An Empirical Study of Web API Versioning Practices

Souhaila Serbout
@souhailaserbout
souhaila.serbout@usi.ch
(Speaker)

Cesare Pautasso
@pautasso@scholar.social
cesare.pautasso@usi.ch
MongoDB module dependencies

Popularity score: 🔴 = 0%, 🔴 = 50%, 🔴 = 100%
(Dev dependencies are excluded from the graph)
Popularity score:🔴 = 0%,🟡 = 50%,🟢 = 100%

(Dev dependencies are included in the graph)
Semantic versioning

- Incremented when backward incompatible changes are introduced
- Incremented if only backwards compatible bug fixes are introduced
- Incremented if new, backwards compatible functionality is introduced

Major.Minor.Patch [-Pre]

[1] https://semver.org/
# Incrementing semantic versions in published packages

To help developers who rely on your code, we recommend starting your package version at `1.0.0` and incrementing as follows:

<table>
<thead>
<tr>
<th>Code status</th>
<th>Stage</th>
<th>Rule</th>
<th>Example version</th>
</tr>
</thead>
<tbody>
<tr>
<td>First release</td>
<td>New product</td>
<td>Start with 1.0.0</td>
<td>1.0.0</td>
</tr>
<tr>
<td>Backward compatible bug fixes</td>
<td>Patch release</td>
<td>Increment the third digit</td>
<td>1.0.1</td>
</tr>
<tr>
<td>Backward compatible new features</td>
<td>Minor release</td>
<td>Increment the middle digit and reset last digit to zero</td>
<td>1.1.0</td>
</tr>
<tr>
<td>Changes that break backward compatibility</td>
<td>Major release</td>
<td>Increment the first digit and reset middle and last digits to zero</td>
<td>2.0.0</td>
</tr>
</tbody>
</table>

[2] https://docs.npmjs.com/about-semantic-versioning
# Incrementing semantic versions in published packages

To help developers who rely on your code, we recommend starting your package version at `1.0.0` and incrementing as follows:

<table>
<thead>
<tr>
<th>Code status</th>
<th>Stage</th>
<th>Rule</th>
<th>Example version</th>
</tr>
</thead>
<tbody>
<tr>
<td>First release</td>
<td>New product</td>
<td>Start with 1.0.0</td>
<td>1.0.0</td>
</tr>
<tr>
<td>Backward compatible bug fixes</td>
<td>Patch release</td>
<td>Increment the third digit</td>
<td>1.0.1</td>
</tr>
<tr>
<td>Backward compatible new features</td>
<td>Minor release</td>
<td>Increment the middle digit and reset last digit to zero</td>
<td>1.1.0</td>
</tr>
<tr>
<td>Changes that break backward compatibility</td>
<td>Major release</td>
<td>Increment the first digit and reset middle and last digits to zero</td>
<td>2.0.0</td>
</tr>
</tbody>
</table>

[2] https://docs.npmjs.com/about-semantic-versioning
1.7.0 introduced loads of breaking changes.

The number of dependant modules which are now broken as a result is huge, personally I think that 1.7.0 should be killed (removed from npm) and 2.0 released - the longer the delay the harder it will be to do this.

underscore.js is solely consumed via package managers that mandate the use of semver, you may personally not like semver but that is what is used by the installers to determine compatibility. Last time this was brought up you stated that if you used semver then we would be on underscore version 47 now - well that is much better than having broken code everywhere and lodash has managed to keep the version number below 4.0.0 without breaking everyone's code.

Semantic versioning and impact of breaking changes in the Maven repository

S. Raemakers1,*, A. van Deursen1,†, J. Visser2
1Vrije Universiteit, Amsterdam, The Netherlands
2Technical University Delft, The Netherlands

SUMMARY

Versioning systems that depend on third-party libraries may have to be updated when updates to those libraries become available in order to benefit from new functionality, security patches, bug fixes, or API improvements. However, such changes come with changes to the existing interfaces of those libraries, possibly causing breakage on the client side. In this paper, we investigate versioning practices in a set of more than 30 000 jar files from Maven Central, spanning over 7 years of history at more than 2 000 different libraries. We investigate to what degree versioning conventions are followed in this repository. Semantic versioning provides strict rules regarding major breaking changes allowed, minor (no breaking changes allowed), and patch releases (only backward-compatible bug fixes allowed). We find that around one third of all releases introduce at least one breaking change. We perform an empirical study on potential relevance caused by breaking changes in libraries and find that breaking changes have a significant impact on client libraries using the changed functionality. We find out that minor releases generally have larger release intervals than minor releases. We also investigate the use of deprecation tags and find out that these tags are applied improperly in our dataset.

1 Introduction

For users of software libraries or application programming interfaces (APIs), backward compatibility is a desirable trait. Without backward compatibility, library users will face increased risk and cost when upgrading their dependencies. In spite of these costs and risks, library upgrades may be desirable or even necessary, for example if the newer version contains required additional functionality or critical security flaws. To conduct the upgrade, the library user will need to know whether there are incompatibilities, and, if so, which ones.

Determining whether there are incompatibilities, however, is hard to do for the library user (if not, in fact, undesirable in general). Therefore, it is the library creator’s responsibility to indicate the level of compatibility of a library update. One way to inform library users about incompatibilities is through version numbers. As an example, semantic versioning1 (semantic) suggests a versioning scheme in which three digit version numbers MAJOR.MINOR.PATCH have the following semantics:

• MAJOR: This number should be incremented when incompatible API changes are made;
• MINOR: This number should be incremented when functionality is added in a backward-compatible manner;
• PATCH: This number should be incremented when backward-compatible bug fixes are made.

