How to organize versionning

Two main branches : `prod` and `staging` .

Both have remote tracking/tracked branches. Only prod is dedicated for deployment.

- `prod` which is production version. No commit. No rebase. Only merges from staging or bug-fix-branches. It is tagged with accurate version number. Tagging allow easy roll back to previous state.
- `staging` which is the container of the *in progress next version*. No commit. No rebase. Only merges from temporary (see below) branches.

> Vocabulary for `prod` branch state :
>
> - production version : it is a version dedicated for jelastic deployment.
> - delivered version : it is a production version a DNS record points to.

Development workflow :

1. New features are developed in temporary branches and merged to `staging` to contribute to next production update.
2. When `staging` is locally stable, it is merged into `prod` . This has to be tested on deployment environment before being delivered to clients.
3. The tested prod version can either : CANNOT DO THIS IF MIGRATION IMPLIED BECAUSE OF A SHARED DATABASE NODE. HAVE TO CREATE A DEDICATED TEMPORARY ENVIRONNEMENT TO TEST A COMPATIBILITY BREAK..
   - be stable : hence,
     - it receive a numbered tag following this policy :

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

       commands are :

       ```
       # locally
       git tag -a vx.y.z prod
       git push --tags
       # on jelastic web-ssh
       git pull --tags
       ```

     - DNS record is updated to point to this deployment. Update *delivered*.
   - be incomplete or buggy : hence,
     - DNS is not updated.
     - new commits are provided to fix issues. (do not forget to commit to staging too, and maybe temporary branches)

Roll back

It may happen that bugs or issues appear on a delivered version. Many scenari are possible :

1. Previous stable delivered version still in a jelastic container : update DNS record.
2. Roll back needed :

```
git checkout my-desired-commit-or-tag
git diff prod > ./diff.patch # way to go from prod to my-desired...
git checkout prod
cat ./diff.patch | git apply # apply changes to prod
rm ./diff.patch # remove patch file so it is not committed
git commit -am "Rolled back to my-desired-commit-or-tag"
git push
```

Bug fix (Hotfix) : made from prod branch !

```
git checkout -b newbugfix vx.y.z # new branch from vx.y.z version
# few commits + local tests
git checkout prod
git merge newbugfix # merge to staging and maybe temporary branches too
git push
# run WAN testing
git tag -a vx.y.z+1 prod
git push --tags
# on jelastic web-ssh : git pull --tags
```

Remarks :

- Every bugfix branch have to be created from `prod` . Indeed, `staging` or `dev` 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 merge (and not rebase ! Remember, none of the remote branches should be rebased) `prod` within `staging` . So, added features are built without the said bug.

Managing temporary branches

Temporary branche purposes are :

- bug fixes,
- new features development.

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` within 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. I think merging is preferable if `staging` contains a bug fix but I am probably wrong.

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.

Git Cheatsheet

- `git checkout -b foo [from]` creates a branch named foo

  Optional `from` argument is reference to a commit (hash, tag or branch-name).

- `git push [-u] origin/branch_name` publish current branch to remote

  Optional `-u` is a shortcut for --set-upstream : keep a tracking reference between the two branches to allow future `git push` (or `git pull` to target the right branch). Think : without tracking, local and remote branches, even if sharing the same name, are totally independant ! (Hey ! with tracking too !!!)

- `git branch -d name` delete a local branch
- `git push origin --delete name` delete a remote branch
- `git branch -m [old] new` rename a branch.

  Use optional `old` argument if branch to rename is not checked out. To rename a remote branch, rename it locally, push this "newly created" branch to remote and delete remote branch.

- merging options :
  - `--no-ff` creates a merge commit even when fast-forward would be possible.
  - `--squash` combines all integrated changes into a single commit instead of preserving them as individual commits.

Commits policy

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 atomic (only one purpose : this leads to a clear history and reduce merging conflicts) and then frequent. They also have to be self-consistent ! So I may use git `stashing` (or `amending`) :

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
```

Alternatively, I can use commit amendment feature :

```
# amending a commit
git commit -m "a partial or erroneous commit"
# make what I need on other branches (and only other branches !)
# and go back to initial branch, performing changes
git add 'what have to be added'
git commit --amend # amend previous commit with new changes
```