How to organize versionning

Branches

Two main branches :

- `prod` which is automatically deployed when pushed to remote,
- `staging` which is the container of the currently built next released version. It also is the place where final testing is performed to get a production ready version. It can be pushed to remote for backup purposes without triggering a deployment. So `staging` is `dev` but staging is, to me, a more meaningful name.

These two main branches are not designed to be directly commited. Instead, they receive merging from other branches.

Each time a merge is done on `prod` branch, it gives a new deployment. So I have to name this deployment. This is done by tagging the merge commit (so I have to force a merge commit ? read about `git tag -a`). My politics on how a version number `v1.1.2` evolves is :

```
v1.1.2
 | | |_ bug fix.
 | |___ feature release.
 |_____ compatibility break (like migration !).
```

Remarks :

- Every bugfix branch have to be created from `prod` . Indeed, `staging` may, at bug discovery time, already have received other merges from feature development. Fixing the bug does not have to put in production features that are maybe not totally finalized AND more than this, bug fixing is a higher priority than new features (correct progression of numbering).
- Intimately bound to the previous remark, when I merge a bugfix on `prod` , I rebase `staging` from `prod` . So, added features are built uppon `prod` version.

  This way keep a clean commit history based uppon production branch together with a correct release numbering.

Temporary branches

Toward these two main branches, several others can exists temporarily :

- to develop bug fixes,
- to develop new features.

They have to receive clear names. For this, I can use prefixes. For example :

- `feature/invoicing`
- `bugfix/issue-123`
- `test/input-model`

Think about merging regularly `staging` in these branches (yes, **from** staging !) :

```
git checkout feature/plouf
git merge staging # or git rebase staging
```

Rebasing is not a problem on not-shared branches. It allow to rewrite commit stories to something more linear.

So when the team grows, it could be preferable to evolve from one `staging` branch to a couple `staging` / `dev` branches, `staging` , shared, staying away from any rebasement.

Commits

Name

Well named commits improves readability. I say structured names. Let's considere this :

```
# commit name grammar :
<type>(<scope>): <subject>
```

Where `type` is in `{docs, feature, fix, refactor, style, test, ...}`

- style(code) : formatting | indenting | missing ';' | typo | etc.
- style(ui) : improve icon appearance.
- style(ux) : improve user exp like better navigational stuff.
- test : new tests created without code change

and `scope` is optional. `subject` have to be a clear summary at present tense. (details have not to be in the commit name).

Commit often

Commits have to be small and then frequent. But, they have to be self-consistent ! So I may use git `stashing` :

When working on multiple branches simultaneously, I may need to switch between branches without committing my current changes (avoid 'work in progress' commit). Git's stashing feature allows me to temporarily save my work in progress and apply it later when needed :

```
# stash my current changes :
git stash save "work in progress for feature X"
# make what you need on other branches
# Apply your stashed changes after switching back to the original branch
git stash apply
```