GIT-FLOW Guidelines

For IT developers

Elaborated by CREM :

- Lesly Houndole, lesly.houndole@crem.ch
- Vincent Roch, vincent.roch@crem.ch

With the support of HES-SO :

- Daniel Hunacek, daniel.hunacek@hevs.ch
- Antoine Widmer antoine.widmer@hevs.ch

Tuesday, 31 January 2017

Funded by the Horizon 2020 programme of the European Union

Date: 31/01/2017

Table of Contents

Introduction 3

Glossary of terms 4

Responsabilities 5

Access rights 5

Git-flow definition 6

How do we use Git-flow? 7

- Branches 7
  - Master and Develop branches 7
  - Feature branches 8
  - Hotfix branches 9
  - Release branches 10

References 11

Introduction

This document has been written to outline and define the version control process we will be using for HotMaps 2020 project. It is aimed at anyone who would like to get involved by contributing to the source code. We will be using Git to version control the codebase.
Our aim is to keep a clean but informative history of commits, allowing us to revert mistakes easily.

We will use “Bump version” in order to increment the version number to a new, unique value. Bump is a software project's VERSION which adds the CHANGES, and tags with GIT. You can download bump at:
https://gist.github.com/pete-otaqui/4188238

Glossary of terms

This section describes the basic terms we use and assume knowledge of. Full documentation of Git terms can be found at:
https://git-scm.com/doc

Term	Description
Repository	A Git repository is essentially a collection of branches, all related to the same code base.
Commit	A commit in Git can be thought of as a snapshot of the repository at any given point.
Commit message	A commit message is the message that describes the changes made in a commit, viewable with 'git log'.
Branch	A branch in Git can be thought of as a sequence of commits which can be separated from all other commits. A branch may be merged or rebased onto another branch to insert its commit history to other branches.
Master	The master branch is the parent branch of all other branches, including develop branch. In our process, it will receive only two types of commits: Hotfixes and Major updates. The master branch will always contain stable code and will be what the public will receive.
Develop	The develop branch is a branch take from the master branch. It is where all feature branches will be derived from. The code contained in develop should always be stable. This branch will be the source of the latest codebase, including nightly builds for example. Develop will receive all Hotfixes as well as master.
Feature	A feature branch is a branch used to develop a new functionality. Very large feature additions may require multiple feature branches to be created.
Hotfix	A hotfix branch is used for critical bug fixes. This type of branch is taken directly from master and, once the work is completed, merged directly to master and develop
Release	A release branch support preparation of a new production release
Tag	A tag is the term used to define a textual label that can be associated with a specific revision.
Pull requests	Pull requests let you tell others about changes you have pushed to a repository on Git. Once a pull request is opened, you can discuss and review the potential changes with collaborators and add follow-up commits before the changes are merged into the repository.

Responsabilities

CREM, HES-SO and TUW will be responsible for reviewing the code. Everyone else will have to use pull request in order to modify master, develop, and release branch.

All source code for HotMaps Project must take place in the Git repositories owned by TUW.

Access rights

There are two kinds of access rights:

- **owner** access
- **collaborator** access

The repository **owner** will have full access to the repository:

- Invite collaborators (https://help.github.com/articles/inviting-collaborators-to-a-personal-repository)
- Change the visibility of the repository
- Merge a pull request on a protected branch, even if there are no approved reviews
- Delete the repository (https://help.github.com/articles/deleting-a-repository)

Collaborator access has limited access to the repository:

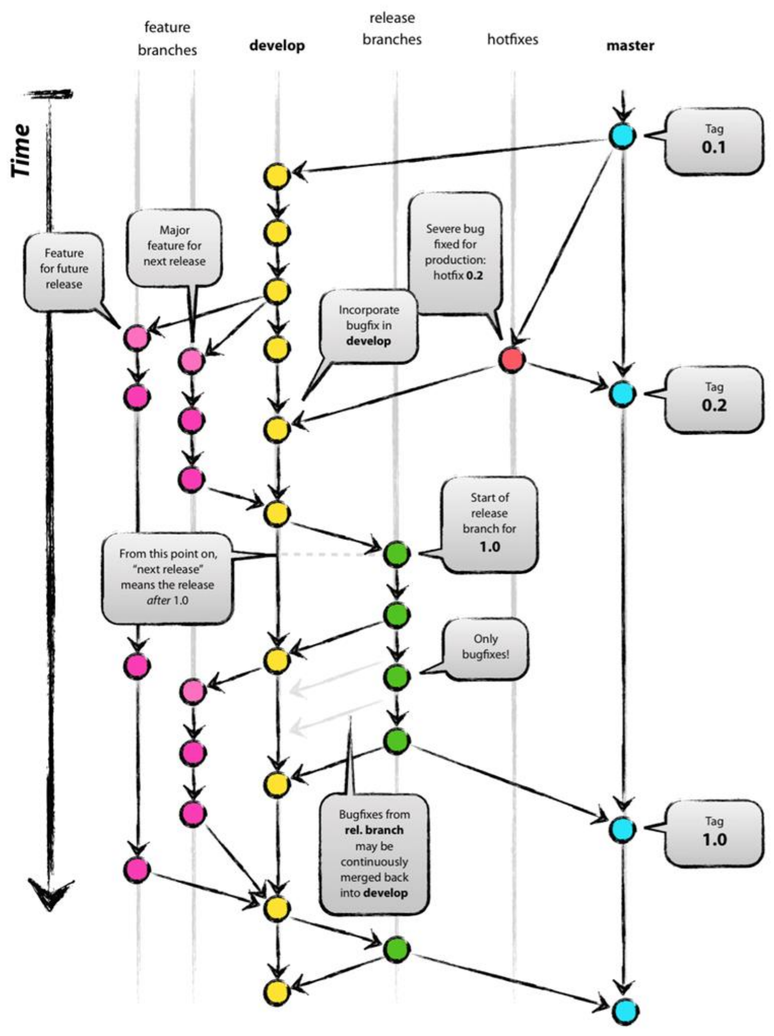
- Push to (write), pull from (read), and fork (copy) the repository
- Apply labels and milestones
- Open, close, re-open, and assign issues
- Edit and delete comments on commits, pull requests, and issues
- Merge and close pull requests
- Send pull requests from forks of the repository
- Create and edit Wikis
- Create and edit Releases
- Remove themselves as collaborators on the repository
- Submit a review on a pull request that will affect its “mergeability”

