Razorfish Healthware

GitHub Workflow

Version 1.0



**Table of Contents**

[1. About this document 4](#_Toc368648543)

[Version Control 4](#_Toc368648544)

[Distribution List 5](#_Toc368648545)

[Reference Documents 6](#_Toc368648546)

[2. Introduction 7](#_Toc368648547)

[3. Github Workflow 8](#_Toc368648548)

[A quick example 8](#_Toc368648549)

[Cloning the repository 8](#_Toc368648550)

[Reverting changes 9](#_Toc368648551)

[Confirming changes 9](#_Toc368648552)

[Creating a local branch 9](#_Toc368648553)

[Listing branches 9](#_Toc368648554)

[Checking out branches 10](#_Toc368648555)

[Shortcut for creating and checking out branches 10](#_Toc368648556)

[Committing changes 10](#_Toc368648557)

[Merging branches 10](#_Toc368648558)

[Deleting branches 10](#_Toc368648559)

[Pushing changes 11](#_Toc368648560)

[Wrapping up 11](#_Toc368648561)

[4. Branching model 12](#_Toc368648562)

[Branch types 12](#_Toc368648563)

[5. Detailed workflow 13](#_Toc368648564)

[Master branch 13](#_Toc368648565)

[Creating a new GitHub repository 13](#_Toc368648566)

[Develop branch 16](#_Toc368648567)

[Feature branches 17](#_Toc368648568)

[Release branches 19](#_Toc368648569)

[Hotfix branches 20](#_Toc368648570)

[6. Multiple developer workflow 22](#_Toc368648571)

[Cloning the remote GitHub repository 22](#_Toc368648572)

[Ongoing development 22](#_Toc368648573)

[Cloning remote repository or updating working directory 23](#_Toc368648574)

[Creating feature branch 23](#_Toc368648575)

[Commiting new work 23](#_Toc368648576)

[Updating working directory 23](#_Toc368648577)

[Resolving merge conflicts 23](#_Toc368648578)

[Pushing to remote repository 23](#_Toc368648579)

[Issuing pull request 24](#_Toc368648580)

[7. Advanced and useful Git tools 25](#_Toc368648581)

[Changing the Last Commit 25](#_Toc368648582)

[Changing Multiple Commit Messages 26](#_Toc368648583)

[Reordering Commits 27](#_Toc368648584)

[Squashing Commits 28](#_Toc368648585)

[Splitting a Commit 28](#_Toc368648586)

[8. Appendix 1 – Glossary 30](#_Toc368648587)

# About this document

This document outlines the source control workflow based on git and GitHub and that is being used in Razorfish Healthware Spain.

This workflow has been defined and refined based on various references mentioned below, and is open to further improvements and modifications.

## Version Control

The following is a list all changes made to this document, the person making the change, the new version number and the reason why the change was necessary.

| Date | Author | Version | Change Description |
| --- | --- | --- | --- |
| 27/09/13 | Rubén Gil | 0.1 | Draft version |
| 02/10/13 | Rubén Gil | 0.2 | Updated following review |
| 03/10/13 | Paul Chadwick | 0.3 | Document tidy up |
| 03/10/13 | Victor Cruz | 0.4 | Added advanced git tools section |
| 04/10/13 | Rubén Gil | 0.5 | Added complex workflow example |
| 04/10/13 | Paul Chadwick | 0.6 | Document tidy up |
| 04/10/13 | Paul Chadwick | 1.0 | Release version |

## Distribution List

The following list contains the names and organisational details of the people to whom this document has been distributed.

| Name | Organisation | Position |
| --- | --- | --- |
| Víctor Cruz | Razorfish Healthware | Technical Architect |
| Rubén Gil | Razorfish Healthware | Senior Technical Architect |
| Enrique Cruz | Razorfish Healthware | Senior Developer |
| Paul Chadwick | Razorfish Healthware | VP, Technology (Europe) |
| Vipul Patel | Razorfish Healthware | Technical Architect |
| Angelo Nutile | Razorfish Healthware | Senior Technical Architect |
| Lino Mari | Razorfish Healthware | Senior Technical Architect |
| Francesco Raimondo | Razorfish Healthware | Presentation Layer Architect |
| Fabio Spacagna | Razorfish Healthware | Senior Technical Architect |
| Jeff Smith | Razorfish Healthware | Global Head of Technology |

