Documentation Git hub

Project Stella Incognita
Author: Team 1
28-02-2013

Table of contents

Introduction

Git hub is a Source Control tool that is used all over the world by developers in order to keep large projects up to date and merge everyone's work into releases without too much pain. Most of you will have already worked with it, but for those who haven't, this document will tell you the how-to's of Git hub.
For those of you who have worked with Git before, it will still be a good idea to read through this document, as many operations will be project-dependent and you might just learn a thing or two. Also, I am going to put some work into this, so it is appreciated by me, the author.

The default tool that will be used for the actual Source Control, is Git Extensions, though I will be explaining the features of Git Bash (which works a lot better for me) as well.
For merging conflicts, the tool that I will be using is TortoiseGit.
This documentation for Git relies on the Development Model (Git Flow), which is way of version control very well suited for large projects with multiple releases (PSI most definitely is such a project). Thus far, only documentation for Git Flow have shown signs that it is only supported in Git Bash, but if possible, documentation for Git Extensions will be added as well. For now, this is a very nice document of how to install Git Flow (at least on Windows).
A complete list of commands for Git Bash can be found here.
For a complete description on Git Extensions, reference this file.

Connecting to Git

Before we can start working on the project, we first need to tell Git from where and to where we want to sync the progression. From where is the repository provided by Git hub. To where is any directory specified by you. Go to the Git hub repository, and copy the repository address.

Git Extensions. Click on Clone Repository and another window will open. Paste the address in the 'Repository to Clone' field and fill in the other fields. When you click 'clone', the whole project will be added into a new folder. Strangely, when cloning seems to be stuck, opening the folder (which has been already created) unstuck my progress.

Git Bash. Open Git Bash and navigate to the directory where you want to clone the project (cd [directory]). Type the following command 'git clone'. Now right-click on the toolbar of Git Bash and click on edit > paste. You full command line should now be 'git clone [repository name].git'. Press enter and Git Bash will do the rest.

What we have done is taken the latest version of the project and placed it in a new folder. This is the build taken from the Master branch (I will discuss branches later on), for now it is important to know that in principle, you never push to the Master branch but always to your current working branch, which will later be merged with the Master branch. This way, there is always a working back-up available online. More about branches in the cleverly named branches section.

You should change your username and email address in the settings > Global settings in GitExtensions (I'm not sure how, if possible, this can be changed in Git Bash), but it's very simple in Git Extensions. Your username should be your HvA username and email-address should be your HvA email.

Assuming that you already have installed a merge-tool like TortoiseGit, you can add this in Git Extensions, in the global settings.

Branches

Git works with branches, which are more or less like directories or folders within the repository. Branches make it a lot easier for teams to be working on separate tasks and merge their work together when they are finished with a certain task. They are what makes Git such a robust Source Control tool. For this project, we will be working with the developer branching model. In this model, there are two origin branches (branches that will be available for the life-span of the project), master and develop. Besides these two branches, there are also a lot of subbranches, which are only available for a given period of time. For a quick overview of how the develop model looks like, see appendix 1.

Master and developer branch.
As stated before, there is always one working build of the project online, which is the origin/Master branch. One should never commit to this branch, since it is always the latest version of the project and should be build-ready.
The origin/Developer branch has the latest delivered development changes for the next release. This branch will hold the version of the project that will also be available to the build server. Whenever a stable version of the project is available on the Developer branch it will be merged back with the Master branch and a new version of the project will be ready on the Master branch. Whenever the Developer branch is merged with the Master branch, this automatically means a new production release.

Supporting branches.
Besides these two origin branches, there can be a variety of other supporting branches, such as feature branches, hot-fix branches and release branches. These are all branches with a limited life-span, since they will be deleted again at some point. These are just normal git branches with a given set of rules as to which branches they can be originating from and which branches they will have to merged back with, they will be discussed in detail in the following section.

Feature branches.
Originates from: development;
Merges back into: development.

Feature branches will be created when a new feature is added to the project. This can be in the near future or in the distant future. The next release doesn't even have to be known yet. These branches will be deleted again when they are merged back into development and are thereby added to the next release or when the feature is no longer required. These branches only exist within the development branch.
Creating Feature branches:

Git Extensions. There are two ways to create new branches in Git Extensions, the first one is to left click in the context menu of the commit-log and select create new branch, so you can create a new branch at a specific commit. The second way is to go to Commands > Create Branch. If you check the 'checkout after create' button, the next commit will be inside of the new branch. In order to switch between branches, there are again two ways of doing so: left clicking the context menu of the commit-log and select checkout [branch name], or go to Commands > Checkout. You can view in the status bar which branch you are currently working in.

