Semantic Versioning 2.0.0

Zusammenfassung

Auf Grundlage einer Versionsnummer von MAJOR.MINOR.PATCH werden die einzelnen Elemente folgendermaßen erhöht:

1. MAJOR wird erhöht, wenn API-inkompatible Änderungen veröffentlicht werden,
2. MINOR wird erhöht, wenn neue Funktionalitäten, welche kompatibel zur bisherigen API sind, veröffentlicht werden, und
3. PATCH wird erhöht, wenn die Änderungen ausschließlich API-kompatible Bugfixes umfassen.

Außerdem sind Bezeichner für Vorveröffentlichungen und Build-Metadaten als Erweiterungen zum MAJOR.MINOR.PATCH Format verfügbar.

Einführung

In der Welt der Softwareentwicklung existiert ein grauenhafter Ort namens “Dependency Hell”. Um so größer ein Projekt wird und je mehr Pakete in die Software integriert werden, desto wahrscheinlicher ist es, dass dieser fürchterliche Ort eines Tages betreten wird.

In Projekten mit vielen Abhängigkeiten kann das Aktualisieren abhängiger Pakete schnell zum Albtraum werden. Sind die Abhängigkeitsspezifikationen des Pakets zu strikt, besteht die Gefahr des “Version Lock” (die Unfähigkeit ein Paket zu aktualisieren, ohne, dass alle abhängigen Pakete dieses Pakets ebenfalls aktualisiert werden müssen). Wenn die Abhängigkeiten des Pakets allerdings zu lasch definiert sind, wird sehr wahrscheinlich ein Problem, das sich “Version Promiscuity” nennt (das Paket gibt vor, mit mehr zukünftigen Versionen seiner abhängigen Pakete kompatibel zu sein, als angemessen ist), eintreten. *Dependency Hell* bezeichnet die Situation, in der entweder *Version Lock* oder *Version Promiscuity*, oder beides den Entwicklungsprozess des Projekts beeinträchtigt.

Als Lösung für dieses Problem schlage ich ein einfaches Regelwerk vor, welches definiert wie Versionsnummern gewählt und erhöht werden. Diese Regeln basieren auf bereits existierenden und weit verbreiteten Verfahren, welche sowohl bei der Entwicklung von Closed- als auch von Open-Source Software verwendet werden, aber beschränken sich nicht zwingend auf diese. Um dieses System nutzen zu können, muss zuerst eine öffentliche API definiert werden. Diese kann entweder in Form einer Dokumentation existieren oder durch den Code selbst erzwungen werden. Egal auf welche Art und Weise die API umgesetzt wird, es ist wichtig, dass sie übersichtlich und präzise ist. Sobald die öffentliche API erstellt wurde, werden Änderungen an dieser durch bestimmten Veränderungen an der Versionsnummer vermittelt. Nimm ein Versionsnummernformat von X.Y.Z (Major.Minor.Patch) an. Bei Einführung von Bugfixes, welche die öffentliche API nicht beeinflussen, wird die Patch Version erhöht, API-kompatible Ergänzungen oder Änderungen erhöhen die Minor Versionsnummer, und Änderungen, welche nicht kompatibel zur aktuellen öffentlichen API sind, erhöhen die Major Version.

Ich nenne dieses System “Semantic Versioning”. Versionsnummern, die nach diesem Schema gewählt und erhöht werden, geben direkten Aufschluss über den entsprechenden Code und was sich von einer zur anderen Version verändert hat.

Semantic Versioning Spezifikation (SemVer)

Die Terme “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, und “OPTIONAL” in diesem Dokument sind, wie in RFC 2119 beschrieben, zu interpretieren.

1. Software, welche *Semantic Versioning* nutzt, muss (MUST) eine öffentliche API definieren. Die API kann entweder im Code selbst deklariert werden oder in einer Dokumentation enthalten sein. Wie auch immer sie umgesetzt wird, es ist wichtig, dass sie präzise und ausführlich ist.

2. Eine gewöhnliche Versionsnummer muss (MUST) dem Format X.Y.Z entsprechen, wobei X, Y und Z Ganzzahlen größer oder gleich Null sind und eine Zahl größer als Null keine führenden Nullen enthalten darf. X ist die Major Version, Y ist die Minor Version und Z ist die Patch Version. Jedes Element muss (MUST) auf numerische Art und Weise erhöht werden. Zum Beispiel: 1.9.0 -> 1.10.0 -> 1.11.0.

3. Sobald eine Version eines Projektes veröffentlicht wurde, darf (MUST NOT) der Inhalt dieser Version nicht mehr verändert werden. Eine Änderung am Inhalt muss (MUST) als eine neue Version veröffentlicht werden.

