How To - Versioning Strategy for Scala

Semantic versioning

Introduction

Semantic versioning (also referred as **SemVer**) is a versioning system that has been on the rise over the last few years. With new plugins, addons, extensions, and libraries being built every day, having a universal way of versioning software development projects is a good thing to help us keep track of what's going on.

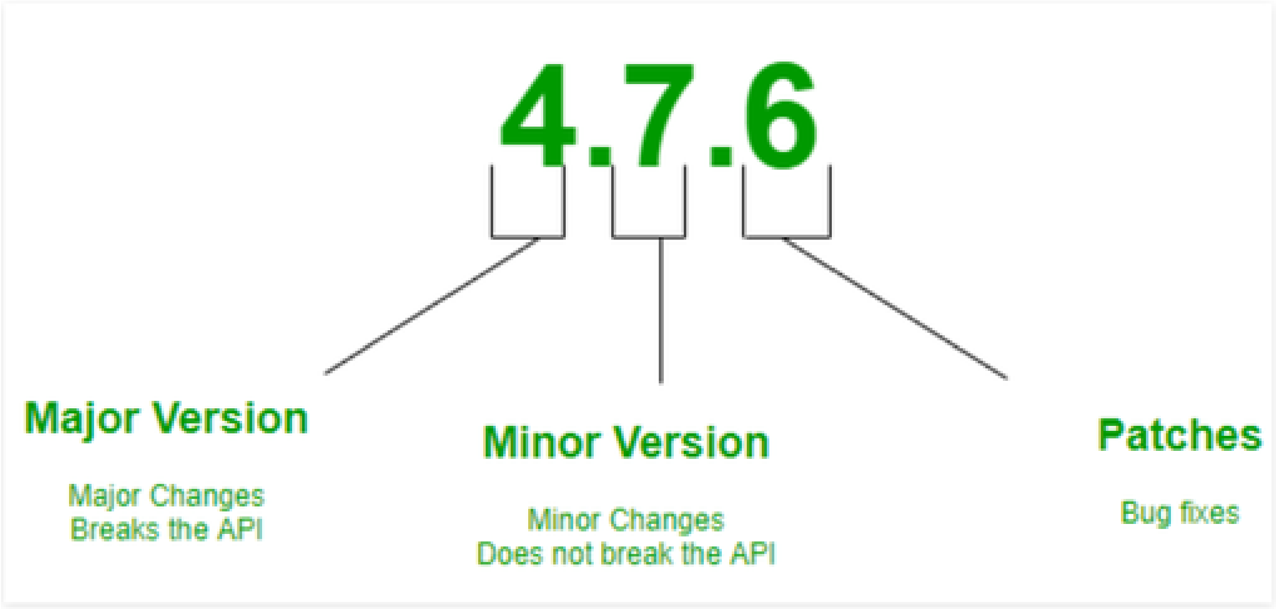
1. SemVer is a short form
2. Uses three numbers for version instead of two ex. 1.1.1
3. Requires a public API

What is SemVer?

SemVer is a 3-component system in the format of `x.y.z` where:

- X stands for a major version.
- Y stands for a minor version.
- Z stands for a patch.

So we have **Major.Minor.Patch**



Major Major changes of API

Minor New features Does not break API

Patch Bug Fixes

Ex. Bug Fix version 4.7.7

EX New feature added 4.8.0

Ex Breaking the API 5.0.0

Pre- Release versioning

- Initial development uses major version 0 e.g. 0.1.0
- You may use Pre-release string e.g. Alpha1,rc3 etc.
- These are appended to end   e.g 1.0.0.alpha1
- A Publicly released API starts at 1.0.0

Incrementing semantic versions in published packages

Code Status	Stage	Rule	Ex version
First Release	New Product	Start with 1.0.0	1.0.0
Backward compatible bug fixes	Patch Release	Increment the third digit	1.0.1
Backward compatible new features	Minor release	Increment the middle digit and reset last digit to zero	1.1.0
Changes that break backward compatibility	Major release	Increment the first digit and reset middle and last digits to zero	2.0.0

Benefits

- Clearer Compatibility /dependencies
- Encouraged well defined APIs

Specification

https://semver.org/

Semantic Version for Scala

sbt-autoversion:

The sbt-autoversion plugin builds on the sbt-release and sbt-git plugins to automatically manage the version bump to apply (major, minor or patch version bumps), based on commits messages patterns.

