SEMANTIC VERSION

Speaker notes

authors :

- s. Lavazais
- d. Six

sources:

SUMMARY

```
version 1.2.3
        ^ ^ ^
        ‖ ‖ ╚═ Patch : backward compatible bug fixes
        ‖ ╚═══ Minor : features backward compatible manner
        ╚═════ Major : breaking changes API
```

Speaker notes

Semantic version is the set of rules dictating how version numbers of an API are assigned and incremented. Based on but not limited to common practices of closed and open source software development.

Consider a version like `X.Y.Z` where `X` is `MAJOR` number, `Y` is `MINOR` number and `Z` is `PATCH` number.

DEPENDENCY HELL

```
my-app 4.7.2
╠═ internal lib 5.6.1
║  └─ logger lib 2016.3.4
║     └─ another external lib 0.3.0
╠═ database lib 12(cookie)
║  └─ io lib 3.0.0
║     └─ another external lib 0.3.0
╠═ list lib 0.1.2alpha
║  └─ external lib 4.8
║     └─ another external lib 0.3.0
```

Speaker notes

When version numbers are not standardized, solving a simple issue of a common dependency can be a real nightmare, since there is no common way to increment version of dependents libs

SET OF RULES

Speaker notes

here are the set of rules defines in specifications

PUBLIC API

only after the release of version 1.0.0

Speaker notes

any application/lib must declare a public API (it could be simple documentation). It must be only define after the first major version.

- rule #1
- rule #5

X.Y.Z FORM

```
1.2.3
^ ^ ^
| | └── Patch
| └──── Minor
└────── Major
```

Speaker notes

a normal version must be formatted with 3 non-negative integer separated by dot. (not lead by `v`)

each integer represent a different type of modification of the application/lib imply.

- rule #2

PRE-RELEASE RULES

init dev start with version "0"

```
0.x.x
```

Speaker notes

initial development start at version 0.x.x, the increment of the version a state can occur any time and the version is not stable.

- rule #4

PRE-RELEASE RULES

add "–" and other identifiers separated by " . "

```
1.1.2-alpha.1
1.1.2-5.7.9
```

Speaker notes

pre-released version can be defined by adding identifiers after the patch integer lead by a hyphen.

then each identifier must be separated by a dot.

- rule #9

BUILD METADATA

can add "+" and other identifiers separated by " . "

```
1.1.2+012
1.1.2+21AF26D3
1.1.2-beta+exp.sha.749f34
```

Speaker notes

versions for build metadata should be separated from the rest of the version by adding a plus sign

- rule #10

INCREMENT RULES

any modification after first release is a new version

```
1.2.3
```

Speaker notes

any modification after the first release must imply a modification of the version

- rule #3

INCREMENT RULES

Patch version

bug fixes (backward compatible)

```
1.2.3
```

Speaker notes

go to the next slide for notes

INCREMENT RULES

Patch version

bug fixes (backward compatible)

```
1.2.4
```

Speaker notes

bug fixes should increment the patch version

- rule #6

INCREMENT RULES

Minor version

new features (backward compatible)

```
1.2.4
```

Speaker notes

go to the next slide for notes

INCREMENT RULES

Minor version

new features (backward compatible)

```
1.3.0
```

Speaker notes

adding a new features should increment the minor version, only if these modifications has not broken the backward compatibility of the application/lib

this incrementation imply that the patch version is reset to 0

- rule #7

INCREMENT RULES

Major version

any backward incompatible changes

```
1.3.0
```

Speaker notes

go to the next slide for notes

INCREMENT RULES

Major version

any backward incompatible changes

```
2.0.0
```

Speaker notes

any breaking changes (that broke the backward compatibility) should increment the major version,

this incrementation imply that the patch and the minor versions are reset to 0

- rule #8

PRECEDENCE RULES

```
1.0.0 < 2.0.0 < 2.1.0 < 2.1.1
```

Speaker notes

the precedences rules define how versions are compared to each other

the 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).

- rule #11

PRECEDENCE RULES

with pre-release

```
1.0.0-alpha < 1.0.0
```

Speaker notes

pre-released version has lower precedence compared to normal version

- rule #11

PRECEDENCE RULES

with pre-release

```
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
```

Speaker notes

precedence of pre-released versions of same core version must determine by comparing each dot separated identifier from left to right until a difference is found.

- rule #11

USING SEMANTIC-RELEASE

📦🚀 semantic-release

```
verb(scope): message

notes
```

```
fix(security): fix security check
```

```
feat(security): add security standard
```

```
feat(security): new security standard

BREAKING CHANGES: don't support old security standard
```

Speaker notes

Semantic Versioning can be easily implemented to any project by using tools like semantic-release

semantic-release parse commit messages to build a version number. by default this tool will build the first release if no tag of version exist (1.0.0)

CONFIG SEMANTIC-RELEASE

```
{
  "branches": ["main"],
  "tagFormat": "${version}",
  "plugins": [
    "@semantic-release/commit-analyzer",
    "@semantic-release/exec",
    "@semantic-release/release-notes-generator",
    "@semantic-release/changelog",
    "@semantic-release/git",
    "@semantic-release/github"
  ]
}
```

Speaker notes

Semantic Release (SR) use a simple file to run (`releaserc`). here an example of settings.

setting `branches` are the release branches. setting `tagFormat` is the format of tag (when a release occur). `plugins` allow SR to manipulate different aspect on the release mechanism

we're gonna tweak `exec` and `git` to allow us to add more parameters to our releases.

CONFIG SEMANTIC-RELEASE

```
["@semantic-release/exec", {
  "prepareCmd": "echo ${nextRelease.version} > version.txt"
}]
```

```
["@semantic-release/git", {
  "assets": ["CHANGELOG.md", "version.txt"],

  "message": "chore(release): version ${nextRelease.version}"
}]
```

Speaker notes

on `exec`, we describe what we do just before the release.

on `git`, we describe what we commit on release, and the commit message

RUN SEMANTIC-RELEASE

```
npx semantic-release
```

Speaker notes

No notes on this slide.

RUN SEMANTIC-RELEASE

```
i  Analyzing commit: feat(presentation): add of semantic-release demo
i  The release type for the commit is minor
i  Analyzing commit: refactor(presentation): improving comment for export to pdf format
i  The commit should not trigger a release
i  Analysis of 2 commits complete: minor release
```

Speaker notes

No notes on this slide.

GOING FURTHER

plugins order matter

```
{
  "branches": ["main"],
  "tagFormat": "${version}",
  "plugins": [
    "@semantic-release/commit-analyzer",
    "@semantic-release/exec",
    "@semantic-release/release-notes-generator",
    "@semantic-release/git",  //will commit without changelog
    "@semantic-release/changelog",
    "@semantic-release/github"
  ]
}
```

Speaker notes

No notes on this slide.

release steps mechanism

- `verifyConditions`
- `analyzeCommits`
- `verifyRelease`

Speaker notes

inside the whole release mechanism plugins, SR allows to manipulate different steps to execute everything withing the release.

THANK YOU

Authors:

S. LAVAZAIS

D. SIX

Sources:

HTTPS://EN.WIKIPEDIA.ORG/WIKI/DEPENDENCY_HELL

HTTPS://SEMVER.ORG/

Speaker notes

No notes on this slide.