For HotMaps projects, the following collaborators will have the **owner** access:

- Daniel Hunacek, HES-SO, daniel.Hunacek@hevs.ch
- Mostafa Fallahnejad, TUW, fallahnejad@eeg.tuwien.ac.at
- Lesly Houndole, CREM, Lesly.houndole@crem.ch
- Sara Fritz TU Wien, fritz@eeg.tuwien.ac.at

Git-flow definition

Git-flow is a workflow for development with the Git version control system. It is how we structure our repository, our commits and branches. Git-flow is essentially no more than a set of procedures that every team member has to follow in order to come to a managed software development process. The diagram below (see URL below) shows an overview of this process:


feature branches
develop
release branches
hotfixes
master
Time
Tag 0.1
Feature for future release
Major feature for next release
Severe bug fixed for production: hotfix 0.2
Incorporate bugfix in develop
Tag 0.2
Start of release branch for 1.0
From this point on, "next release" means the release after 1.0
Only bugfixes!
Bugfixes from rel. branch may be continuously merged back into develop
Tag 1.0

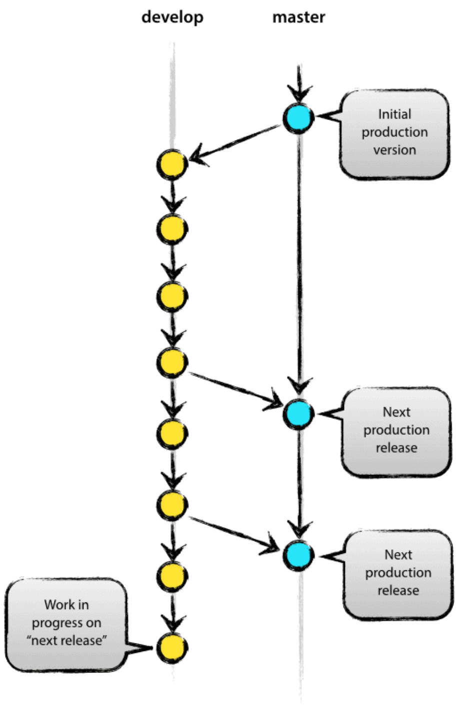
For the most part, this document is based on Vincent Driessen's article "A successful Git branching model" at http://nvie.com/posts/a-successful-git-branching-model/.

The develop branch will always aim to be stable, and bugs will have a branch created to fix them. We will enforce this by running tests for pre- and post- merges of the develop branch, and code reviews for all merge requests.

How do we use Git-flow?

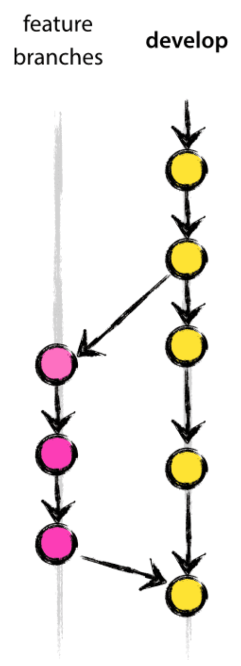
Branches

Master and Develop branches


develop
master
Initial production version
Next production release
Next production release
Work in progress on "next release"

There will be two long running branches, master and develop. These will be protected branches, meaning only a few persons will have the right to push changes directly to these branches. All additions to these branches must be done via a merge request (pull request). This allows code review of everything that enters the framework and ensure both master and develop branches stay stable.