Add the following line your **project/plugins.sbt**

```
addSbtPlugin("org.scala-sbt" % "sbt-autoversion" % "1.0.0")
```

Since sbt-autoversion is an AutoPlugin, it will be automatically available to your projects, given you're including both the sbt-release and sbt-git plugins.

Usage

sbt-autoversion automatically wires itself in the setting of sbt-release's releaseVersion setting, meaning that you can use the sbt-release's release with-defaults command and use the non-interactive release process with the correct version configured.

- **LatestTag** Fetches the latest Git tag, based on Semantic Versioning ordering
- **UnreleasedCommits** lists commits since the latest tag/release
- **suggestedBump** hows what version bump the plugin has computed and would automatically apply on the next release.

Requirements

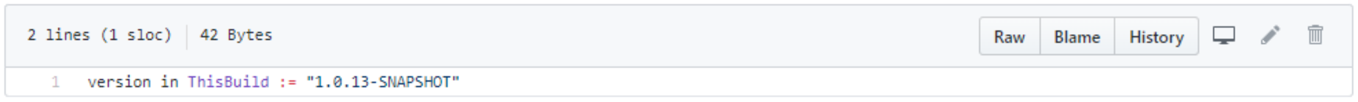
- sbt 0.13.5+
- The version of the project should follow the semantic versioning scheme on semver.org with the following additions:
  - The minor and bugfix (and beyond) part of the version are optional.
  - There is no limit to the number of subversions you may have.
  - The appendix after the bugfix part must be alphanumeric (`[0-9a-zA-Z]`) but may also contain dash characters `-`.
  - These are all valid version numbers:
    - 1.2.3
    - 1.2.3-SNAPSHOT
    - 1.2beta1
    - 1.2-beta.1
    - 1.2
    - 1
    - 1-BETA17
    - 1.2.3.4.5
    - 1.2.3.4.5-SNAPSHOT
- A publish repository configured. (Required only for the default release process.)

Version.sbt

- Since the build definition is actual Scala code.
- For this reason, sbt-release won't ever touch your build definition files, but instead writes the new release or development version to a file defined by the setting release-version-file, which is set to file("version.sbt") by default and points to $PROJECT_ROOT/version.sbt
- By default the version is set on the build level.
- This behavior can be controlled by setting `releaseUseGlobalVersion` to `false`, after which a version like `version := "1.2.3"` wi ll be written to `version.sbt`.

Example:

```
version in ThisBuild := "1.0.13-SNAPSHOT"
```



2 lines (1 sloc) | 42 Bytes

```
version in ThisBuild := "1.0.13-SNAPSHOT"
```

Release Process

The default release process consists of the following tasks:

- Check that the working directory is a git repository and the repository has no outstanding changes. Also prints the hash of the last commit to the console.
- If there are any snapshot dependencies, ask the user whether to continue or not (default: no).
- Ask the user for the `release version` and the `next development version`.
- Run **clean**
- Run **test:test ,** if any test fails, the release process is aborted.
- Write `version in ThisBuild := "$releaseVersion"` to the file `version.sbt` and also apply this setting to the current build state.
- Commit the changes in **version.sbt.**
- Run **Publish**

You can set the release version using the argument `release-version` and next version with `next-version`.

Example:

```
release release-version 1.0.99 next-version 1.2.0-SNAPSHOT
```

```
#############################################
Version Strategy for scala using jenkins file
#############################################

stage("Build") {
            steps {
                script {
                    version = nextVersionFromGit()
                    checkout scm
                    echo 'Start Compiling'
                    build 'Scala'
                }
            }
        }


def nextVersionFromGit() {
def latestVersion = sh returnStdout: true, script: "cat build.sbt | grep
version | awk '{print \$5}' || echo 0.0.0"
latestVersion = latestVersion.replaceAll("\"", "");
def prefix = BRANCH_NAME.replaceAll("/","_")
def (major, minor, patch) = latestVersion.tokenize('.').collect {
it.toInteger() }
def nextVersion
switch (env.BRANCH_NAME) {
case 'master':
nextVersion = "${major + 1}.0.0"
break
case 'develop':
nextVersion = "${major}.${minor + 1}.0"
break
default:
nextVersion = "${major}.${minor}.${patch +
1}_${prefix}_${currentBuild.number}"
break
}
print "Next Version"
print nextVersion
nextVersion
}
```