As an approximation of the (undesirable) notion of backward compatibility, we use the concept of a binary compatibility as defined in the Java language specification. The Java Language Specification2 states that a change to a type is binary compatible with (equivalently, does not break binary compatibility with) pre-existing libraries if pre-existing libraries that previously failed without error will continue to link without error. This is an underestimation, since binary incompatibilities are certainly breaking, but there are likely to be different (semantic) incompatibilities as well. For the purpose of this paper, we define any change that does not maintain binary compatibility between releases to be a breaking change.

1 http://semver.org.

© 2016 Elsevier Inc. All rights reserved.
Semantic versioning and impact of breaking changes in the Maven repository

S. Raemakers, A. van Deursen, J. Visser

1NICTA, Hawthorn, Victoria, Australia 2University of New South Wales, Australia 3Technische Universiteit Delft, The Netherlands 4National ICT Australia Limited

ARTICLE INFO

Article history:
Received 20 February 2015
Revised 17 February 2015
Accepted 6 April 2016
Available online 27 April 2016

Keywords:
Semantic versioning
Breaking changes
Software libraries

ABSTRACT

Systems that depend on third-party libraries may have to be updated when updates to those libraries become available in order to benefit from new functionality, security patches, bug fixes, or API improvements. However, these changes come with changes to the existing interfaces of these libraries, possibly causing brokenness on the client systems. In this paper, we investigate versioning practices in a set of more than 108,000 jar files from Maven Central, spanning over 7 years of history of more than 22,000 different libraries. We investigate to what degree versioning conventions are followed in this repository, semantic versioning on the one hand, and major and minor breaking changes allowed, and patch releases only backward-compatible bug fixes allowed. We find that around one third of all releases introduce at least one breaking change. We perform an empirical study on potential brokenness caused by breaking changes in library releases and find that breaking changes have a significant impact on client libraries using the changed functionality. We find that minor releases generally have larger release intervals than major releases. We also investigate the usage of deprecation tags and find out that these tags are applied improperly in our dataset.

© 2016 Elsevier Inc. All rights reserved.

1. Introduction

For users of software libraries or application programming interfaces (APIs), backward compatibility is a desirable trait. Without backward compatibility, library users will face increased risk and cost when upgrading their dependencies. In spite of these costs and risks, library upgrades may be desirable or even necessary, for example if the newer version contains required additional functionality or critical security fixes. To conduct the upgrade, the library user will need to know whether there are incompatibilities, and, if so, which ones.

Determining whether there are incompatibilities, however, is hard to do for the library user (it is, in fact, undesirable in general). Therefore, it is the library creator’s responsibility to indicate the level of compatibility of a library update. One way to inform library users about incompatibilities is through version numbers. As an example, semantic versioning1 (semver) suggests a versioning scheme in which three digit version numbers MAJOR.MINOR.PATCH have the following semantics:

MAJOR This number should be incremented when incompatible API changes are made;
MINOR This number should be incremented when backward compatibility is added in a backward-compatible manner;
PATCH This number should be incremented when backward-compatible bug fixes are made.

As an approximation of the (irreducible) notion of backward compatibility, we use the concept of a binary compatibility as defined in the Java language specification. The Java Language Specification2 states that a change to a type or binary compatible with (equivalently, does not break binary compatibility with) pre-existing libraries if all pre-existing libraries that previously linked without error will continue to link without error. This is an underestimation, since binary incompatibilities are certainly breaking, but there are likely to be different (semantic) incompatibilities as well. For the purpose of this paper, we define any change that does not maintain binary compatibility between releases to be a breaking change.

Maven in 2017

Breaking changes (BCs) were spread over all the software releases: Major (35.8%), Minor (35.7%), Patch (23.8%).

1 Corresponding author at: Technical University Delft, Delft, The Netherlands. Tel: +31622527293;
E-mail addresses: stevenraemakers@gmail.com (S. Raemakers), a.vandeursen@gmail.com (A. van Deursen), j.visser@gmail.com (J. Visser).
Breaking changes (BCs) were spread over all the software releases: Major (35.8%), Minor (35.7%), Patch (23.8%).

Slight increased adherence to SemVer in Maven Repositories Over the years [4].
Breaking bad? Semantic versioning and impact of breaking changes in Maven Central

An external and differentiated replication study

Lina Ochoa¹ · Thomas Degueule² · Jean-Rémy Falleri²,³ · Jurgen Vinju¹,⁴

Accepted: 30 August 2021 / Published online: 17 March 2022
© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2022

Abstract
Just like any software, libraries evolve to incorporate new features, bug fixes, security patches, and refactorings. However, when a library evolves, it may break the contract previously established with its clients by introducing Breaking Changes (BCs) in its API. These changes might trigger compile-time, link-time, or run-time errors in client code. As a result, clients may hesitate to upgrade their dependencies, raising security concerns and making future upgrades even more difficult. Understanding how libraries evolve helps client developers to know which changes to expect and where to expect them, and library developers to understand how they might impact their clients. In the most extensive study to date, Rae-maekers et al. investigate to what extent developers of Java libraries hosted on the Maven Central Repository (MCR) follow semantic versioning conventions to signal the introduction of BCs and how these changes impact client projects. Their results suggest that BCs are widespread without regard for semantic versioning, with a significant impact on clients. In this paper, we conduct an external and differentiated replication study of their work. We identify and address some limitations of the original protocol and expand the analysis to a new corpus spanning seven more years of the MCR. We also present a novel static analysis tool for Java bytecode, Maracas, which provides us with: (i) the set of all BCs between two versions of a library, and; (ii) the set of locations in client code impacted by individual BCs. Our key findings, derived from the analysis of 119,879 library upgrades and 293,817 clients, contrast with the original study and show that 83.4% of these upgrades do comply with semantic versioning. Furthermore, we observe that the tendency to comply with semantic versioning has significantly increased over time. Finally, we find that most BCs affect code that is not used by any client, and that only 7.9% of all clients are affected by BCs. These findings should help (i) library developers to understand and anticipate the impact of their changes; (ii) library users to estimate library upgrading effort and to pick libraries that are less likely to break, and; (iii) researchers to better understand the dynamics of library-client co-evolution in Java.
Breaking bad? Semantic versioning and impact of breaking changes in Maven Central

