Contents

Introduction

Given a version number `MAJOR.MINOR.PATCH` increment the:

1. `MAJOR` - version when you make incompatible API changes
2. `MINOR` - version when you add functionality in a backward compatible manner
3. `PATCH` - version when you make backward compatible bug fixes

Additional labels for `pre-release` and build `metadata` are available as extensions to the `MAJOR.MINOR.PATCH` format

In the world of software management there exists a dreaded place called dependency hell. The bigger your system grows and the more packages you integrate into your software, the more likely you are to find yourself one day in this pit of despair.

In systems with many dependencies, releasing new package versions can quickly become a nightmare. If the dependency specifications are too tight, you are in danger of version lock - the inability to upgrade a package without having to release a new version of every dependent package. If dependencies are specified too loosely you will inevitably be bitten by version promiscuity.

- `Version lock` - when a dependency specification are too strict - requires an exact version of a very narrow range, it creates a situation where dependent packages cannot be upgraded without releasing a new version of all packages that rely on them. Suppose the package A requires version `1.2.3` of package B. If package B updates to `1.3.0` you can not use the new version of package B unless you also release a new version of package A that explicitly allows `1.3.0` in its dependency specification, this cascades across the dependency tree, making upgrades cumbersome and error-prone
- `Version promiscuity` - when dependency specifications are too loose allowing any version you risk unexpected breakage because the software may end up using version of the dependencies that are incompatible or introduce bugs. Suppose that package A specified it works with `>=1.0.0` of package B, and a breaking change is introduced in `2.0.0` then package A might unexpectedly break when package B updates to `2.0.0`

As a solution to this problem we propose a simple set of rules and requirements that dictate how version numbers are assigned and incremented. These rules are based on but not necessarily limited to `pre-existing` widespread common practices in use in both closed and open source software. For this system to work, you

need to declare a public API. This may consist of documentation or be enforced by the code itself. Regardless, it is important that this API be clear and precise. Once you identify your public API, you communicate changes to it with specific increments to your version number. Consider a version format of `x.y.z` Bug fixes not affecting the API increment the patch version, backward compatible API additions/changes increment the minor version, and backward incompatible API changes increment the major version.

`This system is called semantic versioning, under this scheme version numbers and the way they change, convey meaning about the underlying code and what had been modified from one version to the next.`

Specification

Software using semantic `versioning` must declare a public api. This api could be declared in the code itself or exist strictly in documentation. However it is done, it should be precise and comprehensive.

- A normal version number must take the form of `x.y.z` where x, y and z are non-negative integers, and must not contain leading zeroes. X is the major version, Y is the minor version, and Z is the patch version. Each element must increase numerically.
- Once a versioned package has been released, the contents of that version must not be modified any modifications must be released as a new version.
- Major version 0.y.z is for initial development. Anything may change at any time. The public API should not be considered stable.
- Version 1.0.0 defines the public API. The way in which the version number is incremented after this release is dependent on this public api and how it changes.
- Patch version Z must be incremented if only backward compatible bug fixes are introduces. A bug fix is defined as an internal change that fixes incorrect behavior. 1.1.5
- Minor version Y must be incremented if new backward compatible functionality is introduced to the PUBLIC API. It must be incremented if any public API functionality is marked as deprecated. It maybe incremented if substantial new functionality or improvements are introduced within the private code. It may include patch level changes. Path version must be reset to 0 when minor version is incremented. 1.1.0
- Major version X must be incremented if any backward incompatible changes are introduced to the public API. It may also include minor and patch level changes. Patch and minor version must be reset to 0 when major version is incremented. 1.0.0
- A `pre-release` version may be denoted by appending a hyphen and a series of dot separated identifiers immediately following the patch version. Identifiers must comprise only ASCII alphanumeric and hyphens. Identifiers must not be empty. Numeric identifiers must not include leading zeroes. `Pre-release` versions have a lower precedence than the associated normal version. A `pre-release` version indicates that the version is unstable and might not satisfy the intended compatibility requirements as denoted by its associated normal version. 1.0.0-alpha, 1.0.0-alpha.1, 1.0.0-0.3.7 and so on.
- Build `metadata` may be denoted by appending a plus sign and a series of dot separated identifiers following the patch or `pre-release` version, Identifiers must comprise only ASCII alphanumeric and hyphens. Identifiers must not be empty build `metadata` a must be ignored when determining version precedence. Thus two versions that differ only in the build `metadata` have the same precedence. 1.0.0-alpha+001, 1.0.0+201303313144700, and so on.
- Precedence refers to how versions are compared to each other when ordered. Precedence must be calculated by separating the version into major minor patch and `pre-release` identifiers in that order

