Workflow for the “Projet Long” POPART

Author : Matthieu Pizenberg
Version : v1.0
Date of version : 12/02/2015

Foreword

The project we work on is going to be resumed by POPART team at the end of “projet long”. Therefore we should have a clean workflow, compatible with the POPART team one. This document attempts to formalize our workflow in order to make sure we follow right working methods.

Table of contents

1 Our workflow

1.1 The idea

The idea is to combine the Integration-Manager workflow (see appendix 1) with the Gitflow workflow (see appendix 2). Thereby we have a clear integration process and still a fine granularity of the project.

1.2 The implementation

As proposed in the Integration-Manager workflow, each developer has its own private and public repository. In addition the integrator manages another repository : the reference repository (called “ref”). It will play the role of both integration and reference. Integration will be done on the private clone of the integrator and the public repository will be the reference for all developers.
As so the “ref” repository will only contain master, develop, and release branches but no feature branch. Feature branches will only appear in private (and sometimes public) developers’ repositories. Developers might need to publish some feature branches on their public repository if they work together on a same functionality. In that case they can pull commits from other developers directly.

1.3 The developer point of view

I will present here in practice the way a developer of our project should work. For many reasons we chose GIT as distributed revision control system to version our code. As a developer I am free to use the hosting service I want, to store my public repository provided I can manage the authorizations for other developers. However it is recommended that all developers use Github.

1.3.1 Developer first steps

First I need to fork the “ref” repository (the icon is at the upper right corner of the page). Here is the official project you can fork :
https://github.com/OeufsDePie/MATRIX.git
For example, here is my personal fork of the project :
https://github.com/mpizenberg/MATRIX
Now I can clone locally (my private repository) my own fork of the project (my public repository). The method to do it depends on whether I have configured SSH connection or not (for more information about it see : https://help.github.com/articles/generating-ssh-keys/). For me, I will have to write the following commands :

```
cd /the/path/where/I/want/to/clone/it
git clone git@github.com:mpizenberg/MATRIX.git
```

If I did not have configured SSH, it would be instead :

```
git clone https://github.com/mpizenberg/MATRIX.git
```

Now I can work on my local repository and do whatever I want :)

1.3.2 Developer workflow

First you need to add the remote repository corresponding to the reference repository :

```
git remote add ref https://github.com/OeufsDePie/MATRIX.git
```

Now you can work on your features. Let's say you have developed some features in a branch called `feature1` and you want to integrate your work in the ref repository.

Step 1 : check that your develop branch is up to date :

```
git checkout develop
git fetch ref
git merge --ff-only ref/develop
```

If the last command went without problem, it means that your develop branch has been updated correctly. Otherwise it means that your develop branch had diverged from ref. In that case you need first to rebase your develop branch on the ref develop branch :

```
# Only if last command failed
git rebase ref/develop
```

Step 2 : merge your feature work in develop branch :

```
git merge feature1 --no-ff -m "What feature1 does"
```

Step 3 : push your work to your public repository :

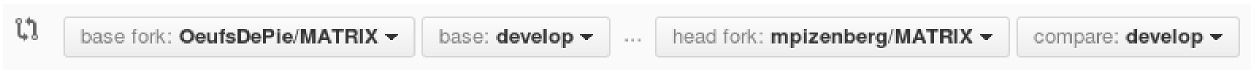
```
git push origin develop
```

Last command may fail if you needed to rebase before. In this case force the push :

```
# Only if last command failed
git push origin develop -f
```

Step 4 : make a pull request from github :

Finally you need to integrate your work in the ref repository. Go to your forked project (for me : https://github.com/mpizenberg/MATRIX). Click on the button "Branch : …" and choose develop. Click on the "compare, review, create a pull request" button (). Check that it compares to OeufsDePie develop branch :



And send your pull request ! Now it is up to the integrator.

Step 5 : Only if the integrator asks you to rebase
It means you were 2 to ask for integration and the last will need to rebase. In that case you will have a few operations to do :

```
# Only if the integrator asks to rebase
git checkout develop
git fetch ref
git rebase ref/develop
git push origin develop -f
# Go to Github and make a pull request (step 4)
```

1.4 The integrator point of view

Once a pull request has been made, you are notified by email and you can view its details and the comments online on Github. In order to integrate properly the pull request it is better to do it manually in a clone of the ref repository. Once everything is set up you can do it. Let's say you want to integrate the develop branch of Nicolas (`nicolas` will be the name of the remote `https://github.com/gaborit-nicolas/MATRIX.git`). Here are the main steps of its operation :

Step 1 : updating from ref repository

```
git checkout develop
git pull ref develop
```

Step 2 : fetching the pull request

```
git fetch nicolas
```

Step 3 : trying to merge fast-forward the pull request in a temporary branch

```
git checkout -b nicolas-develop     #different from nicolas/develop
git merge --ff-only nicolas/develop
# if last command failed, delete the temporary branch
git checkout develop
git branch -d nicolas-develop
```

If merging fails, ask the developer to rebase his work on current ref/develop.

