# A practical take on GitFlow and Semantic Versioning

## Or how I finally slayed my versioning indecision

*Posted on March 28, 2016*

**I have always had a complicated relationship to versioning.** At first glance, versioning is seemingly simple: it is just a number. Looking closer, a version number can carry a lot of meaning besides identifying a set of files and binaries.

For starters, every increase tells a story about the changes made since the last version, and how significant those changes were. Then we have the marketing aspect: sometimes the major and minor parts of the version number are part of your product name, which can lead to a lot of hesitation and uncertainty regarding when and how to bump

the version number. Finally, there is the emotional aspect: are these new features really significant enough to warrant a version 2.0?

Nowadays, my preference is to handle marketing- and technical version numbers separately. This allows me to use names like *MyAwesomeProduct2016* or *MyAwesomeProduct365*, and it takes away all the emotional ties to the real version number. As for the technical side, I like to use Semantic Versioning, especially together with GitFlow. That combination has gained a lot of popularity (although some people feel it is overly complicated), but just pick a system that serves your needs and suits your workflow. The important thing is to get a system in place and eliminate the versioning indecision.

Both GitFlow and Semantic Versioning have very detailed specifications; making it very clear how you should work with branches, what to name them and under what circumstances you should bump your version number. But when it comes to actual implementation details, they only give you a few recommendations in the form of examples. Not very helpful when one is trying to figure out what a "best practices" implementation of GitFlow/SemVer should look like.

> **Just a clarification – I do get the specs and the big picture. :-)**
> What I am contemplating are the little things: should I utilize a build number or not? Are there any other practical reasons for that besides `AssemblyInfo` versioning? Should the build number be reset or not when the version number is bumped? Lots of questions, and none of them very important. But as you all know, the devil is in the details... :-)

Since I am currently setting up a continuous deployment process for a Lancelog, a SaaS product I am working on, I wanted to get my versioning system as good as possible right from the outset.

What I needed was a couple of blog posts outlining what worked well for others and what didn't, but apparently my Googling capabilities sucks because I didn't find any. So – in the hope of helping someone else out, here's my take on the matter. And if you happen to be a black belt in this area, please feel free to correct me on any misconceptions.

# Decision, decisions, decisions

While pouring through the specifications yet again, a lot of questions popped up in my head. As an example, consider SemVer's concepts of pre-release versions and build metadata: which scheme is best suited for what? Just because we have the ability to label a build as `alpha.1` or `beta.32`, should we use it? What about build numbers? What format is most readable? The list of questions kept growing:

- When should I bump to version number? At the start of a feature or when it is being completed?
- What should I call the builds in the `develop` branch? And should they even be tagged?
- What labeling convention should I use when branching out to a new release branch?
- Should I reset the build number when I bump the version number or not?
- Where should I store the current version number?

- How should I translate the three-part SemVer version number to a four-part `AssemblyInfo` version number (with guaranteed uniqueness)?

**The first question**, about when to change the version, is actually being answered in the GitFlow spec; it recommends that one *defers this decision until the creation of a release branch*. Perfectly reasonable, since we never know what might end up in a release. We might have planned to release a certain feature, just to be taken aback by a bug that needs to be fixed before the feature.

## Let's illustrate this and all the other questions with an example.

Say we're on version `1.2.0` in the `master` branch, a.k.a production. We are planning a new feature called X, so the next version should become `1.3.0`.
Then the client reports a critical bug, forcing us to prepare a hotfix. We fix the bug and bump the version to `1.2.1`. But, given the insights gained from fixing the bug, we now realize that we might need to break the public API to get X working the way we want. And if we do that, we should no longer bump to version `1.3.0`. Instead, we should move to version `2.0.0`.

> **Rule:** version bumping occurs either when we branch out from `develop` to `release`, or when we branch out from `master` to a hotfix branch.

Next question: naming conventions in the `develop` branch. Since all changes in the other branches should be merged back into `develop`, I

think the naming should reflect that `develop` is virtually always the latest version of the product. Hence, I use the convention `a.b.c-wip.d`, where `wip` stands for work-in-progress and `d` is the build number.