## Reference Documents

The workflow described in this document is based on the following references.

| Document | Url |
| --- | --- |
| Pro Git (Online book) | <http://git-scm.com/book> |
| A successful Git branching model | http://nvie.com/posts/a-successful-git-branching-model/ |
| Contributing to Ruby on Rails | http://edgeguides.rubyonrails.org/contributing\_to\_ruby\_on\_rails.html |

# Introduction

This document is split in to a number of sections. In the first one, we will go through a workflow example and we will see the git source control system’s most important features and usage.

In the second and third section we will review a branching model that can be used in any kind of projects and that we have successfully implemented in Razorfish Healthware Spain.

The fourth section we describe how multiple developers could work on independent braches and how to manage conflicts when merging.

All examples described in this guide to interact with the git source control system are based on a Unix terminal executed on Mac OSX 10.8.5 (Mountain Lion).

We are using ssh GitHub urls, so the user must properly configure ssh keys in his computer for the examples to work.

There are a number of GUI tools that have been developed to work with GitHub on different operating systems. It is advised that you first understand the principles of GitHub and the git workflow described in this document then apply that knowledge to the tool you use.

A glossary of git terms is provided at the end of this document to help your understanding of the common commands and features of git.

# Github Workflow

## A quick example

First we will see a common workflow that includes the basic commands used on a day to day basis.

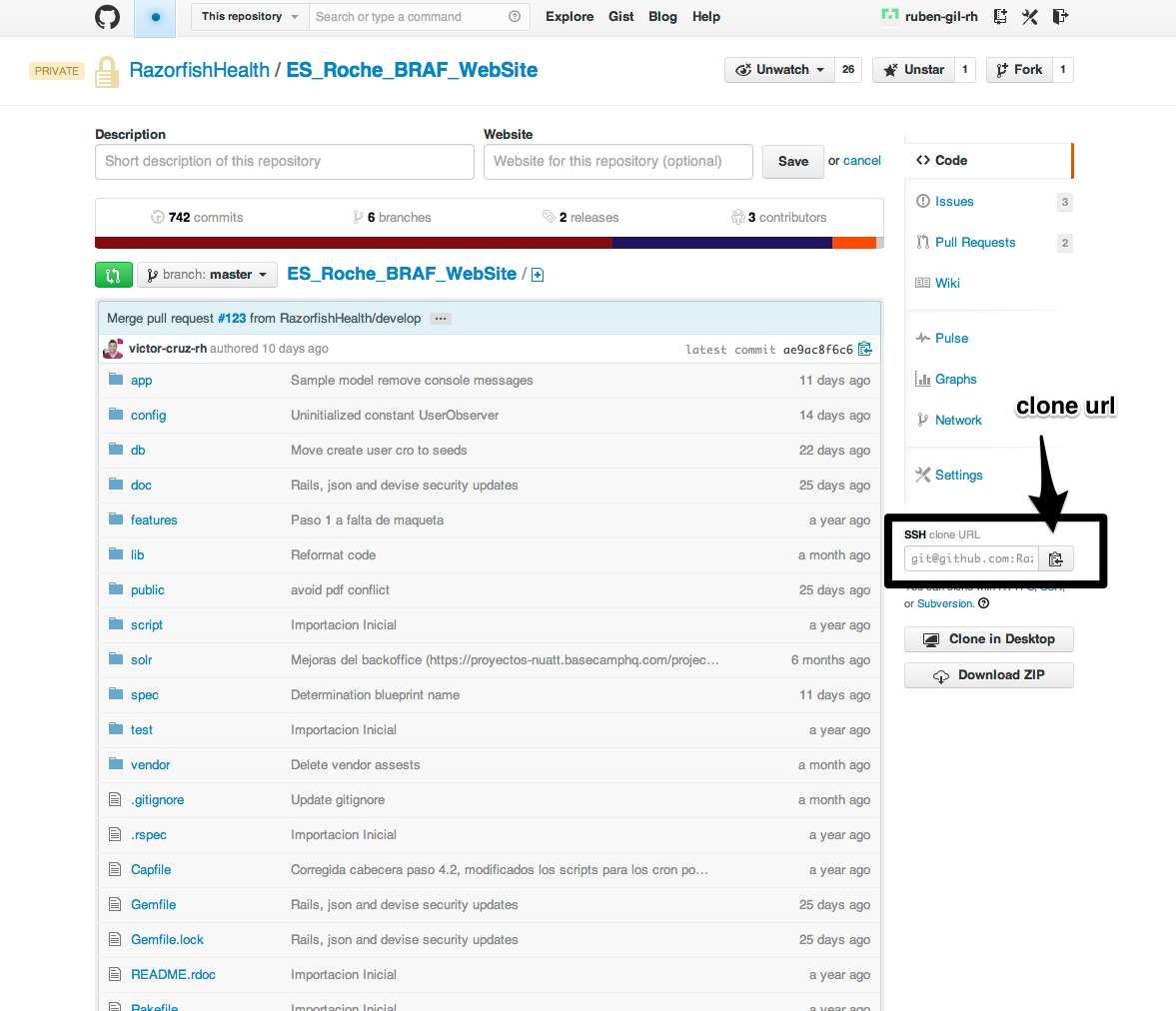
For this example we will work with a project named ES\_Roche\_BRAF\_WebSite.