(build `metadata` does not figure into precedence)

- Precedence is determined by the first difference when comparing each of these identifiers from left to right as follows: major, minor and patch version are always compared numerically. 1.0.0 < 2.0.0 < 2.1.0 < 2.1.1
- When all the major, minor and patch version are equal a `pre-release` version has lower precedence than a normal version 1.0.0-alpha < 1.0.0
- Precedence for two `pre-release` versions with the same major, minor and patch version must be determined by comparing each dot separated identifier from left to right until a difference is found as follows:
    1. Identifiers consisting of only digits are compared numerically.
    2. Identifiers with letters or hyphens are compared lexically in ASCII sort order.
    3. Numeric identifiers always have lower precedence than non-numeric identifiers
    4. Precedence for two `pre-release` versions with the same major, minor and patch versions must be determined by comparing each dot separated identifier from left to right until a difference is found

Why use semantic versioning

This is not a new or revolutionary idea. In fact, you probably do something close to this already. The problem is that close is not good enough. Without compliance to some sort of formal specification version numbers are essentially useless for dependency management. By giving a name and clear definition to the ideas above, it becomes easy to communicate your intentions to the users of your software. Once these intentions are clear flexible but not too flexible dependency specifications can finally be made.

Frequently asked questions

How to deal with revision 0.y.z of initial development

The simplest thing to do is start your initial development release at 0.1.0 and then increment the minor version for each subsequent release.

How to know when to release 1.0.0

If your software is being used in production, it should probably already be 1.0.0 if you have a stable API on which users have come to depend you should be at 1.0.0. If you are worrying a lot about backward compatibility you should already be at 1.0.0

Doesn't this discourage rapid development and fact iteration ?

Major version zero is all about rapid development. If you are changing the API every day you should either still be in version 0.y.z or on a separate development branch working on the next major version.

If even the tiniest backward incompatible change to the public API require a major version, won't we end up at version 42.0.0 very quickly

This is a question of responsible development and foresight. Incompatible changes should not be introduced lightly to software that has a lot of dependent code. The cost that must be incurred to upgrade can be significant. Having to bump a major versions to release incompatible changes means you will think through the impact of your changes, and evaluate the cost/benefit ratio involved.

Documentation is too much work

It is your responsibility as a professional developer to properly document software that is intended for use by other people. Managing software complexity is a hugely important part of keeping a project efficient and that is hard to do if nobody knows how to use your software or what methods are safe to call. In the long run, semantic versioning and the insistence on a well defined public API can keep everyone and everything running smoothly.

What to do if accidentally a backward incompatible change is released as a minor version

As soon as you realize that your have broken the contract and the semantic versioning spec, fix the problem and release a new minor version that corrects the problem and restores the backward compatibility. Even under this circumstance, it is unacceptable to modify versioned releases. If it is appropriate, document the offending version and inform your users of the problem so that they are aware of the offending version.

How to handle deprecated functionality

Deprecating existing functionality is a normal part of software development and is often required to make forward progress. When you deprecate part of your public API, you should do two things - (1) update your documentation to let users know about the change, (2) issue a new minor release with the deprecation in place. Before you completely remove the functionality in a new major release there should be at least one minor release that contains the deprecation so that users can smoothly transition to the new API.

Does SemVer have a size limit of the version string

No, but use a good judgment, A 255 character version string is probably overkill, for example. Also specific systems may impose their own limits on the size of the string

Is “v1.2.3” a valid semantic version

No, that is not a semantic version, however prefixing a semantic version with a v is common way to indicate it is a version number. Abbreviating “version” as “v” is often seen with version control.