Git Bash. In Git Bash, creating a new branch is very easy. The command is git branch [branch name]. To start working in this new branch, you should always checkout that branch first. Now, we

only have created a new branch, but are still working in the branch we were working in. Checkout to the new branch. You might also create a new branch and checkout at the same time with the command git checkout -b [new branch] [originating branch]. A new branch can be created and directly accessed this way. For example git checkout -b jur Develop would create a new branch called 'jur' in de 'develop' branch, and directly move to that branch.

After a feature has been finished, it should be merged back into the development branch to definitely add it to the next release. How to merge is being handled in the 'Merge' section.

Release branches.

Originates from: development;
Merges back into: development and master;
Name convention: release-*.

Release branches are special branches that allow final improvements for the next product release, as well as bug-fixes and meta-data. By doing all of this in a special release branch, the development branch can be cleared to only merge new features.
The moment in time for these branches is when the new release is almost at the desired state. At this point, the features targeted for the next release should be added into the development branch and those that are targeted for future releases will have to wait until after the release branches have been branched off.
It is in this moment (when release branch(es) is/are created, that the development branch gets assigned a new version number. The new version number is decided by what should be included in the new release branch(es) (big release, small release, patch).
A release branch is created by the following rule of thumb: A big release is coming up and the current release version is v1.1.5. We branch off to a new release branch and bump the next release version to v1.2 (this is only decided at this moment, when a new release branch is branched off). We give the new release branch a new name that reflects the next release version. For more information, read this article.

Git Extensions. At this moment, I am not sure if this is supported in Git Extensions.

Git Bash. First, create the new branch and navigate to it: git checkout -b release-1.2 development and bump the release version to the new version number, in this case 1.2: ./bump-version.sh 1.2 Next, we are going to commit (see the push and pull section of this document).

What happened is that a new release branch was created and committed to the repository. Also, the version number was bumped to version 1.2 (this might be a manual change, but in this case, Git Flow was used to fix this, see introduction for a how-to install Git Flow). When a release branch is finished, AKA the reason for it's existence has been carried out, we merge it back into the master branch (for every merge with master also means a release). The commit to the master branch should be tagged for easy future reference and also, the release branch should be merged into the develop branch, so the next releases also include it's bug fixes and what-have-you's. This will probably lead to some conflicts, discussed in the merging section. After fixing those conflicts, we commit the changes we made, again giving a neat tag and finally, we remove the release branch (release-1.2).

Git Bash. git checkout master. We are now working in the master branch. Next thing to do is to merge with the release branch: git merge –no-ff release 1.2, adding a nice clear and summarizing tag: git tag -a 1.2. Now we have merged with the master branch. We do the same for the development branch, after which the release branch has been completely merged with the rest of the

project (after fixing the conflicts of course). Now that all these steps are completed, we remove the release branch: git branch -d release-1.2 and the release can transpire. **Hotfix branches.**

Originates from: master;
merges back into: develop and master;
name convention hotfix-*.

Hotfix branches are much like release branches in the way that they are also meant to prepare for a new production release, though they are unplanned. They should be created when a live product is in an undesired state and needs to be fixed before the new release. This can be done by creating a hotfix branch, so that the bug can be fixed from the latest master product (instead of a potential unstable develop branch), while other people are still working on merging features into the develop branch.

Git Bash. First, create a new branch that originates from the master branch: git checkout -b hotfix-1.2.1 master. Bump the whole project to that patch version: ./bump-version.sh 1.2.1 and commit it with a clear message: git commit -a -m "Bumped version number to 1.2.1". Next stop, fix the bug(s) and commit it: git commit -m "Fixed a severe bug".

After the hotfix branch has been finished, we merge it back into develop and master in the exact same way as release branches, the only exception is when we already have a release branch branched off from the develop branch, in that case, the hotfix branch should be merged into the release branch, instead of the develop branch, except when work in the develop branch is dependent on this hotfix, in which case it should be merged into the develop branch.
When the hotfix branch is merged back into the release (or develop) branch and the master branch, it is removed, again in the same way as the release branch was removed as well.

Summary branches.
Using the develop model makes it very easy for teams to work together on different features that will later be merged back into the master and develop branch.