An external and differentiated replication study

Lina Ochoa¹ · Thomas Degueule² · Jean-Rémy Falleri²,³ · Jurgen Vinju¹,⁴

Abstract

Just like any software, libraries evolve to incorporate new features, bug fixes, security patches, and refactorings. However, when a library evolves, it may break the contract previously established with its clients by introducing Breaking Changes (BCs) in its API. These changes might trigger compile-time, link-time, or run-time errors in client code. As a result, clients may hesitate to upgrade their dependencies, raising security concerns and making future upgrades even more difficult. Understanding how libraries evolve helps client developers to know which changes to expect and where to expect them, and library developers to understand how they might impact their clients. In the most extensive study to date, Rae-makers et al. investigate to what extent developers of Java libraries hosted on the Maven Central Repository (MCR) follow semantic versioning conventions to signal the introduction of BCs and how these changes impact client projects. Their results suggest that BCs are widespread without regard for semantic versioning, with a significant impact on clients. In this paper, we conduct an external and differentiated replication study of their work. We identify and address some limitations of the original protocol and expand the analysis to a new corpus spanning seven more years of the MCR. We also present a novel static analysis tool for Java bytecode, Maracas, which provides us with: (i) the set of all BCs between two versions of a library, and; (ii) the set of locations in client code impacted by individual BCs. Our key findings, derived from the analysis of 119,879 library upgrades and 293,817 clients, contrast with the original study and show that 83.4% of these upgrades do comply with semantic versioning, with a significant impact on clients. We observe that the tendency to comply with semantic versioning has significantly increased over time. Finally, we find that most BCs affect code that is not used by any client, and that only 7.9% of all clients are affected by BCs. These findings should help (i) library developers to understand and anticipate the impact of their changes; (ii) library users to estimate library upgrading effort and to pick libraries that are less likely to break, and; (iii) researchers to better understand the dynamics of library-client co-evolution in Java.

Maven in 2022

83.4% of all library upgrades comply with SemVer principles; Breaking changes were introduced only when they are expected [5].

Communicated by: Gabriele Bavota

Extended author information available on the last page of the article.
What about Web APIs?
What about Web APIs?

APIs are not centrally deployed
How to version web APIs?
Versioning in API design: What it is, and deciding which version of versioning is right for you

March 29, 2018

Martin Nally
Software Developer and API designer, Apigee

There’s a lot of advice on the web about API versioning, much of it contradictory and inconclusive: One expert says to put version identifiers in HTTP headers, another expert insists on version identifiers in URL paths, and a third says that versioning of APIs is not necessary at all. (For some examples of those divergent views, take a look at this post and its bibliography and this interview with the author of the RESTful design pattern.)

I am planning to write a RESTful API and I am clueless how to handle versioning. I have read many discussions and blog articles, which suggest to use the accept header for versioning.

But then I found following website listening popular REST APIs and their versioning method and most of them using the URL for versioning. Why?

Why are most people saying: "Don’t use the URL, but use the accept header", but popular APIs using URL?
You're thinking about API versioning in the wrong way.
And the way you have implemented versioning is not correct.
API Management

Versioning in API design: What it is, and deciding which version of versioning is right for you

March 29, 2018

Software Development

You're thinking about API versioning the wrong way.

And the way you have implemented versioning is not correct.

Xeno Fox
May 13, 2019 - 11 min read - Listen

REST versioning -

Now brace yourself, here is what you need to take away from reading this article.

“"You should not be versioning your API at all.”"

Now before you Alt+Tab away, you might as well hear me out, since you've read this far into the article. Versioning your API is not the correct way to resolve the problem you are facing. Instead you should be versioning your

Version in URL

The easiest way to handle multiple versions is to put the version number into the URL. You can find this approach for example in Twitter API.

http://myapplication.com/api/v1/user/1
http://myapplication.com/api/v2/user/1
Versioning in API design: What it is, and deciding which version of versioning is right for you

March 29, 2018

Now brace yourself, here is what you need to take away from reading this article.

“You should not be versioning your API at all.”

Your API versioning is wrong, which is why I decided to do it 3 different wrong ways

10 FEBRUARY 2014

In the end, I decided the fairest, most balanced way was to piss everyone off equally. Of course I’m talking about API versioning and not since the great "tabs versus spaces" debate have I seen so many strong beliefs in entirely different camps.

REST versioning -

Version in URL

The easiest way to handle multiple versions is to put the version number into the URL. You can find this approach for example in Twitter API.

http://myapplication.com/api/v1/user/1
http://myapplication.com/api/v2/user/1
Versioning in API design: What it is, and deciding which version of versioning is right for you

March 29, 2018

Now brace yourself, here is what you need to know in this article.

“You should not be versioning”

Now, traditionally, REST architectures have talked about versioning as something you should do. But I think an even better way to think about versioning is that it’s something you might as well be doing. Think of it as an API being “broken” over time. As you are adding features, you are breaking your API. Instead of being broken, you should be versioning.