> **Rule:** always make sure that the version number in the `develop` branch is in sync with the latest number in any `hotfix` or `release` branches.

In the example above, the `develop` branch was at `1.2.0-wip.123` when we created the hotfix branch `hotfix/1.2.1`. Following the last rule, when we merge back to the `develop` branch, it gets bumped to `1.2.1-wip.x`.

This rule can get complicated when we are working on a `hotfix`- and a `release` branch simultaneously. Which SemVer number should propagate back to `develop`? In most cases, the `release` branch version number should trump the `hotfix` one. Especially since we probably want to merge `hotfix` changes back into the `release` branch before wrapping it up.

What about naming conventions in the `release` branches? I don't expect releases to stay in this branch for very long, so the full alpha/beta/rc1/rc2 life cycle seems like overkill. In the end, I decided to use the `rc`-prefix, which leads to the following notation: `a.b.c-rc.d`. Again, `d` is the build number.

**Let's continue with the example.** We decide to break the API, so we create a new release branch from `develop` called `release/2.0.0`. The first tag in this branch should then be `2.0.0-rc.x`, where x is the build number. But what should x be in this case? This brings up the

question about when the counter should be reset, if ever. Let's examine each alternative:

> **Alternative 1:** Never reset the build counter

This approach guarantees uniqueness across all builds, but in my opinion, it looks ugly. Another peripheral concern is that .NET `AssemblyInfo` revision numbers are limited to a maximum value of 65535. Ok, probably not a real-world issue on a small project unless you are extremely trigger happy.

> **Alternative 2:** Reset the build counter whenever the version number is bumped

Together with the SemVer version number, this also guarantees uniqueness. It also looks way nicer, since the build number seldom will reach beyond three digits.

However, it introduces another problem: to remember to reset the counter whenever you change the version number. Or, if we go for full automation, how to detect when the version number has changed.

Anyway, as of now, alternative 2 this is my preferred approach. I haven't yet figured out how to achieve full automation regarding the build numbers, but maybe Git hooks and TeamCity's REST API can be a solution.

In the end, the takeaway here is that you should pick a system that always produces unique version numbers, regardless what you will use them for.

> **Rule:** ensure that each versioning tag is unique in the repo.

# Where should we store the version number?

At first, I thought the build server was the perfect choice for this. But what if the team grows; maybe we want to give everyone the ability to bump the number, but we don't want everyone to have access to the build server? So I changed my mind. Until a better idea presents itself, I store the SemVer part of the version number in a simple text file on disk. It is easy enough to parse that file using Powershell and feed the version into TeamCity's build pipeline.

What about the build counter? Well, like I mentioned above, it would be nice with full automation – I just need to figure out the best way to accomplish that first. For now, it lives in TeamCity as a regular build counter, which I reset manually.

# A question of trust

**Back to the question about tagging the develop branch**: is it really necessary? Well, I want the `develop` branch to be deployable at all times. But, to be pragmatic, sometimes bugs slip through. With a build process in place that runs all the tests and tags every successful build, that is no problem: every tag in the `develop` branch is a receipt that everything was a-ok at that point in time. Without the tags, there is no easy way of knowing where to start looking, since we wouldn't know which one of the commits that was the latest one that worked.

# One final example

Just to really hammer it in, here's an example of how the repo might evolve over time using the above rules (using the alternative where the build counter is being reset):

| Project event/activity | Tag |
|---|---|
| Project starts, `master` is empty and we are committing our first feature in `develop` | `0.0.0-wip.1` |
| Feature A is committed to `develop` from a feature branch | `0.0.0-wip.2` |
| Quick bug fix directly in `develop` | `0.0.0-wip.3` |
| Feature B is merged into `develop` from a feature branch | `0.0.0-wip.4` |
| Time for the first minor release! Branch out from `develop` to a new release branch called `release/0.1.0` and reset the build counter | `0.1.0-rc.1` |
| We polish the release a bit and commit | `0.1.0-rc.2` |
| To keep `develop` in sync, we merge the release branch back, creating a merge commit | `0.1.0-wip.3` |
| In the meantime, another team member commits feature C in `develop` | `0.1.0-wip.4` |
| Time for production release: we merge `release/0.1.0` into `master` and `develop` | `0.1.0-release.5`<br>`0.1.0-wip.6` |
| A small refactoring is committed to `develop` | `0.1.0-wip.7` |