## Cloning the repository

The first step is cloning the repository to your local machine:

git clone [git@github.com:RazorfishHealth/ES\_Roche\_BRAF\_WebSite.git](mailto:git@github.com:RazorfishHealth/ES_Roche_BRAF_WebSite.git)

To obtain the clone url, go to the GitHub project page and click the button ‘Clone url’



This command will create the project folder ES\_Roche\_BRAF\_Website/ in the directory where we have executed the command.

Now we can proceed to work with the project locally.

Lets suppose that we modify the file ES\_Roche\_BRAF\_WebSite/Gemfile with our text editor or IDE.

## Reverting changes

In case we want to revert the changes we have made to the file and return to the initial state, we will issue the following command:

git checkout Gemfile

## Confirming changes

In case we want to confirm the changes, we will stage the file with the command:

git add Gemfile

Staged files are the ones that will be commited with the commit git command.

## Creating a local branch

It's a recommended practice to work in local branches to implement new features, and not directly in the master branch.

Let's suppose that we need to add a new line to our Gemfile file. For this we will create a new branch named add-cucumber:

git branch add-cucumber-to-gemfile

Now we have created a new branch named add-cucumber-to-gemfile that we will use to work on the change described above.

## Listing branches

To list all branches in our local git repository, we will issue the following command.

git branch

## Checking out branches

Now that we have created our new branch we need to check it out to begin working with it. We will do so with the following command:

git checkout add-cucumber-to-gemfile

We can check that new this branch is the current branch with the command git branch. We can see a trailing asterisk in the current branch.

## Shortcut for creating and checking out branches

As creating a new branch and checking it out are processes that are normally executed in a row, we can use the following shortcut to execute the two processes at once:

git checkout -b add-cucumber-to-gemfile

Now we can edit the GemFile file and add the following line at the end:

gem cucumber.

With this change we are done, and can proceed committing our changes.

## Committing changes

To commit all the changes into our current branch, we will issue the following command:

git commit -m 'Added cucumber gem'

## Merging branches

Finally we will merge (integrate) the changes into our master branch, before we push (upload) the changes to our remote GitHub repository.

git checkout master

git merge add-cucumber-to-gemfile

## Deleting branches

Now the contents of our master and add-cucumber-to-gemfile branches are identical, so we can proceed to remove the add-cucumber-to-gemfile branch, as once we have developed our feature we won’t use it anymore.

git branch -d add-cucumber-to-gemfile

## Pushing changes

Now, we are ready to push our local changes to a remote repository, so we can share our work with the rest of the team:

git push remote origin

## Wrapping up

In this section we have reviewed the basic actions of a GitHub workflow, that include:

* Cloning a remote GitHub repository.
* Reverting local changes
* Creating branches
* Checking out branches
* Commiting
* Deleting branches
* Pushing to a remote repository

With this foundation we are ready to go on to the following section, where we will see a branching model that will allow us to organize our development releases.