Your API versioning is wrong, which is why I decided to do it 3 different wrong ways

In the end, I decided the fairest, most balanced way was to piss everyone off equally. Of course I’m talking about API versioning and not since the great "tabs versus spaces" debate have I seen so many strong beliefs in entirely different camps.

API versioning best practices: When you need versioning and when you don't

May 15, 2017

Martin Nally
Software Developer and API designer, Apigee

When versioning makes sense—and when it doesn’t
How to version web APIs?
How do developers version web APIs?
7,114 Web APIs
APIs with more than 10 commits
186,259 Commits
7,114 Web APIs
APIs with more than 10 commits

186,259 Commits

2015  2022
Xero OAuth 2 Identity Service API

https://api.xero.com

33 version changes in 154 days

First commit: September 2020
First commit: September 2020

Xero OAuth 2 Identity Service API
https://api.xero.com
Xero OAuth 2 Identity Service API

https://api.xero.com

First commit: September 2020

- Minor
- No version change
- Version decrease!
- No major releases
First commit: September 2020

- Breaking changes introduced in patches.
- Breaking changes introduced without version change.

Xero OAuth 2 Identity Service API
https://api.xero.com

No major releases
Xero OAuth 2 Identity Service API

Version: 2.9.4
Description: These endpoints are related to managing authentication tokens and identity for Xero API
openapi: 3.0.0
info:
    version: "2.3.0"
    title: Xero OAuth 2 identity service
    description: This specifying endpoints related to managing authentication tokens and identity for Xero API

contact:
    name: "Xero Platform Team"
    email: "api@xero.com"
    url: "https://developer.xero.com"
license:
    name: MIT
    url: 'https://github.com/XeroAPI/Xero-OpenAPI/blob/master/LICENSE'
servers:
    - description: Xero Identity service API
      url: 'https://api.xero.com'
openapi: 3.0.0
info:
  version: "2.3.0"
  title: Xero OAuth 2 identity service
  description: This specifying endpoints related to managing authentication tokens and identity for Xero API
contact:
  name: "Xero Platform Team"
  email: "api@xero.com"
  url: "https://developer.xero.com"
license:
  name: MIT
  url: "https://github.com/XeroAPI/Xero-OpenAPI/blob/master/LICENSE"
servers:
  - description: Xero Identity service API
    url: "https://api.xero.com"
Metadata-based versioning

```json
openapi: 3.0.0
info:
  version: "2.3.0"
  title: Xero OAuth 2 identity service
  description: This specifying endpoints related to managing authentication tokens and identity for Xero API
  termsOfService:
    "https://developer.xero.com/xero-developer-platform-terms-conditions/"
contact:
  name: "Xero Platform Team"
  email: "api@xero.com"
  url: "https://developer.xero.com"
license:
  name: MIT
  url: "https://github.com/XeroAPI/Xero-OpenAPI/blob/master/LICENSE"
servers:
  - description: Xero Identity service API
    url: "https://api.xero.com"
```
Dynamic Versioning
Dynamic Versioning

Visualisation tool: OAS2Tree

OAS2Tree

Install

Trouble Installing?
Um, dude?! There's a problem with the API!!

Yeah, that makes sense...

It's because we deployed a new version, and all the endpoints have changed.

Are you kidding? And when were you planning on telling us??

Uh... I mean I was planning to tell you...

Right now! I'm telling you about it right now! Now you know about the changes!

Really? When??

Now go fix your code.
"All in production" interface evolution pattern

“All in production” interface evolution pattern

APIs with multiple coexistent versions

![Bar chart showing the distribution of APIs with multiple coexistent versions.](chart.png)

- 119 APIs have exactly 2 versions.
- 30 APIs have exactly 3 versions.
- 10 APIs have exactly 4 versions.
- 5 APIs have exactly 5 versions.
- 5 APIs have exactly 6 versions.
- 3 APIs have exactly 7 versions.
- 1 API has exactly 8 versions.
- 1 API has exactly 10 versions.
- 1 API has exactly 14 versions.
APIs with multiple coexistent versions

Among 7,114 Web APIs

175 Web API adopting the “Two in production” Interface evolution pattern
Among 7,114 Web APIs

175 Web API adopting the “Two in production” Interface evolution pattern
APIs with multiple coexistent versions

Among 7,114 Web APIs

175 Web API adopting the “Two in production” Interface evolution pattern
Path-based versioning

https://{DomainName}/{basePath}
Path-based versioning

https://{DomainName}/{basePath}

http://myAPI.domain.com/v1/ressources
Path-based versioning

https://{DomainName}/{basePath}

http://myAPI.domain.com/v1/ressources

http://myAPI.domain.com/v2/ressources

http://myAPI.domain.com/v3/ressources
Versioning strategy should be defined upfront
Versioning strategy should be defined upfront

http://myAPI.domain.com/ressources

http://myAPI.domain.com/v1/ressources
Versioning strategy should be defined upfront