4. Versionsnummern mit einer Major Version von 0 (0.y.z) sind für die initiale Development Phase gedacht. Änderungen können in jeder denkbaren Form und zu jeder Zeit auftreten. Die öffentliche API sollte nicht als *stable* betrachtet werden.

5. Die Version 1.0.0 definiert die öffentliche API. Ab dieser Veröffentlichung hängt die Art und Weise, wie die Versionsnummer erhöht und verändert wird, von der öffentlichen API und den Änderungen, die an ihr vollzogen werden, ab.

6. Die Patch Version Z (x.y.Z | x > 0) muss (MUST) erhöht werden, wenn ausschließlich API-kompatible Bugfixes eingeführt werden. Ein Bugfix ist als eine interne Änderung, welche ein fehlerhaftes Verhalten korrigiert, definiert.

7. Die Minor Version Y (x.Y.z | x > 0) muss (MUST) erhöht werden, wenn neue Funktionalitäten, welche kompatibel zur bisherigen API sind, veröffentlicht werden. Sie muss (MUST) außerdem erhöht werden, wenn eine Funktion der öffentlichen API als *deprecated* markiert wird. Wenn umfangreiche Änderungen an internem Code eingeführt werden, darf (MAY) die Minor Version ebenfalls erhöht werden. Wenn diese Versionsnummer erhöht wird, muss (MUST) die Patch Version auf Null zurückgesetzt werden.

8. Die Major Version X (X.y.z | X > 0) muss (MUST) immer dann erhöht werden, wenn API-inkompatible Änderungen in die öffentlichen API eingeführt werden. Die Änderungen dürfen (MAY) auch Änderungen umfassen, die ansonsten die Minor Version oder die Patch Version erhöht hätten. Wenn diese Versionsnummer erhöht wird, muss (MUST) sowohl die Minor Version als auch die Patch Version auf Null zurückgesetzt werden.

9. Eine Vorveröffentlichung kann (MAY) gekennzeichnet werden, indem ein Bindestrich, gefolgt von dem Vorveröffentlichungs-Bezeichner, dessen Elemente durch Punkte voneinander getrennt werden, an die Patch Version angehängt wird. Die Elemente des Bezeichners dürfen (MUST) nur aus alphanumerischen ASCII Zeichen und dem Bindestrich ([0-9A-Za-z-]) bestehen. Sie dürfen (MUST NOT) außerdem nicht leer sein. Wenn ein Element ausschließlich aus Ziffern besteht, darf (MUST NOT) es keine führenden Nullen enthalten. Eine Vorveröffentlichungs-Version hat einen niedrigeren Rang als die entsprechende reguläre Version. Ein Vorveröffentlichungs-Bezeichner kennzeichnet, dass die Version als *unstable* zu betrachten ist und dass sie unter Umständen nicht den Kompatibilitätsanforderungen, die für die entsprechende regulären Version bestimmt wurden, entspricht. Beispiele: 1.0.0-alpha, 1.0.0-alpha.1, 1.0.0-0.3.7, 1.0.0-x.7.z.92.

10. Build-Metadaten können (MAY) ausgezeichnet werden, indem ein Plus Symbol, gefolgt von den Metadaten, deren Elemente durch Punkte voneinander getrennt werden, an die Patch Version oder den Vorveröffentlichungs-Bezeichner angehängt wird. Die Elemente der Metadaten dürfen (MUST) nur aus alphanumerischen ASCII Zeichen und dem Bindestrich ([0-9A-Za-z-]) bestehen. Sie dürfen (MUST NOT) außerdem nicht leer sein. Die Build-Metadaten haben keinerlei Einfluss auf den Rang einer Version, sodass zwei Versionen, deren Versionsnummern sich nur in den Build-Metadaten unterscheiden, denselben Rang einnehmen. Beispiele: 1.0.0-alpha+001, 1.0.0+20130313144700, 1.0.0-beta+exp.sha.5114f85.

11. Der Rang einer Version bestimmt, welche Versionsnummer einer anderen übergeordnet ist, wenn diese bei einer Sortierung miteinander verglichen werden. Der Rang wird (MUST) aus der Major, Minor und Patch Version sowie dem Vorveröffentlichungs-Bezeichner berechnet (Die Build-Metadaten haben keinerlei Einfluss auf den Rang einer Version). Er wird bestimmt, indem der erste Unterschied zwischen dem oben aufgeführten Elementen ermittelt wird. Dabei wird von links nach rechts, in der oben genannten Reihenfolge vorgegangen. Die Major, Minor und Patch Versionen werden numerisch miteinander verglichen. Beispiel: 1.0.0 < 2.0.0 < 2.1.0 < 2.1.1. Beim Vergleich von zwei Versionsnummern, deren Major, Minor und Patch Versionen gleich sind, nimmt eine Vorveröffentlichung einen niedrigeren Rang als die reguläre Version ein. Beispiel: 1.0.0-alpha < 1.0.0. Sind beide dieser Versionen Vorveröffentlichungen, wird (MUST) der Rang