# Branching model

We will review the branching model we are using in our development and that is based on the model described in the document http://nvie.com/posts/a-successful-git-branching-model/.

## Branch types

The proposed branching model is based on the following branch types.

1. **Master Branch** - Code in the master branch reflects a production-ready state.
2. **Develop Branch** - Contains the later delivered changes for new developed functionalities. Integration testing for these features should be performed on this branch.
3. **Feature Branch** - For every feature developed a feature branch should be created from develop on the GitHub server. This branch will be merged in develop branch through a Pull Request.
4. **Hotfix Branch** - If a bug in production arises, a hotfix branch will be created from master. Once the bug is fixed, the branch will be merged into master.
5. **Release Branches** - When a release to production is close, a release branch will be created. In this branch will include fixes and improvements arisen during the Quality Assurance process.

# Detailed workflow

Now we will see a detailed workflow to show how to work with the different types of branches described above.

In this section, we will work with a new example project named github\_example.

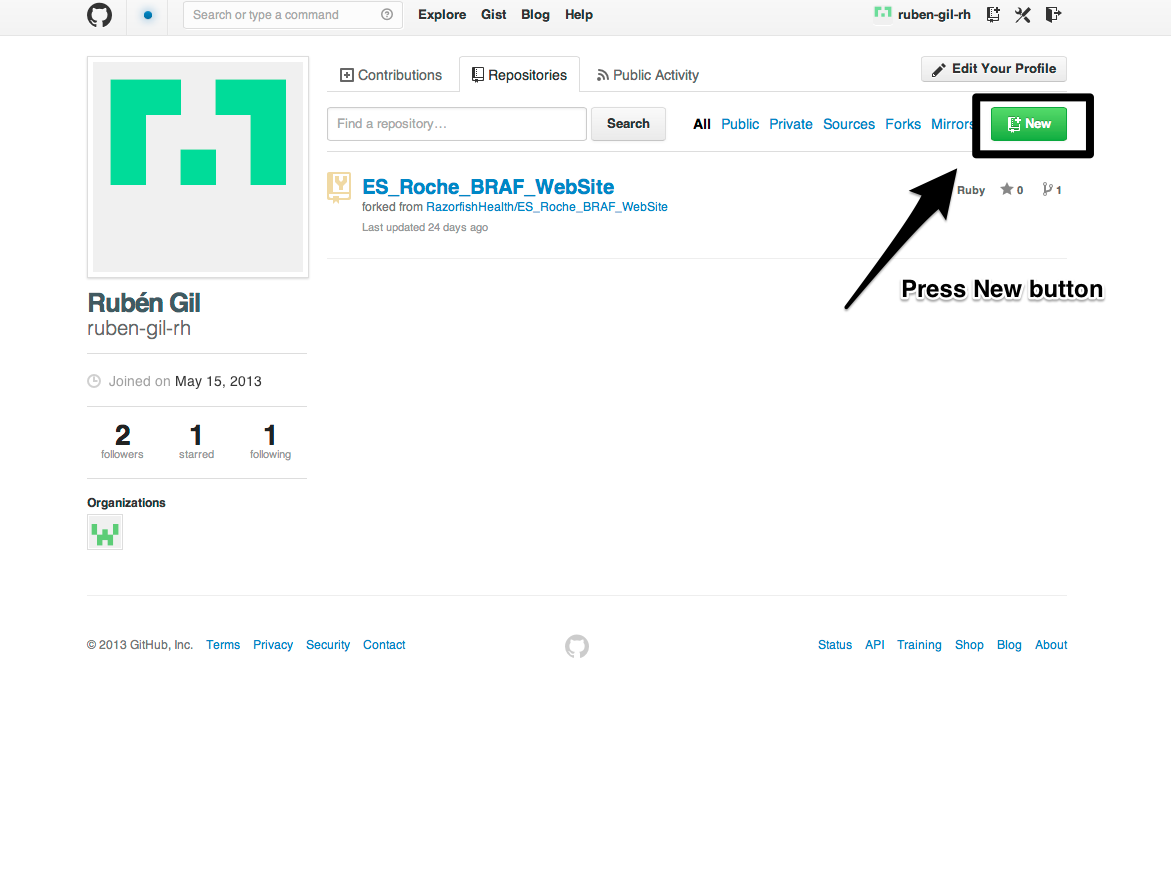
## Master branch

The master branch is the default branch and the one that is created automatically when we create a new git repository.

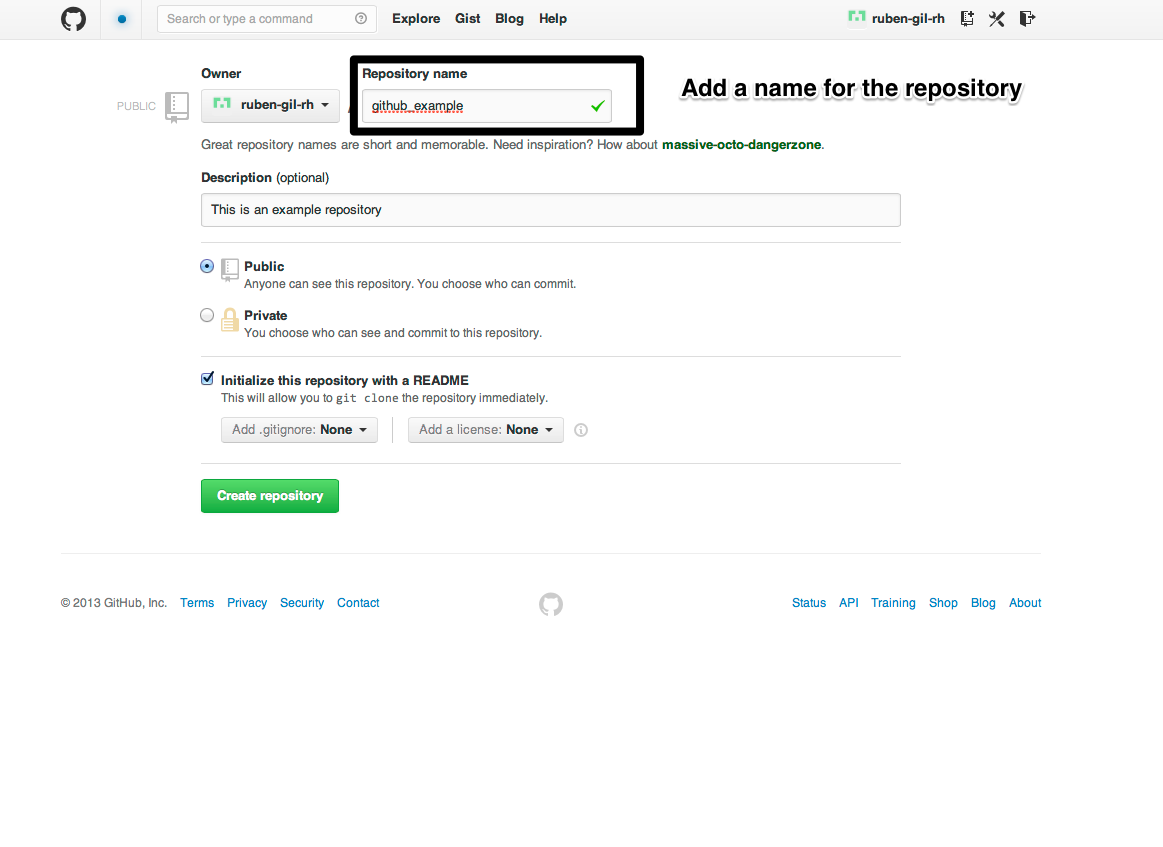
Let's suppose that we are beginning the project from scratch.