http://myAPI.domain.com/ressources

http://myAPI.domain.com/v1/ressources
7,114 Web APIs
186,259 Commits

Path-based versioning 1%
Meta data-based versioning 36%
Dynamic versioning 70%
Dynamic versioning 3%
What are the formats of the version identifiers?
"/versions":
  get:
    summary: Get all API versions
    description: Get all supported GitHub API versions.
    tags:
      - meta
    operationId: meta/get-all-versions
    externalDocs:
      description: API method documentation
      url: https://docs.github.com/rest/reference/meta#get-all-api-versions
    responses:
      '200':
        description: Response
        content:
          application/json:
            schema:
              type: array
              items:
                type: string
                format: date
                example: '2021-01-01'
            examples:
              default:
                value:
                  - '2021-01-01'
                  - '2021-06-01'
                  - '2022-01-01'
"/versions":
  get:
    summary: Get all API versions
    description: Get all supported GitHub API versions.
    tags:
    - meta
    operationId: meta/get-all-versions
    externalDocs:
      description: API method documentation
      url: https://docs.github.com/rest/reference/meta#get-all-api-versions
    responses:
      '200':
        description: Response
        content:
          application/json:
            schema:
              type: array
            items:
              type: string
              format: date
            example: '2021-01-01'
            examples:
              default:
                value:
                - '2021-01-01'
                - '2021-06-01'
                - '2022-01-01'
"/versions":
  get:
    summary: Get all API versions
    description: Get all supported GitHub API versions.
    tags:
      - meta
    operationId: meta/get-all-versions
    externalDocs:
      description: API method documentation
      url: https://docs.github.com/rest/reference/meta#get-all-api-versions
    responses:
      '200':
        description: Response
        content:
          application/json:
            schema:
              type: array
              items:
                type: string
                format: date
                example: '2021-01-01'
          examples:
            default:
              value:
                - '2021-01-01'
                - '2021-06-01'
                - '2022-01-01'
"/versions":
  get:
    summary: Get all API versions
    description: Get all supported GitHub API versions.
    tags:
      - meta
    operationId: meta/get-all-versions
    externalDocs:
      description: API method documentation
      url: https://docs.github.com/rest/reference/meta#get-all-api-versions
    responses:
      '200':
        description: Response
        content:
          application/json:
            schema:
              type: array
              items:
                type: string
                format: date
              examples:
                default:
                  value:
                    - '2021-01-01'
                    - '2021-06-01'
                    - '2022-01-01'
186,259 API Specs

→

Extract all metadata versions
186,259 API Specs

Extract all metadata versions

Version identifiers appearing in Metadata

5511 Distinct version identifiers
186,259 API Specs

Extract all metadata versions

Version identifiers appearing in Metadata

5498 Distinct version identifiers
186,259 API Specs  1,411,337 API Endpoints

Extract all metadata versions

Version identifiers appearing in Metadata

Extract versions from API endpoints

Metadata version identifiers appearing in API Endpoints

5498 Distinct version identifiers

385 Distinct version identifiers
186,259 API Specs

Extract all metadata versions

Version identifiers appearing in Metadata

5498 Distinct version identifiers

1,411,337 API Endpoints

Extract versions from API endpoints

Metadata version identifiers appearing in API Endpoints

385 Distinct version identifiers
5511 Distinct version identifiers
5511 Distinct version identifiers
5511 Distinct version identifiers
Version identifiers appearing in Metadata

5511 Distinct version identifiers
5511 Distinct version identifiers
Version identifiers appearing in Metadata

5511 Distinct version identifiers
Version identifiers appearing in Metadata

5511 Distinct version identifiers
5511 Distinct version identifiers
Version identifiers appearing in Metadata

5511 Distinct version identifiers
5511 Distinct version identifiers
385 Distinct version identifiers
385 Distinct version identifiers
https://github.com/USI-INF-Software/API-Versioning-practices-detection
Version format