| Project event/activity | Tag |
|---|---|
| A critical bug is reported in production; create the hotfix branch `hotfix/0.1.1` | `0.1.1-hotfix.1` |
| We merge the updated version file back to `develop` to avoid duplicate tags (since the counter got reset) | `0.1.1-wip.2` |
| In the meantime, someone commits another refactoring in `develop` | `0.1.1-wip.3` |
| The hotfix is finalized | `0.1.1-hotfix.4` |
| We merge it back into `master` and `develop`, thus creating to additional merge commits | `0.1.1-release.5`<br>`0.1.1-wip.6` |
| The team then merge in another feature branch in `develop` | `0.1.1-wip.7` |
| Time for another minor release; we create the release branch `release/0.2.0` from `develop` and reset the build counter | `0.2.0-rc.1` |

...and so on. You get the idea. As the example shows, I am not a fan of fast-forwarding; I think merge commits tell an important part of the development story. Also, I like to use a global build counter, like the **Autoincrementer** plugin for TeamCity.

To wrap things up – this is just a proposal implementation, and by no means "best practice". I just implemented it myself, so it will probably evolve over time. All feedback is appreciated! :-)

## Did you like this post?

**Join my mailing list and get notified whenever a new post is out.**

Topics range from team development, to application development strategy, to productivity tactics.

Email address

Subscribe

I won't spam you, and you can unsubscribe at any time.

## Filed under

Categories: CI/CD | Workflow | Development Practices

Tags: Semantic Versioning | GitFlow | Conventions

**Igal** • 5 months ago

I have a quick question; currently I am working in a Scala/Sbt project and we are using Travis for CI. We have configured in such way that after every merge on develop Travis bumps up the version and makes a commit to the repository since the version.sbt file is within the repository. I was just wondering what could be another way of using automated increments without giving this responsibility to Travis.



**Fredrik Holm** Mod → Igal • 5 months ago

On the product I'm working on now (https://www.lancelog.io), I bump my SemVer numbers manually while Team City handles the build counter. In other words, pretty much the same setup as you describe.

In a solo project like mine I guess I could move the build counter out of the CI process, but within the confines of a team I think it should be a centralized responsibility. I guess you resort to hooks in the source control system, but then you're only making a lateral move. Think I would keep it in Travis. :-)



**Henrik Lynggaard Hansen** • a year ago

I really liked your article, although your example about hotfixes doesn't take into account how to deal with the version number already being used

Example:
Master: 1.1.0
Release branch: 1.1.1

Now 1.1.1 is built and ready but has only made it to the test environment (dev -> test -> preprod ->prod) when a new hotfix is needed.

If you want to guarantee minimal changes (as this is a production hotfix), you branch of master and bump the version number, but alas 1.1.1 is already taken.
- I could pick 1.1.2 but that indicates it has the changes from 1.1.1 is included which they aren't
- I could pick a version such as 1.1.1-hotfix.1 which would be free but not semantically clean ?
- I could force developers to base the fix of 1.1.1 but that might require extra testing and would not constitute a minimal change..

How would you solve above?



**Fredrik Holm** Mod → Henrik Lynggaard Hansen • a year ago

Tricky one! Think I would go with your second option, using 1.1.1, but with some additional changes. With my current setup (TC + Octopus) the order of releases isn't as important as every release having a unique name, so I would probably tag the hotfix branch 1.1.1-hotfix1. Then I would simply bump the version number in the release branch to 1.1.2 (and maybe rename that branch as well).
Not perfect, but what do you do... :-)



**David Fallah** • 2 years ago

Great article. I've also adopted a GitFlow/Semantic versioning approach to my projects but recently I've tried setting up a CI workflow and have had trouble deciding what version numbering scheme to adopt for this. I've opened a question about it here: http://softwareengineering....

