Semantic *Versioning* 2.0.0

Tom Preston-Werner

Summary

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.

Introduction

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 new versions of every dependent package). If dependencies are specified too loosely, you will inevitably be bitten by version promiscuity (assuming compatibility with more future versions than is reasonable). Dependency hell is where you are when version lock and/or version promiscuity prevent you from easily and safely moving your project forward.

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 first 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 (Major.Minor.Patch). 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.

We call this system "Semantic Versioning." Under this scheme, version numbers and the way they change convey meaning about the underlying code and what has been modified from one version to the next.

Semantic Versioning Specification (SemVer)

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.

1. 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.
2. A normal version number MUST take the form 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. For instance: 1.9.0 -> 1.10.0 -> 1.11.0.
3. 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.
4. Major version zero (0.y.z) is for initial development. Anything MAY change at any time. The public API SHOULD NOT be considered stable.
5. 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.
6. Patch version Z (x.y.Z | x > 0) MUST be incremented if only backward compatible bug fixes are introduced. A bug fix is defined as an internal change that fixes incorrect behavior.
7. Minor version Y (x.Y.z | x > 0) 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 MAY be incremented if substantial new functionality or improvements are introduced within the private code. It MAY include patch level changes. Patch version MUST be reset to 0 when minor version is incremented.
8. Major version X (X.y.z | X > 0) 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 versions MUST be reset to 0 when major version is incremented.
9. 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 alphanumerics and hyphens [0-9A-Za-z-]. 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. Examples: 1.0.0-alpha, 1.0.0-alpha.1, 1.0.0-0.3.7, 1.0.0-x.7.z.92, 1.0.0-x-y-z.--.
10. Build metadata MAY be denoted by appending a plus sign and a series of dot separated identifiers immediately following the patch or pre-release version. Identifiers MUST comprise only ASCII alphanumerics and hyphens [0-9A-Za-z-]. Identifiers MUST NOT be empty. Build metadata MUST be ignored when determining version precedence. Thus two versions that differ only in the build metadata, have the same precedence. Examples: 1.0.0-alpha+001, 1.0.0+20130313144700, 1.0.0-beta+exp.sha.5114f85, 1.0.0+21AF26D3----117B344092BD.
11. Precedence refers to how versions are compared to each other when ordered.
    1. 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).
    2. Precedence is determined by the first difference when comparing each of these identifiers from left to right as follows: Major, minor, and patch versions are always compared numerically.

       Example: 1.0.0 < 2.0.0 < 2.1.0 < 2.1.1.
    3. When major, minor, and patch are equal, a pre-release version has lower precedence than a normal version:

       Example: 1.0.0-alpha < 1.0.0.
    4. 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. A larger set of pre-release fields has a higher precedence than a smaller set, if all of the preceding identifiers are equal.

       Example: 1.0.0-alpha < 1.0.0-alpha.1 < 1.0.0-alpha.beta < 1.0.0-beta < 1.0.0-beta.2 < 1.0.0-beta.11 < 1.0.0-rc.1 < 1.0.0.

Backus–Naur Form Grammar for Valid SemVer Versions

```
<valid semver> ::= <version core>
                 | <version core> "-" <pre-release>
                 | <version core> "+" <build>
                 | <version core> "-" <pre-release> "+" <build>

<version core> ::= <major> "." <minor> "." <patch>

<major> ::= <numeric identifier>

<minor> ::= <numeric identifier>

<patch> ::= <numeric identifier>

<pre-release> ::= <dot-separated pre-release identifiers>

<dot-separated pre-release identifiers> ::= <pre-release identifier>
                                          | <pre-release identifier> "." <dot-separated pre-release identifiers>

<build> ::= <dot-separated build identifiers>

<dot-separated build identifiers> ::= <build identifier>
                                    | <build identifier> "." <dot-separated build identifiers>

<pre-release identifier> ::= <alphanumeric identifier>
                           | <numeric identifier>

<build identifier> ::= <alphanumeric identifier>
                     | <digits>

<alphanumeric identifier> ::= <non-digit>
                            | <non-digit> <identifier characters>
                            | <identifier characters> <non-digit>
                            | <identifier characters> <non-digit> <identifier characters>

<numeric identifier> ::= "0"
                       | <positive digit>
                       | <positive digit> <digits>

<identifier characters> ::= <identifier character>
                          | <identifier character> <identifier characters>

<identifier character> ::= <digit>
                         | <non-digit>

<non-digit> ::= <letter>
              | "-"

<digits> ::= <digit>
           | <digit> <digits>

<digit> ::= "0"
          | <positive digit>

<positive digit> ::= "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

<letter> ::= "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J"
           | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T"
           | "U" | "V" | "W" | "X" | "Y" | "Z" | "a" | "b" | "c" | "d"
           | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m" | "n"
           | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x"
           | "y" | "z"
```

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" isn't 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 above ideas, 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.