Stable release version format

Pre-release version format
<table>
<thead>
<tr>
<th>Format</th>
<th>Most Frequent Version Identifier</th>
<th>#APIs</th>
<th>#Commits</th>
<th>VC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>max</td>
<td>avg</td>
</tr>
<tr>
<td>semver-3</td>
<td>1.0.0</td>
<td>40.45%</td>
<td>3531</td>
<td>1031</td>
</tr>
<tr>
<td>semver-2</td>
<td>1.0</td>
<td>64.92%</td>
<td>1093</td>
<td>3585</td>
</tr>
<tr>
<td>v*</td>
<td>v1</td>
<td>80.32%</td>
<td>489</td>
<td>692</td>
</tr>
<tr>
<td>date(yyyy-mm-dd)</td>
<td>2017-03-01</td>
<td>4.87%</td>
<td>327</td>
<td>52</td>
</tr>
<tr>
<td>other</td>
<td>v1b3</td>
<td>7.23%</td>
<td>213</td>
<td>222</td>
</tr>
<tr>
<td>integer</td>
<td>1</td>
<td>36.30%</td>
<td>48</td>
<td>143</td>
</tr>
<tr>
<td>v<em>beta</em></td>
<td>v1beta1</td>
<td>60.10%</td>
<td>115</td>
<td>360</td>
</tr>
<tr>
<td>date-preview*</td>
<td>2015-10-01-preview</td>
<td>11.93%</td>
<td>72</td>
<td>47</td>
</tr>
<tr>
<td>semver-3#</td>
<td>1.0.0-oas3</td>
<td>27.62%</td>
<td>33</td>
<td>215</td>
</tr>
<tr>
<td>v<em>beta</em>.</td>
<td>v2beta1.1</td>
<td>19.44%</td>
<td>26</td>
<td>30</td>
</tr>
<tr>
<td>latest*</td>
<td>latest</td>
<td>52.75%</td>
<td>25</td>
<td>137</td>
</tr>
<tr>
<td>v<em>alpha</em></td>
<td>v1alpha</td>
<td>51.34%</td>
<td>18</td>
<td>339</td>
</tr>
<tr>
<td>semver-SNAPSHOT*</td>
<td>1.0.0-SNAPSHOT</td>
<td>31.61%</td>
<td>18</td>
<td>172</td>
</tr>
<tr>
<td>semver-beta</td>
<td>v1.0-beta</td>
<td>28.37%</td>
<td>17</td>
<td>113</td>
</tr>
<tr>
<td>v<em>p</em>beta*</td>
<td>v1p3beta1</td>
<td>23.45%</td>
<td>9</td>
<td>347</td>
</tr>
<tr>
<td><em>beta</em></td>
<td>1beta1</td>
<td>100.00%</td>
<td>7</td>
<td>37</td>
</tr>
<tr>
<td>beta*</td>
<td>beta</td>
<td>65.49%</td>
<td>7</td>
<td>47</td>
</tr>
<tr>
<td>semver-alpha*</td>
<td>1.0.0-alpha</td>
<td>28.04%</td>
<td>7</td>
<td>48</td>
</tr>
<tr>
<td>semver-2#</td>
<td>1.3-DUMMY</td>
<td>12.26%</td>
<td>6</td>
<td>24</td>
</tr>
<tr>
<td>semver (beta*)</td>
<td>1.0 (beta)</td>
<td>29.89%</td>
<td>6</td>
<td>58</td>
</tr>
<tr>
<td>date(yyyy.mm.dd)</td>
<td>2019.10.15</td>
<td>10.45%</td>
<td>6</td>
<td>24</td>
</tr>
<tr>
<td>#semver-3</td>
<td>2019.0.0</td>
<td>29.73%</td>
<td>5</td>
<td>37</td>
</tr>
<tr>
<td>semver-rc*</td>
<td>1.0.0-rc1</td>
<td>38.14%</td>
<td>4</td>
<td>190</td>
</tr>
<tr>
<td>semver-4</td>
<td>6.4.3.0</td>
<td>3.31%</td>
<td>4</td>
<td>23</td>
</tr>
<tr>
<td>semver-rc*.</td>
<td>2.0.0-RC1.0</td>
<td>41.69%</td>
<td>4</td>
<td>85</td>
</tr>
<tr>
<td>v<em>alpha</em>.</td>
<td>v2alpha2.6</td>
<td>61.76%</td>
<td>3</td>
<td>26</td>
</tr>
<tr>
<td>alpha*</td>
<td>alpha</td>
<td>73.85%</td>
<td>2</td>
<td>35</td>
</tr>
<tr>
<td>dev*</td>
<td>dev</td>
<td>98.38%</td>
<td>2</td>
<td>172</td>
</tr>
<tr>
<td>date(yyyy-mm)</td>
<td>2021-10</td>
<td>67.44%</td>
<td>2</td>
<td>14</td>
</tr>
<tr>
<td>semver-pre*.</td>
<td>3.5.0-pre.0</td>
<td>100.00%</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>date(yyyymmd)</td>
<td>20190111</td>
<td>29.63%</td>
<td>1</td>
<td>13</td>
</tr>
<tr>
<td>semver-dev*</td>
<td>0.7.0.dev20191230</td>
<td>15.52%</td>
<td>1</td>
<td>40</td>
</tr>
<tr>
<td>v*date</td>
<td>v1-20160622</td>
<td>57.14%</td>
<td>1</td>
<td>18</td>
</tr>
<tr>
<td>semver-alpha*.</td>
<td>1.1.0-alpha.1</td>
<td>4.94%</td>
<td>1</td>
<td>146</td>
</tr>
<tr>
<td>Format</td>
<td>Most Frequent</td>
<td>#APIs</td>
<td>#Commits</td>
<td>VC</td>
</tr>
<tr>
<td>-------------------------</td>
<td>------------------------</td>
<td>-------</td>
<td>----------</td>
<td>----</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>max avg mdn stdev</td>
<td>max avg mdn stdev</td>
</tr>
<tr>
<td>semver-3</td>