### Creating a new GitHub repository

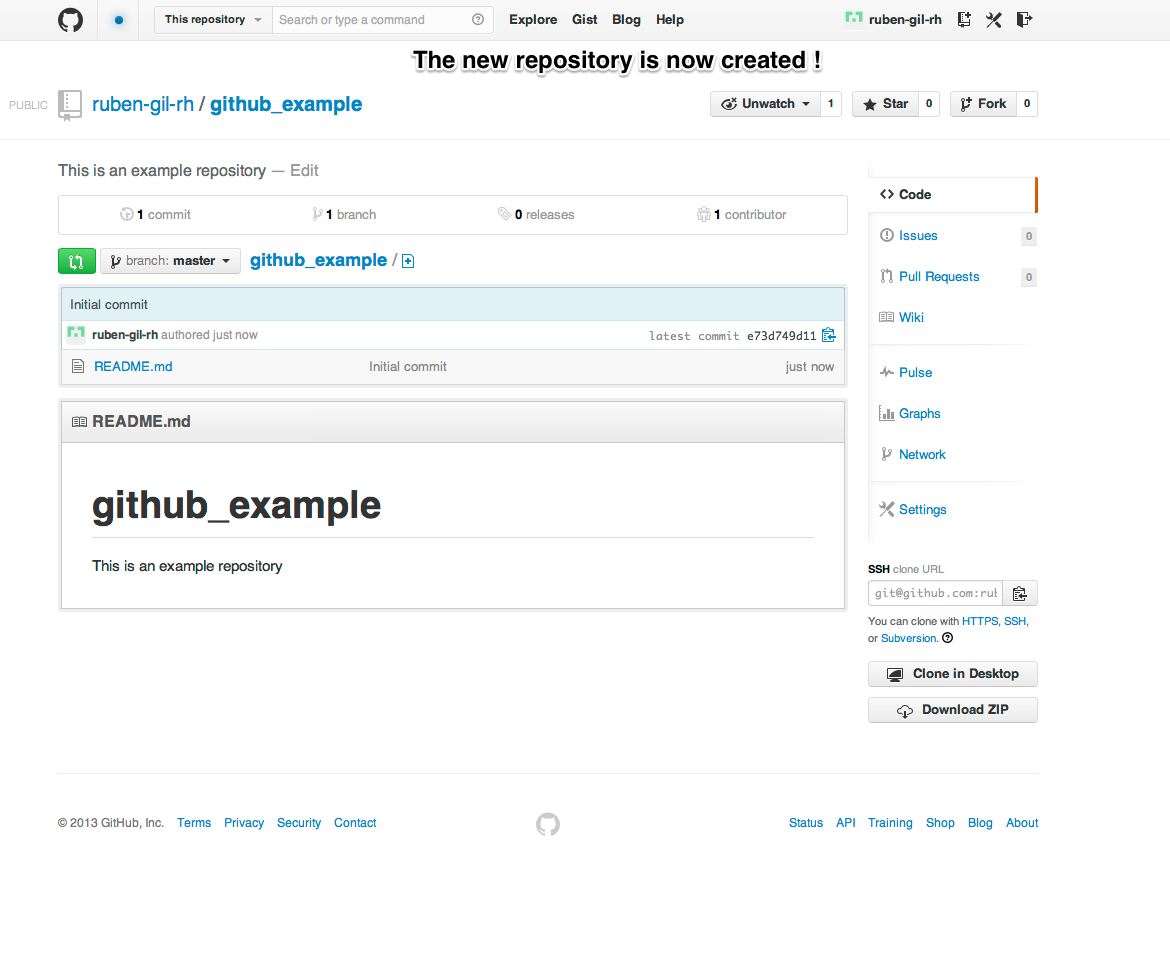
To create a new GitHub repository we will first go to the url github.com and login with our user and password. Then we will click the 'New' button under the 'Repositories' tab



Then, we will choose a name for the repository and will click the 'Create repository' button



We will see our new created repository (we have chosen to initialize it with a README file)



In case we are working in a new project and don't need to add existing local files, we will clone the repository in our local machine and begin developing on it.