Step 4 : If merging ff worked, test the integrated work and push it to ref

If last command worked and that there are compilation and tests to run in order to check that the new code is valid, you can do it now. **If the new code is successfully compiled and tested** you can merge and push develop to ref :

```
# if compilation and tests passed
git checkout develop
git merge --ff-only nicolas-develop
git branch -d nicolas-develop
git push ref develop
```

Otherwise, if there are problems notify the developer in the comments of the pull request. You also need to delete the temporary branch :

```
# If compilation or tests failed
git checkout develop
git branch -d nicolas-develop
```

Aliases to make it easier

In order to make all this easier and not wasting time remember the right order and the right commands you should use extended aliases (call shell commands). It is possible by starting the alias with the character “!”. Moreover it can take arguments, using the function syntax. Here are examples of aliases you should use to integrate easily pull request :

```
# Integrate work from a remote : pull the request
integratepull = "!f() { \
echo '################################################'; \
echo 'Updating from ref repository :'; \
echo ''; \
git checkout $2; \
git pull ref $2; \
echo '################################################'; \
echo "fetching the pull request from $1 :"; \
echo ''; \
git fetch $1; \
echo '################################################'; \
echo 'Trying to merge fast-forward the pull request :'; \
echo ''; \
git checkout -b $1-$2; \
git merge --ff-only $1/$2; \
}; f"
```

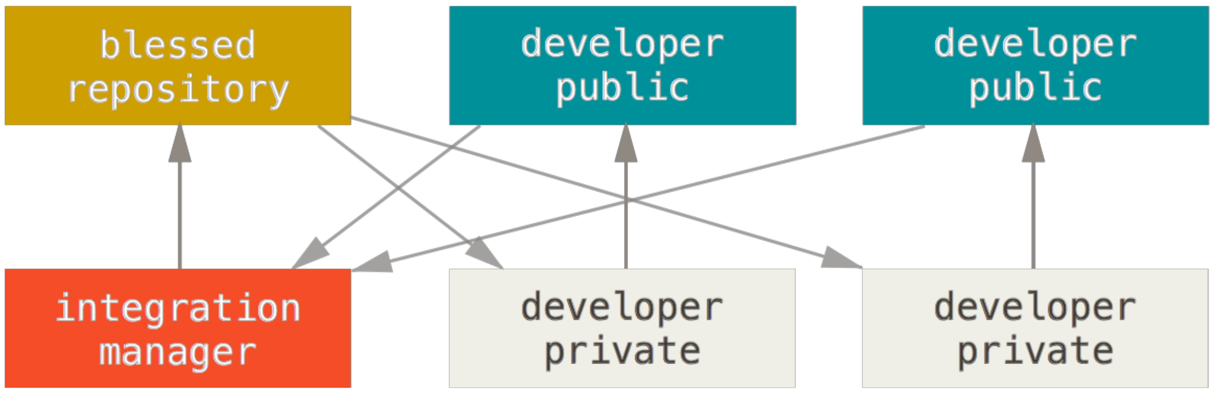
```
# Integrate work from a remote : push the validated work
integratepush = "!f() { \
git checkout $2; \
git merge --ff-only $1-$2; \
git branch -D $1-$2; \
git push ref $2; \
}; f"
```

```
# Integrate work from a remote : cancel the integration process
# because builds or tests failed
integratecancel = "!f() { \
git checkout $2; \
git branch -D $1-$2; \
}; f"
```

The way you could use those aliases would be something like that :

```
# Integrate work from a remote : pull the request
git integratePull nicolas develop
# If it did not work (developer need to rebase)
git integrateCancel nicolas develop
# Else (it worked), then if builds and tests passed
git integratePush nicolas develop
# Else : builds or tests failed
git integrateCancel nicolas develop
```