A simple example will demonstrate how Semantic Versioning can make dependency hell a thing of the past. Consider a library called "Firetruck." It requires a Semantically Versioned package named "Ladder." At the time that Firetruck is created, Ladder is at version 3.1.0. Since Firetruck uses some functionality that was first introduced in 3.1.0, you can safely specify the Ladder dependency as greater than or equal to 3.1.0 but less than 4.0.0. Now, when Ladder version 3.1.1 and 3.2.0 become available, you can release them to your package management system and know that they will be compatible with existing dependent software.

As a responsible developer you will, of course, want to verify that any package upgrades function as advertised. The real world is a messy place; there's nothing we can do about that but be vigilant. What you can do is let Semantic Versioning provide you with a sane way to release and upgrade packages without having to roll new versions of dependent packages, saving you time and hassle.

If all of this sounds desirable, all you need to do to start using Semantic Versioning is to declare that you are doing so and then follow the rules. Link to this website from your README so others know the rules and can benefit from them.

FAQ

How should I deal with revisions in the 0.y.z initial development phase?

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 do I 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 1.0.0. If you're worrying a lot about backward compatibility, you should probably already be 1.0.0.

Doesn't this discourage rapid development and fast iteration?

Major version zero is all about rapid development. If you're 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 changes to the public API require a major version bump, won't I end up at version 42.0.0 very rapidly?

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 major versions to release incompatible changes means you'll think through the impact of your changes, and evaluate the cost/benefit ratio involved.

Documenting the entire public API is too much work!

It is your responsibility as a professional developer to properly document software that is intended for use by others. Managing software complexity is a hugely important part of keeping a project efficient, and that's 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 do I do if I accidentally release a backward incompatible change as a minor version?

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

What should I do if I update my own dependencies without changing the public API?

That would be considered compatible since it does not affect the public API. Software that explicitly depends on the same dependencies as your package should have their own dependency specifications and the author will notice any conflicts. Determining whether the change is a patch level or minor level modification depends on whether you updated your dependencies in order to fix a bug or introduce new functionality. We would usually expect additional code for the latter instance, in which case it's obviously a minor level increment.

What if I inadvertently alter the public API in a way that is not compliant with the version number change (i.e. the code incorrectly introduces a major breaking change in a patch release)?

Use your best judgment. If you have a huge audience that will be drastically impacted by changing the behavior back to what the public API intended, then it may be best to perform a major version release, even though the fix could strictly be considered a patch release. Remember, Semantic Versioning is all about conveying meaning by how the version number changes. If these changes are important to your users, use the version number to inform them.

How should I handle deprecating 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 on the version string?

No, but use 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 semantic version?

No, "v1.2.3" is not a semantic version. However, prefixing a semantic version with a "v" is a common way (in English) to indicate it is a version number. Abbreviating "version" as "v" is often seen with version control. Example: `git tag v1.2.3 -m "Release version 1.2.3"`, in which case "v1.2.3" is a tag name and the semantic version is "1.2.3".

Is there a suggested regular expression (RegEx) to check a SemVer string?

There are two. One with named groups for those systems that support them (PCRE [Perl Compatible Regular Expressions, i.e. Perl, PHP and R], Python and Go).

See: https://regex101.com/r/Ly7O1x/3/

```
^(?P<major>0|[1-9]\d*)\.(?P<minor>0|[1-9]\d*)\.(?P<patch>0|[1-9]\d*)(?:-(?P<prerelease>(?:0|[1-9]\d*|\d*[a-zA-Z-][0-9a-zA-Z-]*)(?:\.(?:0|[1-9]\d*|\d*[a-zA-Z-][0-9a-zA-Z-]*))*))?(?:\+(?P<buildmetadata>[0-9a-zA-Z-]+(?:\.[0-9a-zA-Z-]+)*))?$
```

And one with numbered capture groups instead (so cg1 = major, cg2 = minor, cg3 = patch, cg4 = prerelease and cg5 = buildmetadata) that is compatible with ECMA Script (JavaScript), PCRE (Perl Compatible Regular Expressions, i.e. Perl, PHP and R), Python and Go.

See: https://regex101.com/r/vkijKf/1/

```
^(0|[1-9]\d*)\.(0|[1-9]\d*)\.(0|[1-9]\d*)(?:-((?:0|[1-9]\d*|\d*[a-zA-Z-][0-9a-zA-Z-]*)(?:\.(?:0|[1-9]\d*|\d*[a-zA-Z-][0-9a-zA-Z-]*))*))?(?:\+([0-9a-zA-Z-]+(?:\.[0-9a-zA-Z-]+)*))?$
```

About

The Semantic Versioning specification was originally authored by Tom Preston-Werner, inventor of Gravatar and cofounder of GitHub.

If you'd like to leave feedback, please open an issue on GitHub.

License

Creative Commons — CC BY 3.0

Licensed under Creative Commons Attribution 3.0 Unported (CC BY 3.0) · creativecommons.org/licenses/by/3.0/