Semantic Versioning Explained

This lesson explains semantic versioning in detail with respect to the factor of having an API or not. Moreover, we also take a look at the problems faced due to custom versioning instead of semantic versioning.

We'll cover the following

Semantic versioning

Semantic versioning is easy to explain and leaves very little to interpretation. We have three numbers called major, minor, and patch. If we increment minor, a patch is reset to zero or one. Similarly, if major is incremented, both the minor and the patch are set to zero or one. The number that is incremented depends on the type of change we're releasing.

Given a version number `MAJOR.MINOR.PATCH`, we will increment each of the segments using the rules that follow:

- `PATCH` is incremented when we release bug fixes.
- `MINOR` is incremented when new functionality is added in a backward-compatible manner.
- `MAJOR` is incremented when changes are not backward compatible.

Versioning for an application having an API

If, for example, our application has an API, incrementing the major version would be a clear signal to our users that they would need to adapt or continue using the previous (older) major version (assuming we keep both, as we should). A change in the minor version would mean that users do not need to adapt, even though there

are new features included in the release. All other cases would increment only the patch version.

Versioning for an application without an API

Deducing which version to increment for an application that does not contain an API is a bit harder. If a change to a client-side web application is backwards compatible or not largely depends on how humans (the only users of a web app) perceive backwards compatibility. If, for example, we change the location of the fields in the login screen, we are likely backwards compatible. However, if we add a new required field, we are not and, therefore, we should increase the major version. That might sound confusing and hard to figure out for non-API based applications. But there is a more reliable way if we have automated testing. If one of the existing tests fails, we either introduced a bug, or we made a change that is not backwards compatible and therefore needs to increment the major version.

Problem with using custom versioning

I believe that semantic versioning should (almost) always be used. I cannot think of a reason why we shouldn't. It contains a clear set of rules that everyone can apply, and it allows everyone to know what type of change is released. However, I've seen teams that don't want to follow those rules. For example:

- Some want to increment a number at the end of **each sprint**.
- Others want to use **dates**, **quarters**, **project numbers**, or any other versioning system they are used to.

I believe that in most of those cases, the problem is in accepting change and not a *"real"* need for custom versioning. We tend to be too proud of what we did, even when the rest of the industry tells us that its time to move on.

The problem with using our own versioning schema is in *expectations*. New developers joining your company likely expect semantic versioning since it is by far the most commonly used. Developers working on applications that communicate with your application expect semantic versioning because they need to know whether you released a new feature and whether it is backward compatible. The end-users (e.g., people visiting your Web site) generally do not care about versioning, so please don't use them as an excuse to "reinvent the wheel".

Reasons to use semantic versioning

Reasons to use semantic versioning

I would say that there are two main reasons why you should use semantic versioning.

1. First of all, something or someone depends on your application. Those who depend on it need to know when an incompatibility is introduced (a change of the major version).
2. Even if that's not the case, the team developing the application should have an easy way to distinguish types of releases.

Now, you might say that you don't care about any of those reasons for using semantic versioning, so let me give additional motivation. Many of the tools you are using expect it. That's the power of conventions. If most of the teams use semantic versioning, many of the tools that interact with releases will assume so. As a result, by choosing a different versioning schema, you might not be able to benefit from the "out-of-the-box" experience. Jenkins X is one of those tools. It assumes that you do want to use semantic versioning because that is the most commonly used schema. We'll see that in practice soon.

Before we proceed, I must make it clear that I am not against you using some other versioning schema. Please do if you have an excellent reason for it and if the benefits outweigh the effort. You will need to invest in overcoming hurdles created by not using something that is widely adopted. Coming up with your own way of doing things is great, as long as the reason for the deviation is based on the hope that you can do something better than others, not only because you like it more. When we do things differently for no good reason, we are going to pay a high price later in maintenance and investment. The need to comply with standards and conventions is not unique to versioning, but to almost everything we do. So, the short version of what I'm trying to say is that you should use standards and conventions unless you have a good reason not to.

That's enough of theory, we'll continue through practical hands-on examples of how Jenkins X helps us to version our releases.