¿Qué es el versionamiento semántico y por qué es importante?

Cuando decides utilizar un código que no es tuyo como puede ser un framework o una librería, una de las cosas de las que tienes que preocuparte es de estar al día con las últimas versiones que vayan sacando. No solo porque quizá incluyan cosas que te puedan interesar, sino porque puede que la última versión del framework en el que acabas de basar toda tu aplicación no sea compatible con código de versiones anteriores.

Aquí nacen dos problemas del mundo del software. Uno, para el consumidor de ese código, que necesita una forma de saber si la última versión de una herramienta es un cambio absoluto del comportamiento o solo arregla unos fallos menores. Y dos, para el creador de código, que necesita de una manera de comunicar a sus usuarios la naturaleza y el alcance de la última versión que va a publicar.

¿Cómo se marca una versión como major, minor o patch?

Cada vez que enviemos código al repositorio crearemos un nuevo tag siguiendo el convenio semántico, dependiendo de los cambios introducidos. El tag contendrá la información, separando las versiones de cada tipo por puntos, de la formamajor.minor.patch. Esta versión en concreto, el tag que lo representa, no puede ser cambiado jamás, para que si alguien depende de esa versión en particular, pueda seguir haciéndolo sin problemas.

Si el framework o librería que yo utilizo sigue el versionamiento semántico a rajatabla, sé que puedo actualizar sin miedo todas las versiones marcadas como patches o minor porque no romperán mi código. Yo podría actualizar de la versión 5.3.3 a la 5.3.4 o incluso 5.4.0, porque solo estarían incrementando las versiones minor y patch, que son compatibles con código existente.

Si por el contrario pasase de la versión 1.4 a la versión 2.0, debería tener cuidado porque seguramente habría muchas cosas que dejarían de funcionar: ha cambiado la major.

Especificación del Versionado Semántico (SemVer)

Las palabras clave “DEBE”, “NO DEBE”, “OBLIGATORIO”, “DEBERÁ”, “NO DEBERÁ”, “DEBERÍA”, “NO DEBERÍA”, “RECOMENDADO”, “PUEDE” y “OPCIONAL” en este documento serán interpretadas como se describe en RFC 2119-es.

1. El Software que usa Versionado Semántico DEBE declarar un API público. Este API puede ser declarado en el propio código o debe existir estrictamente en la documentación. Sin importar como sea realizado, este DEBERÍA ser preciso y comprehensivo.
2. Un número de versión normal DEBE tener la forma de X.Y.Z donde X, Y y Z son números enteros no negativos, y NO DEBEN ser precedidos de ceros. X es la versión mayor, Y es la versión menor, y Z es la versión parche. Cada elemento DEBE incrementarse numéricamente. Por ejemplo: 1.9.0 -> 1.10.0 -> 1.11.0
3. Una vez que el paquete versionado ha sido publicado, el contenido de esa versión NO DEBE ser modificado. Cualquier modificación DEBE ser publicada como una nueva versión.
4. Una versión mayor en cero (0.y.z) se considera como desarrollo inicial. Todo PUEDE cambiar en cualquier momento. El API público NO DEBERÍA ser considerado estable.
5. La versión 1.0.0 define el API público. La manera en que cada número de versión es incrementado después de esta publicación dependerá de su API público y cómo cambia.

|  |  |
| --- | --- |
| La versión parche Z (x.y.Z | x > 0) DEBE ser incrementada si solamente se introducen correcciones de errores compatibles con versiones anteriores. Una corrección de error se define como un cambio interno que corrige un comportamiento incorrecto. |

|  |  |
| --- | --- |
| La versión menor Y (x.Y.z | x > 0) DEBE ser incrementada si se introduce funcionalidad nueva y compatible con la versión anterior del API público. Ésta DEBE ser incrementada si se introduce cualquier funcionalidad al API público o mejora al código privado. Este PUEDE incluir cambios a nivel de parches. La versión parche DEBE reiniciarse a 0 cuando una versión menor se incrementa. |

|  |  |
| --- | --- |
| La versión mayor (X.y.z | X > 0) DEBE ser incrementada solamente si se introducen cambios incompatibles con la versión anterior del API público. Este PUEDE incluir cambios de nivel menor y parches. Versiones parche y menores DEBEN ser reiniciadas a 0 cuando una versión mayor es incrementada. |

1. Una versión de prelanzamiento PUEDE ser denotada agregando un guion y una serie de identificadores separados por puntos, inmediatamente seguida de la versión parche. Los identificadores DEBEN ser compuestos sólo de caracteres alfanuméricos ASCII y guion ([0-9A-Za-z-]). Los identificadores NO DEBEN estar vacíos. Identificadores numéricos NO DEBEN ser precedidos de cero. Versiones de prelanzamiento tienen una precedencia inferior que una versión normal. Una versión de prelanzamiento indica que esa versión no es estable y puede no satisfacer los requerimientos de compatibilidad destinados como se denota en la versión normal asociada. Por ejemplo: 1.0.0-alpha, 1.0.0-alpha.1, 1.0.0-0.3.7, 1.0.0-x.7.z.92.
2. Metadatos de compilación PUEDEN ser denotados agregando el signo más y una serie de identificadores separados por puntos, inmediatamente seguido de la versión parche o prelanzamiento. Los identificadores DEBEN ser compuestos de sólo caracteres alfanuméricos ASCII y guión ([0-9A-Za-z-]). Los identificadores NO DEBEN estar vacíos. Identificadores numéricos NO DEBEN ser precedidos de cero. Los metadatos de compilación DEBEN ser ignorados cuando se determina la precedencia de la versión. Por lo tanto, dos versiones que difieren solamente en los metadatos de compilación tienen la misma precedencia. Por ejemplo: 1.0.0-alpha+001, 1.0.0+20130313144700, 1.0.0-beta+exp.sha.5114f85.
3. La precedencia se refiere a cómo las versiones se comparan entre ellas cuando se ordenan.
   1. La precedencia DEBE ser calculada separando los identificadores de la versión en mayor, menor, parche y prelanzamiento (dejando de lado los metadatos de compilación).
   2. La precedencia es determinada por la primera diferencia al comparar cada uno de los identificadores de izquierda a derecha como se indica: mayor, menor y parche son siempre comparadas numéricamente.

Por ejemplo: 1.0.0 < 2.0.0 < 2.1.0 < 2.1.1.

* 1. Cuando la versión mayor, menor y parche son iguales, la versión de prelanzamiento tiene menor precedencia que la versión normal.

Por ejemplo: 1.0.0-alpha < 1.0.0.

* 1. La precedencia para dos versiones de prelanzamiento con la misma versión mayor, menor y parche DEBE ser determinada comparando cada identificador separado por punto, de izquierda a derecha, hasta que una diferencia sea encontrada como se indica:
     1. identificadores compuestos solamente por números son comparados numéricamente
     2. los identificadores con letras o guiones son comparados léxicamente en orden ASCII.
     3. Identificadores numéricos siempre tienen menor precedencia que identificadores no numéricos.
     4. Un conjunto de campos de prelanzamiento más numeroso tiene mayor precedencia que un conjunto menos numeroso, si todos los identificadores anteriores son iguales.

Por ejemplo: 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.