Feature branches



When developing new functionalities or features, or simply refactoring/optimizing core code, you should create a feature branch. Feature branches are branched from develop and named after their ticket number or a brief description of the work being done.

Rules for feature branch:

- Must branch off from develop
- Must merge back into develop

Naming convention:

Feature branches should always be prefixed with “feature-”, for example “feature-12345” or “feature-something”.

Commands :

Switched to a new branch "myfeature":

```
$ git checkout -b myfeature develop
```

After adding new feature “myfeature” branch

Switch to branch *develop*:

```
$ git checkout develop
```

Merge “myfeature” branch to develop

```
$ git merge --no-ff myfeature
```

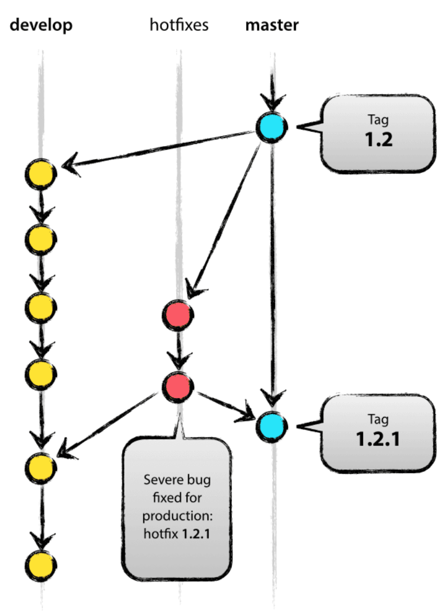
Delete branch “myfeature”

```
$ git branch -d myfeature
```

Push to develop branch

```
$ git push origin develop
```

Hotfix branches



When a serious bug is found or a fix is needed immediately, you should create a hotfix branch. Hotfix branches are spun directly from master, tested as soon as possible and merged into the master branch using a -squash merge. This gives us a commit checkpoint of the fix being implemented. Hotfix branches should also be merged directly into develop.

Rules for Hotfix branch:

- Must branch off from master
- Must merge back into develop and master

Naming convention:

Hotfix branches should always be prefixed with “hotfix-”, for example “hotfix-*”

Command:

Switched to a new branch "hotfix-1.2.1"

```
$ git checkout -b hotfix-1.2.1 master
```

Files modified successfully, version bumped to 1.2.1.

```
$ ./bumpversion.sh 1.2.1
```

[hotfix-1.2.1 41e61bb] Bumped version number to 1.2.1

```
$ git commit -a -m "Bumped version number to 1.2.1"
1 files changed, 1 insertions(+), 1 deletions(-)
```

It is really important to bump the version number after branching off!

Then you should commit the fix

```
$ git commit -m "Fixed severe production problem"
[hotfix-1.2.1 abbe5d6] Fixed severe production problem
5 files changed, 32 insertions(+), 17 deletions
```

At the end we merge back into master and develop.
Switched to branch master

```
$ git checkout master
```

Merge made by recursive. (Summary of changes).

```
$ git merge --no-ff hotfix-1.2.1
```

You should not forget to tag this version of code.

```
$ git tag -a 1.2.1
```

After the code is pushed on master you should push it again on develop.
Switched to branch develop.

```
$ git checkout develop
```

Merge made by recursive. (Summary of changes)

```
$ git merge --no-ff hotfix-1.2.1
```

There is an exception to the rules we write above. If there is a release branch, you must merge on release branch instead of develop.

Rules for Hotfix branch when release branch currently exists:

- Must branch off from master
- Must merge back into release and master and develop

Delete branch hotfix-1.2.1 (was abbe5d6).

```
$ git branch -d hotfix-1.2.1
```

Release branches

Release branch is the branch that will be used for the new production release. Release branches are created from develop branch

Rules for release branches:

- Must branch off from develop
- Must merge back into develop and master

Naming convention:

Release branches should always be prefixed with “release-*/”, for example “release-1.2”

Command:

Switch to a new branch "release-1.2"

```
$ git checkout -b release-1.2 develop
```

Files modified successfully, version bumped to 1.2.

```
$ ./bump-version.sh 1.2
```

After you should make a commit
```
$ git commit -a -m "Bumped version number to 1.2"
```

When the release branch is online, you should switch to release and merge it to master, then tag the master branch. Afterwards you should merge back to develop as follow.

Switched to branch master
```
$ git checkout master
```

Merge made by recursive. (Summary of changes)
```
$ git merge --no-ff release-1.2
```

Tag this version of code
```
$ git tag -a 1.2
```

Switched to branch develop
```
$ git checkout develop
```

Merge made by recursive. (Summary of changes)
```
$ git merge --no-ff release-1.2
```

References

- https://gist.github.com/pete-otaqui/4188238
- http://nvie.com/posts/a-successful-git-branching-model/
- https://git-scm.com/doc
- https://git-scm.com/book/en/v2/Git-Tools-Submodules

www.hotmaps-project.eu

Funded by the Horizon 2020 programme of the European Union