Pushing and pulling

Today we're going to review another basic yet powerful concept that Git among other version control systems of its type has: distribution! As you may know, your commits are all local, and repositories are simply clones of each other. That means the real work in distributing your projects is in synchronizing the changes via git push and git pull.
See appendix 2 for an overview of how git pushing and pulling works.

Git Bash. First, we commit our changes locally. You might first want to add the files that have been modified, but not added yet. Use the command git status and it will show these files for the current branch. Before we commit, we should add every file that we wish to commit with the command git add <files>. This can be a complete directory: git add src/ or a single file: git add src/example.cpp/. For simplicities sake, you can use git commit -a, this will add every file that has been added before and have been modified between the last and the current commit.

When the files are added, they can than safely be pushed to the current working branch (either a feature-, release- or hotfix branch).
After the files have been pushed to the remote branch, other developers can pull your work from the branch and work with it.

Merging

When a certain task has been completed, for example when a feature has been completed, it has to be merged back into the develop branch. Merging branches itself is fairly easy: all that needs to be done is go into the branch that needs to be merged, in this case develop. Before we do this, we should commit and push the branch that should be merged. See the push and pull section on how to to this. After this, we checkout to the develop branch and merge the two branches:

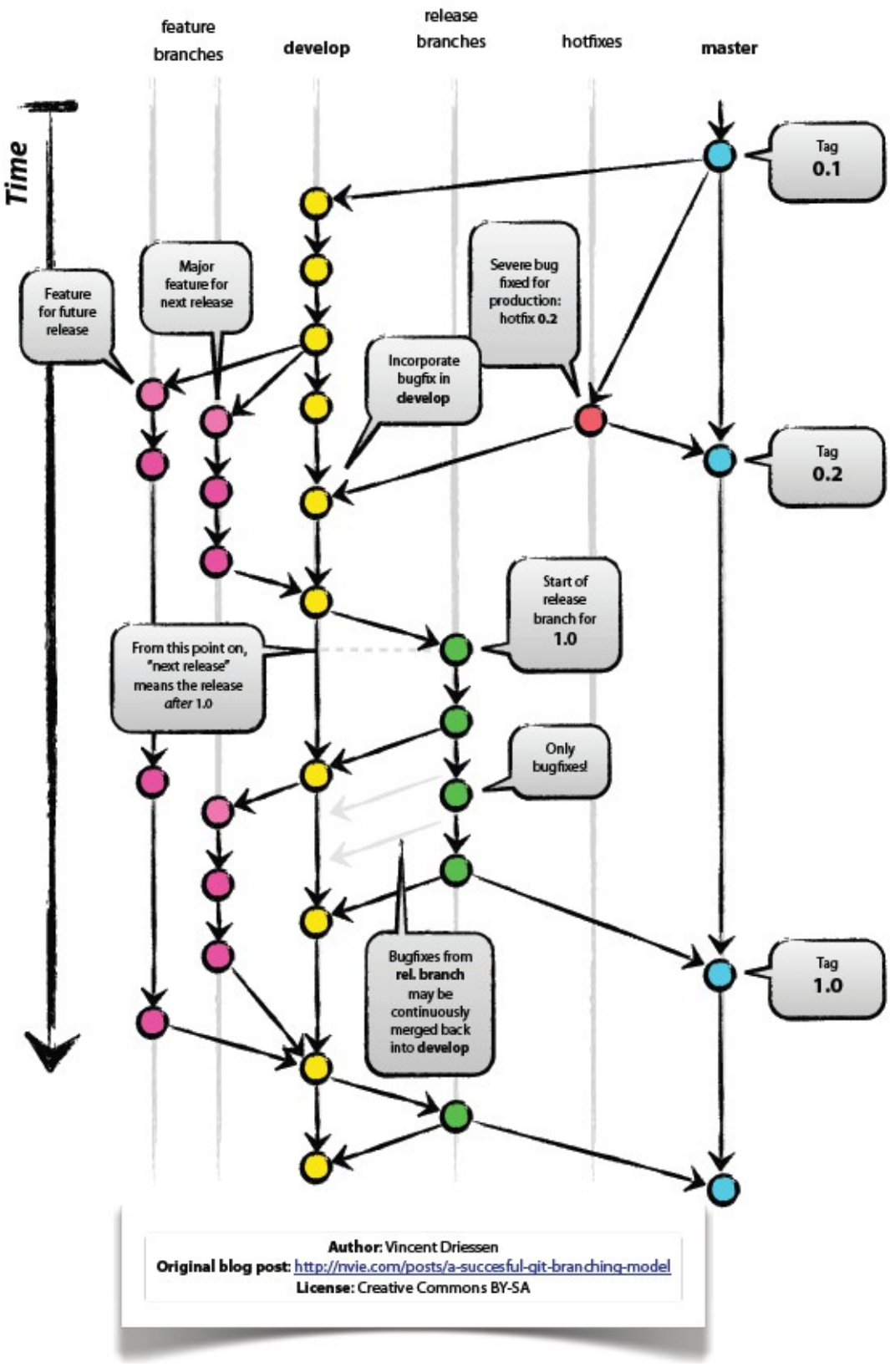
Git Bash. (while working in the feature branch) git checkout develop -switches to the develop branch. The next thing that needs to be done (assuming that the feature branch has been completely pushed to the remote repository) is to merge the two branches with the command
git merge --no-f <branch name>. By using this merge command, a new commit object is created, which avoids losing the historical existence of a branch and groups together all commits that were added to the feature. For an example of what the difference in between these two different uses of merge, see appendix 3.