<td>1.0.0</td>
<td>40.45%</td>
<td>3531</td>
<td>1031 28 17 37 496 4 0 17</td>
</tr>
<tr>
<td>semver-2</td>
<td>1.0</td>
<td>64.92%</td>
<td>1093</td>
<td>8585 30 15 116 77 1 0 4</td>
</tr>
<tr>
<td>v*</td>
<td>v1</td>
<td>80.32%</td>
<td>489</td>
<td>692 42 20 74 4 0 0 0</td>
</tr>
<tr>
<td>date(yyyy-mm-dd)</td>
<td>2017-03-01</td>
<td>4.87%</td>
<td>327</td>
<td>52 14 12 4 52 0 0 3</td>
</tr>
<tr>
<td>other</td>
<td>v1beta</td>
<td>7.25%</td>
<td>213</td>
<td>222 29 18 32 33 1 0 3</td>
</tr>
<tr>
<td>integer</td>
<td>1</td>
<td>36.30%</td>
<td>48</td>
<td>143 27 17 24 113 5 0 20</td>
</tr>
<tr>
<td>v<em>beta</em></td>
<td>v1beta1</td>
<td>60.10%</td>
<td>115</td>
<td>360 136 35 146 3 0 0 1</td>
</tr>
<tr>
<td>date-preview*</td>
<td>2015-10-01-preview</td>
<td>11.93%</td>
<td>72</td>
<td>47 13 12 5 2 0 0 0</td>
</tr>
<tr>
<td>semver-3#</td>
<td>1.0.0-oas3</td>
<td>27.62%</td>
<td>33</td>
<td>215 32 15 41 18 2 0 4</td>
</tr>
<tr>
<td>v<em>beta</em>,*</td>
<td>v2beta1.1</td>
<td>19.44%</td>
<td>26</td>
<td>30 24 24 4 12 3 3 4</td>
</tr>
<tr>
<td>latest*</td>
<td>latest</td>
<td>52.75%</td>
<td>25</td>
<td>137 27 15 28 2 0 0 0</td>
</tr>
<tr>
<td>v<em>alpha</em></td>
<td>v1alpha</td>
<td>51.34%</td>
<td>18</td>
<td>339 56 24 91 3 0 0 1</td>
</tr>
<tr>
<td>semver-SNAPSHOT*</td>
<td>1.0.0-SNAPSHOT</td>
<td>31.61%</td>
<td>18</td>
<td>172 32 16 38 36 5 0 9</td>
</tr>
<tr>
<td>semver-beta*</td>
<td>v1.0-beta</td>
<td>28.37%</td>
<td>17</td>
<td>113 40 29 29 9 1 0 2</td>
</tr>
<tr>
<td>v<em>p</em>beta*</td>
<td>v1p3beta1</td>
<td>23.45%</td>
<td>9</td>
<td>347 162 35 153 3 1 0 1</td>
</tr>
<tr>
<td><em>beta</em></td>
<td>1beta1</td>
<td>100.00%</td>
<td>7</td>
<td>37 15 11 9 0 0 0 0</td>
</tr>
<tr>
<td>beta*</td>
<td>beta</td>
<td>65.49%</td>
<td>7</td>
<td>47 26 26 12 0 0 0 0</td>
</tr>
<tr>
<td>semver-alpha*</td>
<td>1.0.0-alpha</td>
<td>28.04%</td>
<td>7</td>
<td>48 23 15 15 2 0 0 1</td>
</tr>
<tr>
<td>semver-2#</td>
<td>1.3-DUMMY</td>
<td>12.26%</td>
<td>6</td>
<td>24 16 15 5 3 2 2 1</td>
</tr>
<tr>
<td>semver (beta*)</td>
<td>1.0 (beta)</td>
<td>29.89%</td>
<td>6</td>
<td>58 39 46 13 46 18 26 18</td>
</tr>
<tr>
<td>date(yyyy.mm.dd)</td>
<td>2019.10.15</td>
<td>10.45%</td>
<td>6</td>
<td>24 22 24 4 24 20 24 9</td>
</tr>
<tr>
<td>#semver-3</td>
<td>2019.0.0</td>
<td>29.73%</td>
<td>5</td>
<td>37 22 17 11 3 1 2 1</td>
</tr>
<tr>
<td>semver-rc*</td>
<td>1.0.0-rc1</td>
<td>38.14%</td>
<td>4</td>
<td>190 60 20 75 8 4 5 3</td>
</tr>
<tr>
<td>semver-4</td>
<td>6.4.3.0</td>
<td>3.31%</td>
<td>4</td>
<td>23 16 17 5 9 2 0 4</td>
</tr>
<tr>
<td>semver-rc*,*</td>
<td>2.0.0-RC1.0</td>
<td>41.69%</td>
<td>4</td>
<td>85 54 63 26 0 0 0 0</td>
</tr>
<tr>
<td>v<em>alpha</em>,*</td>
<td>v2alpha2.6</td>
<td>61.76%</td>
<td>3</td>
<td>26 23 22 2 4 1 0 2</td>
</tr>
<tr>
<td>alpha*</td>
<td>alpha</td>
<td>73.85%</td>
<td>2</td>
<td>35 26 35 9 0 0 0 0</td>
</tr>
<tr>
<td>dev*</td>
<td>dev</td>
<td>98.38%</td>
<td>2</td>
<td>172 91 172 81 0 0 0 0</td>
</tr>
<tr>
<td>date(yyyy-mm)</td>
<td>2021-10</td>
<td>67.44%</td>
<td>2</td>
<td>14 13 14 1 2 1 2 1</td>
</tr>
<tr>
<td>semver-pre*,*</td>
<td>3.5.0-pre.0</td>
<td>100.00%</td>
<td>1</td>
<td>10 10 10 0 0 0 0 0</td>
</tr>
<tr>
<td>date(yyymmd)</td>
<td>20190111</td>
<td>29.63%</td>
<td>1</td>
<td>13 13 13 0 0 0 0 0</td>
</tr>
<tr>
<td>semver-dev*</td>
<td>0.7.0.dev20191230</td>
<td>15.52%</td>
<td>1</td>
<td>40 40 40 0 0 0 0 0</td>
</tr>
<tr>
<td>v*-date</td>
<td>v1-20160622</td>
<td>57.14%</td>
<td>1</td>
<td>18 18 18 0 2 2 2 0</td>
</tr>
<tr>
<td>semver-alpha*,*</td>
<td>1.1.0-alpha.1</td>
<td>4.94%</td>
<td>1</td>
<td>146 146 146 0 0 0 0 0</td>
</tr>
</tbody>
</table>
Version format

