021 API is version 51. My last check shows they support every previous version back to Spring 2014 (that's v30!). Yep, Salesforce supports 20 past versions side-by-side.

Open-source champion Eric S. Raymond associated "forking" with a tragic or costly event. Creating software forks usually indicates an unresolvable schism and means lots of duplicated effort. And, as we've seen with the Salesforce model, that's true in the API space, too.

If you're committed to releasing breaking changes, you also need to commit to side-by-side release, documentation, online support, and so on.

Some parting shots

So, the whole versioning story is really all about change over time. When it comes to APIs, we can't stop time and we can't avoid change. Instead we need to come up with a stable, cost-effective way to deal with them.

The typical way API producers have handled this is to push the cost and effort onto API consumers. But that does not scale, especially in cases where customers are paying for the right to use your API. Each breaking change adds to the cost and burden of your API for the consumer and, if there's a competitor out there who can deliver at lower cost, you'll lose those customers.

All through the history of the Internet, the scalable solution has been to design services and APIs that support backward-compatible changes and avoid breaking consumers. TCP/IP, HTTP, and HTML are all great examples of this commitment to non-breaking changes.

And by adopting the "Hippocratic Oath of APIs" and sticking to the principle of "First, do no harm" to API consumers—along with the “Three Rules of Backward Compatibility”— you can introduce practices within your organization to support change over time while avoiding breaking API consumers.

Finally, in cases where you can't avoid breaking changes, you can fall back on side-by-side deployments to isolate the breaking and reduce the burden of change for your API client applications.

Companies that recognise the challenge and meet the demand will be well-placed to succeed as more and more of our business commerce is handled through APIs.

⟵ **Previous article**

Vaccines Are Saving the World! But Did Content Delivery Netw...

Next article ⟶

WTF Is Policy as Code?

Leave your Comment

Your name

Your email address

Your website (optional)

Your comment

Container Solutions needs the contact information you provide to us to contact you about our products and services. You may unsubscribe from these communications at any time. For information on how to unsubscribe, as well as our privacy practices and commitment to protecting your privacy, please review our Privacy Policy.

Post your comment

Container **Solutions**

Home

Events

Services

Latest News

Blog

Careers

Talk to us

WTF is Cloud Native

Terms and conditions

Privacy policy

Data Privacy for Job Seekers

Cookie policy

Events Code of Conduct

Talk to sales

info@container-Solutions.com

Stay In Touch

© 2020 Container Solutions