Whenever a merge has been completed, there is always the chance that some conflicts arise. This means that git has two different versions of a file and doesn't know which one to use. In this case you can either manually change the files that give a conflict – which is a pain – or you can use a merge tool, such as TortoiseGit, which I will be using. After a merge and conflicts do arise, open the merge tool:

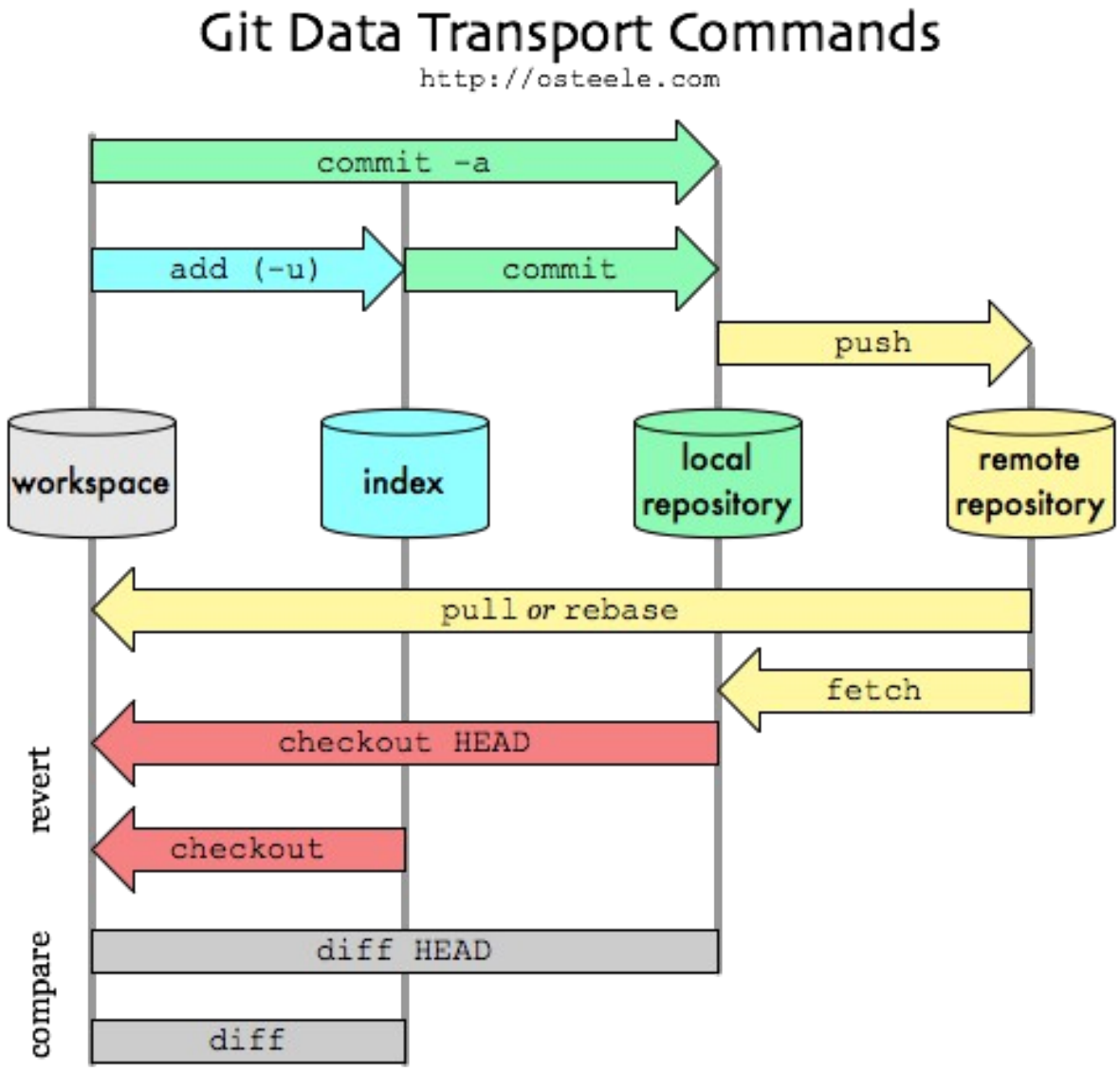
Git Bash. Open the merge tool with the command git mergetool. This will open the the merge tool options screen and you can hit enter for every conflicted file. You should always resolve conflicts before working on with files, for they might cause very nasty bugs and error throughout the project.

