A simple guide to semantic versioning

If you're like me and stem from a breed that has been developing way before version control systems like SVN or GIT were born, you probably haven't been paying a lot of attention to versioning your code.

However, with modern modular development approaches using version control systems and package managers, semantic versioning has become a very important and often neglected aspect of development.

Benefits

- Keep a semantic historical track of a component
- Know which version of a component is no longer backwards compatible
- Avoid dependency hell when using a component in different places
- Allow a component to be distributed correctly with package managers

The semantic version format

In essence a semantic version looks like this:

Major.Minor.Patch

So v1.3.8 has a major component with a value of 1, a minor component with a value of 3 and a patch component with a value of 8.

Rules for increasing versions

When you make changes to your code, it is important that you update one or more components of the version as well, so the new version implicitly indicates the type of changes you made to your code.

So don't just change the version to anything you like, but apply the following rules:

1. Each increment happens numerically with an increment of 1.
2. When you **fix a bug** and your code stays **backwards compatible**, you increase the patch component:
3. v0.0.3 // Before bugfix

v0.0.4 // After bugfix

4. When you **add functionality** and your code stays **backwards compatible**, you increase the minor component and reset the patch component to zero:
5. v0.2.4 // Before addition of new functionality

v0.3.0 // After addition of new functionality

6. When you **implement changes** and your code becomes **backwards incompatible**, you increase the major component and reset the minor and patch components to zero:
7. v7.3.5 // Before implementing backwards incompatible changes

v8.0.0 // After implementing backwards incompatible changes

Some unofficial conventions

- 0.y.z (a major component of zero) is generally used for initial development
- When your code is used in production, you should increase to version 1.0.0 (a major component of one or higher)

A real world example

```
v0.0.0 // New project
v0.1.0 // Add some functionality
v0.2.0 // Add other new functionality
v0.2.1 // Fix bug
v0.3.0 // Add some functionality
v0.3.1 // Fix bug
v0.3.2 // Fix bug
v0.3.3 // Fix bug
v0.3.4 // Fix bug
v0.4.0 // Add some functionality
v0.4.1 // Fix bug
v0.4.2 // Fix bug
v1.0.0 // Code is being used in production
v1.1.0 // Add some functionality
v1.2.0 // Add other new functionality
v1.2.1 // Fix bug
v2.0.0 // Implement changes that causes public API of code to become backwards incompatible

...
```

Using semantic versions with Git

Git uses the concept of tags to assign versions to your Git repository:
To view the available tags of your repository:

```
$ git tag
```

To add a tag to your repository:

```
$ git tag -a v0.2.4 -m 'my version 0.2.4'
```

To push tags to a remote repository e.g. to origin:

```
$ git push origin --tags
```

View this page on tagging for more detailed options.

Using semantic versions with Bower

Bower uses Git tags as versions.

To specify the version of your own Bower component in your bower.json file:

```
{
  "name": "Package name",
  "version": "0.2.4"
}
```

To view the available versions of a registered Bower component:

```
$ bower info CoconutJS
  CoconutJS

  Versions:
  - v0.2.4
  - v0.2.0
  - v0.1.0
```

To install a specific version of a registered Bower component:

```
$ bower install CoconutJS#v0.2.4
```

To install a specific version of a non-registered component from GitHub using Bower:

```
$ bower install <github-username>/<github-repository>#<tag>
```

For example to install version 0.2.4 of CoconutJS from GitHub using Bower, you would:

```
$ bower install jvandemo/CoconutJS#v0.2.4
```

To use dependency versions in your bower.json file:

```
{
  "name": "Package name",
  "version": "0.2.4",
  "dependencies": {
    "jquery": "~2.0.0"
  }
}
```

Semantic version ranges

When dealing with dependencies you can use version ranges to prevent yourself from having to update your configuration each time a new version of a dependency is released.

Here's a list of some interesting semantic version ranges from the npmjs.org website:

- version: must match version exactly
- =version: same as just version
- >version: must be greater than version
- >=version: must be greater than or equal to version
- &lt;version: must be less thanversion
- &lt;=version: must be less than or equal to version

- ~1.2.0: must be at least as high as 1.2.0 but less than the next major version 2.0.0
- 1.2.x: must start with 1.2 but any digit may be used in place of the x
- *: matches any version
- "": (empty string) matches any version (same as *)
- version1 - version2: same as >=version1 &lt;=version2
- range1 - range2: passes if range1 or range2 are satisfied`

For example, these are valid ranges:
```
{ "dependencies" :
   {
      "foo" : "1.0.0 - 2.9999.9999",
      "bar" : ">=1.0.2 <2.1.2",
      "baz" : ">1.0.2 <=2.3.4",
      "boo" : "2.0.1",
      "qux" : "<1.0.0 || >=2.3.1 <2.4.5 || >=2.5.2 <3.0.0",
      "til" : "~1.2",
      "elf" : "~1.2.3",
      "two" : "2.x",
      "thr" : "3.3.x"
   }
}
```

Why semantic versioning is important
Suppose you use the awesome jQuery library in your component and you specify ~2.0.0 as a dependency, then you want to rely on the fact that the jQuery team doesn't release a fix or functionality in version 2.1.7 that breaks backwards compatibility, because your code could break.

The same goes for components that you maintain and may be used by others. Other developers expect you to know what you're doing and apply your versions correctly because their software depends on your code and their software could break when you apply a wrong version.

Conclusion
Development has changed a lot over the years. Open source initiatives have become so prevalent that code often depends on components written by developers from all over the world.

To keep things organized and manageable, it is important that every component is well documented and correctly versioned.

You can only imagine the impact if a widely used component would undergo a change where the code is no longer backwards compatible and it's version is not updated correctly.

I hope that by reading this article you now have a better understanding of what a version really means, what impact it has and why it is so important to update it correctly, especially if you write components that are used by others.

Who knows, one day the world may depend on one of your components...

Have a great one!

Available: https://www.jvandemo.com/a-simple-guide-to-semantic-versioning/
Accessed: 7/25/2020 5:33 PM