2 Appendix 1 : Integration-Manager workflow



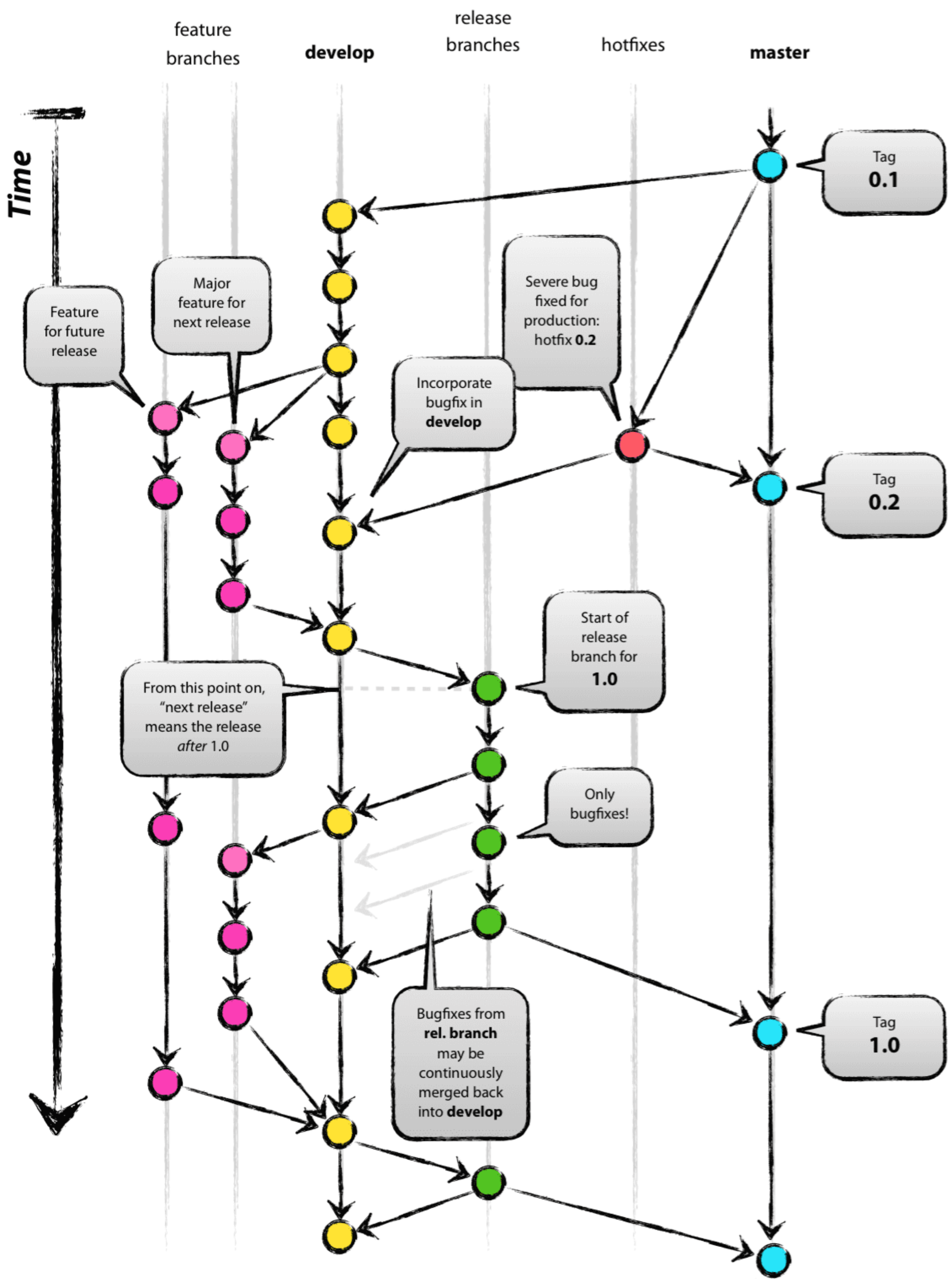
That figure is quite explicit. Each developer has its own fork of a blessed repository (which we call in our project “ref” for reference). This is made possible since you can have multiple remotes in Git. Therefore, each developer will have at least 2 remotes :

1. origin : the remote pointing to your fork of the original repository (ref).
2. ref : the original and only official repository of the project.

When a developer has developed a feature mature enough to be integrated to the official ref repository, he makes a pull request so the integrator retrieves his work and push it to the ref (if it is valid of course).

Finally every other developer will be able to pull from ref to update their own repository and get the work that had been integrated.

3 Appendix 2 : The Gitflow workflow


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

The workflow we will draw upon is an adaptation of the Gitflow workflow made by Vincent Driessen. You will find a full description of his workflow here. The previous figure depicts pretty well the whole workflow and I will resume here the main ideas.

3.1 The Main Branches

The ref repository holds two main branches with an infinite lifetime :

- master : the stable, production-ready state of the project
- develop : the integration branch of every feature

When the code in develop is stable enough we start a process to merge it into master.
Theoretically master should be stable enough to use hooks at commits to launch builds and tests automatically.

3.2 Supporting Branches

We should use also other branches with limited life time such as :

- Feature branches : exist as long as a functionality is in development. It starts from develop and must be merged back in develop at some point.
- Release branches : started when develop almost reflects the state of a new release. It starts from develop and must be merged back into master and develop.
- Hotfix branches : when a critical bug in production version must be resolved immediately. It starts from master and must be merged back into master and develop (or rather a release than develop if there currently exists one).