Appendix

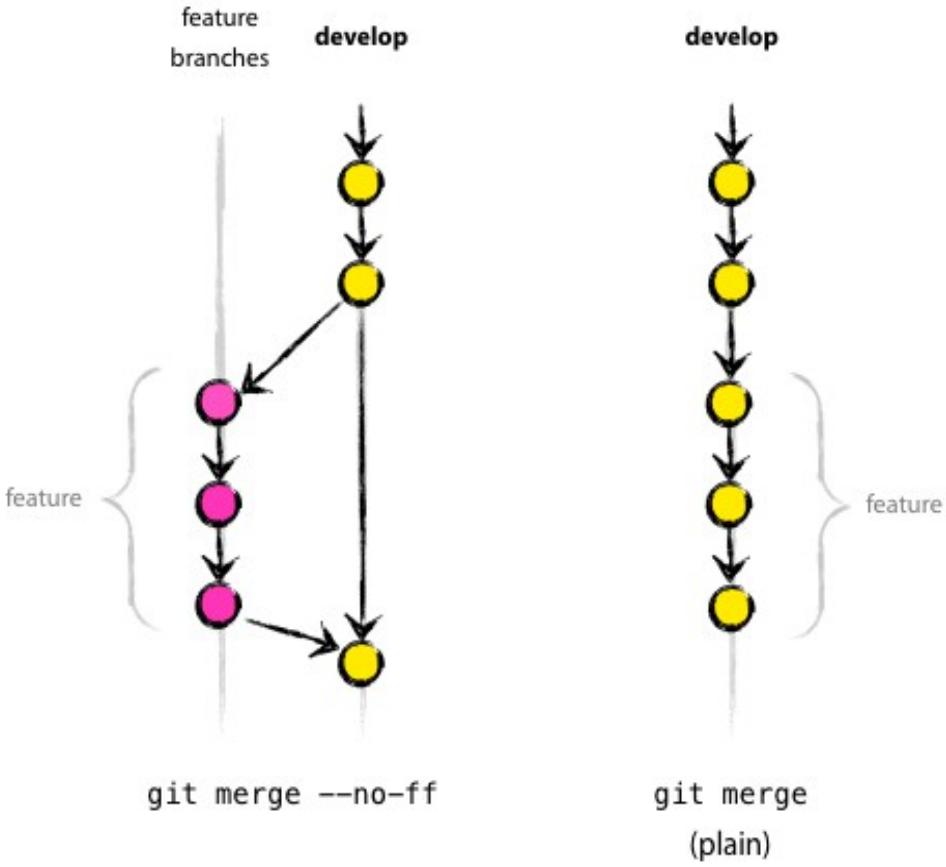
1


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
Author: Vincent Driessen
Original blog post: http://nvie.com/posts/a-succesful-git-branching-model
License: Creative Commons BY-SA

2


Git Data Transport Commands
http://osteele.com
commit -a
add (-u)
commit
push
workspace
index
local repository
remote repository
pull or rebase
fetch
checkout HEAD
checkout
diff HEAD
diff
revert
compare

3


feature
branches
develop
feature
git merge --no-ff
develop
feature
git merge
(plain)