It was very nice that you asked exactly the same questions that I was thinking and provided answers for them. I considered almost exactly the same approach as you but chose to dismiss it because of this issue (quoting from http://semver.org/):

"When major, minor, and patch are equal, a pre-release version has lower precedence than a normal version. Example: 1.0.0-alpha < 1.0.0"

In your example, you would first tag a release as, say, 1.2.0, and then tag *future versions* of your project (on the develop branch) as, e.g. 1.2.0-wip1, 1.2.0-wip2. However, from SemVer's perspective these constitute *earlier* versions than 1.2.0; a build tagged 1.2.0-wip1 would be considered *leading up to* a 1.2.0 build.

This issue would cause problems with package management systems that assume SemVer versioning schemes, such as NuGet package manager. If you wanted to deploy a NuGet package for v1.2.0-wip3 of your project, the NuGet package manager would still show you the 1.2.0 build as the most up-to-date one, even if you had the "show pre-release" option enabled.



**Fredrik Holm** Mod ➚ David Fallah • 2 years ago

Thanks for pointing this out, makes me wonder how I could miss something that obvious..! :-)
The problem persists though - what should one call the current WIP-build? Ok - this problem might not occur in larger organizations where releases are (hopefully) planned in advance, but for small startup it might be more of a moving target.

What did you decide upon instead?

On a side note, I use TeamCity together with Octopus Deploy, and it works perfectly fine in spite of being semantically wrong. :-)



**David Fallah** ➚ Fredrik Holm • 2 years ago

I haven't started yet, but this is what I have planned:

Let's say that I've just released v1.1.0 of my project. The moment that I start work on the develop branch, I immediately bump the patch number (v1.1.1). This is because I know for a fact that *any* changes I make between releases will at a minimum constitute patch-level changes.

Similarly, as I'm progressing along I'll bump the minor version (v1.2.0) as soon as I start introducing new (non-breaking) features. This will typically be at the start of the first new feature branch since the previous release.

Ditto with breaking changes - bump the major version (v2.0.0) as soon as I start introducing any.

This approach would mean that all my builds were semantically "correct", even very early WIP builds off the develop branch. It would also accommodate for unanticipated early releases. For example, if I start work on the develop branch with the intention of implementing a few extra features before the next release, I might be tempted to immediately bump the version number to 1.2.0. However, if I discover a critical bug early on, I would need to break off hotfix/1.1.1 and end up going from (e.g.) v1.2.0-build0049 to v1.1.1-build0050 - not very nice. With this approach, this would not happen as I would reserve bumping the minor or major components until necessary. At the time I break off hotfix/1.1.1, my project.json would still be at v1.1.1 and so I wouldn't need to change anything.

One issue with this approach is that you might end up going from e.g. v1.1.1-build0067 to v1.2.0-build0068 (without any release in-between) but I don't consider that *too* problematic.

^ | v • Reply • Share ›

**Fredrik Holm** Mod ➚ David Fallah • 2 years ago

Awesome, love the idea of "at least a patch" - will definitely accommodate that in my process.

As for the hotfix, shouldn't that just be backmerged into develop, thus bumping up develop to 1.2.1-build0050?

^ | v • Reply • Share ›

**David Fallah** ➚ Fredrik Holm • 2 years ago

What I mean is that the project.json (version string) won't need to be modified during creation of the hotfix branch or at any point within it, as you'd have done the patch-bump right at the start of the develop branch (following on from the prior release).

You're free to merge that hotfix branch whenever but I assume you'd want to get it done as soon as possible, before starting work on any other features. In this case, the hotfix branch would get merged into both master and develop. The project.json on the master branch

would one patch-level higher than that in the previous commit on that branch, and identical to the project.json already in develop.

Reply • Share ›

**Fredrik Holm** Mod ➚ David Fallah • 2 years ago

Yeah, true that. :-)

There sure is a lot of scenarios to cater for. Got another tip in a FB discussion for this post; namely to use GitVersion: https://github.com/GitTools....

Haven't got around to try it out myself but it looks promising.

Reply • Share ›



Gender-neutral Language Disclaimer

Powered by BeSharper Blogging Platform