ermittelt, indem jedes Element eines Vorveröffentlichungs-Bezeichners (durch Punkte voneinander getrennt) mit dem der anderen Version verglichen wird bis ein Unterschied festgestellt wird. Auch hierbei wird von links nach rechts vorgegangen. Elemente, welche ausschließlich aus Ziffern bestehen, werden numerisch miteinander verglichen. Der Vergleich aller anderen Elemente erfolgt auf Basis der ASCII-Stellenwerte ihrer Zeichen. Numerische Elemente haben immer einen niedrigeren Rang als solche, die auch andere Zeichen enthalten. Falls alle Elemente identisch sind, nimmt der Bezeichner mit den meisten Elementen den höheren Rang ein. Beispiel: 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 Grammatik für valide SemVer Versionen

```
<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-sepa

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

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

<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 char

<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"
```

Weshalb sollte man Semantic Versioning nutzen?

Dieses System ist keine neue oder revolutionäre Idee. Tatsächlich hast du wahrscheinlich bereits ein ähnliches System genutzt. Das Problem ist, dass “ähnlich” nicht ausreichend ist. Ohne Einhaltung irgendeiner Art von offizieller Spezifikation, sind Versionsnummern nicht besonders hilfreich beim Verwalten von Abhängigkeiten. Durch das Benennen und Aufstellen von Regeln für die, am Anfang dieses Dokuments angesprochenen, Ideen, wird es einfach, Nutzer der Software über die Art und den Umfang der Änderungen zu informieren. Sobald diese Informationen ersichtlich sind, können endlich flexible (aber nicht zu flexible) Abhängigkeitsangaben gemacht werden werden.

Wie *Semantic Versioning* die *Dependency Hell* ein Problem der Vergangenheit werden lassen kann, wird hier an einem einfachen Beispiel veranschaulicht. Geh von einer Library namens “Firetruck” aus. Sie ist abhängig von einem *Semantically versioned* Paket namens “Ladder”. Zum Zeitpunkt der Erstellung von Firetruck befindet sich Ladder in Version 3.1.0. Da Firetruck eine Funktion von Ladder verwendet, welche erst ab Version 3.1.0 verfügbar ist, kann die Abhängigkeit als *größer oder gleich 3.1.0, aber kleiner als 4.0.0* definiert werden. Wenn also jetzt Version 3.1.1 oder 3.2.0 von Ladder veröffentlicht wird, kann diese einfach im Package Management System freigeschaltet werden, mit der Gewissheit, dass sie mit der abhängigen Software (Firetruck) kompatibel ist.

Als ein verantwortungsbewusster Programmierer will man natürlich prüfen, ob die aktualisierten Pakete wie beschrieben funktionieren. Die richtige Welt ist ein chaotischer Ort; Es gibt nichts, was wir tun können um dies sicherstellen, aber sei wachsam! Was wir tun können ist, *Semantic Versioning* zu nutzen um auf eine angemessen einfache Art und Weise Pakete zu aktualisieren ohne auch deren abhängige Pakete aktualisieren zu müssen und damit Zeit und Aufwand zu sparen.

Wenn das alles wünschenswert klingt und von den Vorteilen profitiert werden soll, muss nichts Weiteres getan werden, als anzugeben, dass ein Projekt *Semantic Versioning* verwendet und anschließend den oben genannte Regeln Folge zu leisten. Verweise in der README auf diese Webseite, sodass auch andere über die Regeln Bescheid wissen und von ihnen profitieren können.

FAQ

Wie soll ich bei der Versionierung in der initialen Development Phase (0.y.z) verfahren?

Das Einfachste ist, die Versionierung bei 0.1.0 zu beginnen und dann bei jedem folgender Veröffentlichung die Minor Version zu erhöhen.

Woher weiß ich, wann es Zeit ist Version 1.0.0 zu veröffentlichen?

Wenn die Software schon in der Produktion verwendet wird, sollte sie bereits in Version 1.0.0 vorliegen. Falls eine *stable* API existiert, auf die sich Nutzer bereits verlassen, sollte es ebenfalls die Version 1.0.0 sein. Auch wenn Kompatibilität zu vorherigen Versionen bereits eine wichtige Rolle spielt, ist Version 1.0.0 angebracht.

Hält das nicht von *Rapid Development* und *Fast Iteration* ab?

In Versionen mit einer Major Version von Null dreht sich alles um *Rapid Development*. Wenn sich die API tagtäglich verändert, sollte sich das Projekt entweder noch in Version 0.y.z befinden oder es sollte auf einem separate Development Branch an der nächsten Major Veröffentlichung gearbeitet werden.

Wenn schon die kleinsten API-inkompatiblen Änderungen an der öffentlichen API eine Anhebung der Major Version erfordern, wird eine Version wie 42.0.0 nicht sehr schnell erreicht

werden?