Stable release version format
- Major version number
- SemVer
- Tag
- Date
- Other

Pre-release version format
- Develop
- Snapshot
- Preview
- Alpha
- Beta
- Release Candidate

55

27

28
### APIs with stable formats

<table>
<thead>
<tr>
<th>Type</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>SemVer</td>
<td>4941</td>
</tr>
<tr>
<td>Major version number</td>
<td>804</td>
</tr>
<tr>
<td>Date</td>
<td>336</td>
</tr>
<tr>
<td>No version</td>
<td>268</td>
</tr>
<tr>
<td>Preview</td>
<td>73</td>
</tr>
<tr>
<td>Other</td>
<td>61</td>
</tr>
<tr>
<td>Beta</td>
<td>37</td>
</tr>
<tr>
<td>Tag</td>
<td>25</td>
</tr>
<tr>
<td>Snapshot</td>
<td>17</td>
</tr>
<tr>
<td>Alpha</td>
<td>10</td>
</tr>
<tr>
<td>Release Candidate</td>
<td>8</td>
</tr>
</tbody>
</table>
# APIs with stable formats

<table>
<thead>
<tr>
<th>Version Type</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>SemVer</td>
<td>4941</td>
</tr>
<tr>
<td>Major version number</td>
<td>804</td>
</tr>
<tr>
<td>Date</td>
<td>336</td>
</tr>
<tr>
<td>No version</td>
<td>268</td>
</tr>
<tr>
<td>Preview</td>
<td>73</td>
</tr>
<tr>
<td>Other</td>
<td>61</td>
</tr>
<tr>
<td>Beta</td>
<td>37</td>
</tr>
<tr>
<td>Tag</td>
<td>25</td>
</tr>
<tr>
<td>Snapshot</td>
<td>17</td>
</tr>
<tr>
<td>Alpha</td>
<td>10</td>
</tr>
<tr>
<td>Release Candidate</td>
<td>8</td>
</tr>
</tbody>
</table>
The diagram shows the adoption of semantic versioning (SV) and versioned previews (VP) over the years 2015 to 2022. The y-axis represents the number of commits, while the x-axis shows the years. Each bar represents a year, with different shading indicating different semantic versioning and versioned preview adoption rates.

- **Stable (Semantic Versioning)**: Light gray shading
- **Stable (Versioned)**: Dark gray shading
- **Preview (Semantic Versioning)**: Medium gray shading
- **Preview (Versioned)**: Black shading

Over the years, the adoption of semantic versioning has increased significantly, with a notable peak in 2020. The versioned previews show a more gradual increase, with a significant rise in 2021.

The data suggests a growing trend towards using semantic versioning and versioned previews, indicating a shift towards more structured and predictable software releases.
v*beta* or v*alpha*
Provisioner

Version: v1beta1.4
Description: With the Provisioner service in Splunk Cloud Services, you can provision and manage tenants.
Version identifiers formats in APIs with multiple coexistent versions

![Bar chart showing the distribution of APIs with different numbers of coexisting versions.](image-url)

- 119 APIs with exactly 2 coexisting versions
- 30 APIs with exactly 3 coexisting versions
- 10 APIs with exactly 4 coexisting versions
- 5 APIs with exactly 5 coexisting versions
- 5 APIs with exactly 6 coexisting versions
- 3 APIs with exactly 7 coexisting versions
- 1 API with exactly 8, 10, or 14 coexisting versions
- No APIs with exactly 1 or 2 coexisting versions
Version identifiers formats in APIs with multiple coexistent versions

- Major Version Number (MVN)
- Other version formats

Number of coexisting versions:
- 2: 2,102
- 3: 462
- 4: 184
- 5: 73
- 6: 51
- 7: 25
- 8: 3
- 9: 1
- 10: 5

#Commits:
- 0
- 1,000
- 2,000
Version identifiers formats in APIs with multiple coexistent versions

Number of coexisting versions

#Commits

- Major Version Number (MVN)
- Other version formats

Number of coexisting versions

#Commits

- Tag—MVN (169)
- SemVer (100)
- Date (64)
- Beta (60)
- Other (60)
- MVN—Other (58)
- Date—MVN (28)
- Beta—MVN (18)
- Tag (17)
- SemVer—MVN (12)
- Tag—Beta (5)
- SemVer—Beta (1)
- Beta—Other (1)
Version identifiers formats in APIs with multiple coexistent versions

- Major Version Number (MVN)
- Other version formats

Number of coexisting versions:
- 2: 2,102
- 3: 462
- 4: 184
- 5: 73
- 6: 51
- 7: 25
- 8: 3
- 9: 1
- 10: 5

Commit counts by version format:
- Tag—MVN (169)
- SemVer (100)
- Date (64)
- Beta (60)
- Other (60)
- MVN—Other (58)
- Date—MVN (28)
- Beta—MVN (18)
- Tag (17)
- SemVer—MVN (12)
- Tag—Beta (5)
- SemVer—Beta (1)
- Beta—Other (1)
How do developers version Web API?

Approach

- **Usage of the two in-production patterns** in 175/7114 APIs. Up to 14 coexistent versions in the case of an API.

- **Usage of Path-based versioning.** 36% of the APIs used Path-based versioning.
  - 496 APIs switched to/from Path-based versioning in the middle of their history.

- **Usage of Metadata-based versioning.** 70% of the APIs use Metadata-based versioning.
How do developers version Web API?

Version identifiers formats

- Version identifiers are expressed in **55 different formats**

- Noticeable **switch to SemVer during histories of API** that change version identifiers formats.

- **4941 APIs used only SemVer** during their whole history

- Significant **increase in the use of simpler pre-release** versioning formats.
Future Work

How do developers change the version identifier on each API change?

✓ Focus on a subset of APIs with parsable version identifiers during all their history
Future Work

How do developers change the version identifier on each API change?

✓ Focus on a subset of APIs with parsable version identifiers during all their history
✓ Analyse the version increase and corresponding APIs changes
References

[1] https://semver.org/
[2] https://docs.npmjs.com/about-semantic-versioning
An Empirical Study of **Web API Versioning Practices**

Souhaila Serbout (souhaila.serbout@usi.ch), Cesare Pautasso (cesare.pautasso@usi.ch)

https://github.com/USI-INF-Software/API-Versioning-practices-detection

API Visualisation tool: http://api-ace.inf.usi.ch/openapi-to-tree/