Das ist eine Frage von verantwortungsbewusstem Development und Weitsicht. API-inkompatible Änderungen sollten nicht leichtfertig eingeführt werden, da das Aktualisieren des Pakets in Software von Dritten, welche eine große Menge an API-spezifischen Code enthalten, mit drastischem Aufwand verbunden sein kann. Die Tatsache, dass die Major Version beim Einführen von API-inkompatiblen Änderungen angehoben werden muss, drängt einen dazu, die Auswirkungen der Änderungen noch einmal zu überdenken und das Kosten-Nutzen-Verhältnis abzuwägen.

Die gesamte öffentliche API zu dokumentieren ist viel zu viel Arbeit!

Es ist die Aufgabe eines professionellen Developers, Software, welche für die Verwendung durch andere bestimmt ist, ordentlich zu dokumentieren. Das Verwalten der Komplexität einer Software ist ein enorm wichtiger Teil, wenn es darum geht ein Projekt erfolgreich zu betreiben, was schwer ist, wenn niemand weiß wie eine Software zu verwenden ist oder welche Funktionen sie anbietet. Langfristig gesehen, werden *Semantic Versioning* und eine gut definierte öffentliche API ein System sicherstellen, in dem alles reibungslos ineinandergreift.

Was soll ich tun wenn ich versehentlich eine API-inkompatible Änderung in einer Minor Version veröffentlicht habe?

Sobald du feststellst, dass du die *Semantic Versioning* Spezifikation nicht befolgt hast, korrigiere den Fehler, indem du eine neue Minor Version veröffentlichst, welche das Problem behebt und die Kompatibilität zur API wiederherstellt. Selbst unter diesen Umständen ist es nicht erlaubt, eine bereits veröffentlichte Version zu verändern. Falls es angemessen ist, dokumentiere welche Version problematisch ist, sodass die Nutzer über diese Version Bescheid wissen.

Was soll ich tun wenn ich die Abhängigkeitsangaben meines Projekts ändere, aber keine Änderungen an der öffentlichen API einführe?

Dies würde als kompatibel angesehen werden, da es die öffentliche API nicht beeinflusst. Eine Software, welche ausdrücklich dieselben Abhängigkeiten wie das Paket hat, sollte seine eigenen Abhängigkeitsangaben haben und der Autor der Software wird mögliche Konflikte selbstständig feststellen. Ob nun die Minor Version oder aber die Patch Version erhöht wird, hängt davon ab ob die Abhängigkeiten aktualisiert wurden um einen

Bug zu beseitigen oder um eine neue Funktionalität zu ergänzen. Normalerweise geschieht dies aus letzterem Grund, bei dem natürlich die Minor Version angehoben werden müsste.

Was soll ich tun wenn ich die öffentliche API versehentlich derartig verändert habe, dass sie nicht mit der Änderung an der Versionsnummer harmoniert (Das heißt, der Code zerstört fälschlicherweise in einer Patch Version die API-Konformität)?

Entscheide nach deinem eigene Ermessen. Wenn du eine große Nutzergemeinde hast, die von der aktuellen API stark abhängt, dann wäre es wahrscheinlich das Beste, die Veröffentlichung als eine Major Version zu publizieren, auch wenn die Änderungen eigentlich nur einen Patch darstellen sollten. Denk dran, bei *Semantic Versioning* dreht sich alles darum, die Art und den Umfang der Änderungen am Code durch die Änderungen an der Versionsnummer zu vermitteln.

Wie soll ich mit *deprecated* Funktionen verfahren?

Funktionalitäten als *deprecated* zu markieren ist ein gewöhnlicher Teil von Software Development und ist häufig notwendig um mit der Entwicklung voranzuschreiten. Wenn etwas in der öffentlichen API als *deprecated* markiert wird, sollte erstens, die Dokumentation bezüglich der Änderungen angepasst werden, und zweitens, eine neue Minor Version mit der *deprecated* Funktionalität veröffentlicht werden. Bevor die Funktionalität in einer Major Veröffentlichung vollständig entfernt wird, sollte mindestens eine Minor Version, die die *Deprecation* enthält, veröffentlicht werden, sodass Nutzer einfach zur neuen API migrieren können.

Ist die Länge eines SemVer Version Strings limitiert?

Nein, aber sei vernünftig. Zum Beispiel, wäre ein 255 Zeichen langer Version String wahrscheinlich ein wenig übertrieben. Außerdem könnten bestimmte Systeme ihre eigenen Limits definieren.

Über

Die *Semantic Versioning* Spezifikation wurde von Tom Preston-Werner, Erfinder von Gravatars und Mitbegründer von GitHub, erstellt.

Für Feedback, eröffne bitte ein Issue auf GitHub.

Lizenz