For this we need to copy the clone url from our repository and issue the following command:

git clone git@github.com:ruben-gil-rh/github\_example.git

This will create a github\_example directory cloning the remote repository.

From this point on we can begin developing on our master branch.

In case we have code that we want to add to our new created repository, we will first initialize a local repository:

cd github\_example

git init

Here we have moved to the project folder and initialized a local git repository. As explained before, this new repository has a single master branch, as we can see with the following command:

git status

Now we will connect the local and remote repository (we will use the same clone url as before)

git remote add origin [git@github.com](mailto:git@github.com):ruben-gil-rh/github\_example.git

Finally we will add a new file to the repository, commit changes and push changes into the master branch to the server:

touch my\_first\_file

git add my\_first\_file

git commit 'Add my\_first\_file'

git push origin master

With this command we are pushing to the remote repository named origin the branch named master.

In this case, as origin is GitHub default remote repository, and master is the default branch we could have used the command:

git push

## Develop branch

The develop branch is the one in which we will integrate the features as long as they are developed, and will deploy them in an integration server. Usually the develop branch is forked from the master branch, and the creation process is like this.

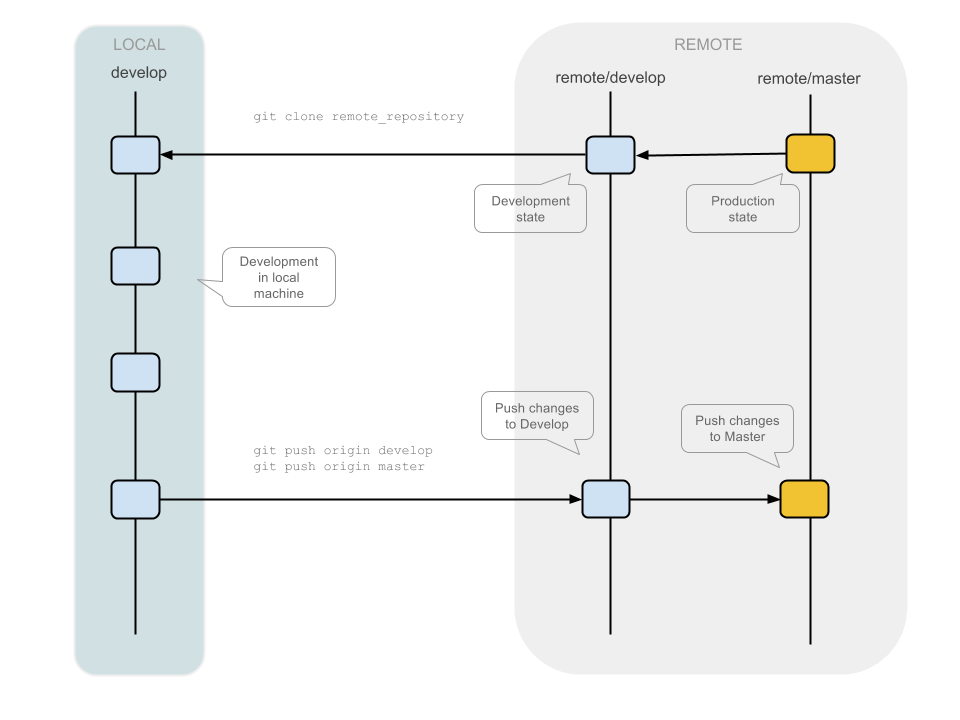
git checkout -b develop

git push origin develop

We have basically created a local branch named develop and pushed it to the remote GitHub server.

From now on, all development should be performed on this develop branch that will integrate the new features.

Once we are ready to deploy in production we will merge the contents of the development branch into master.



## Feature branches

The development of new features will be performed against feature branches. This branches will be forked from development (remember that the only branches that can be forked from master are hotfix ones), and will be merged back to development.

The process is as follows:

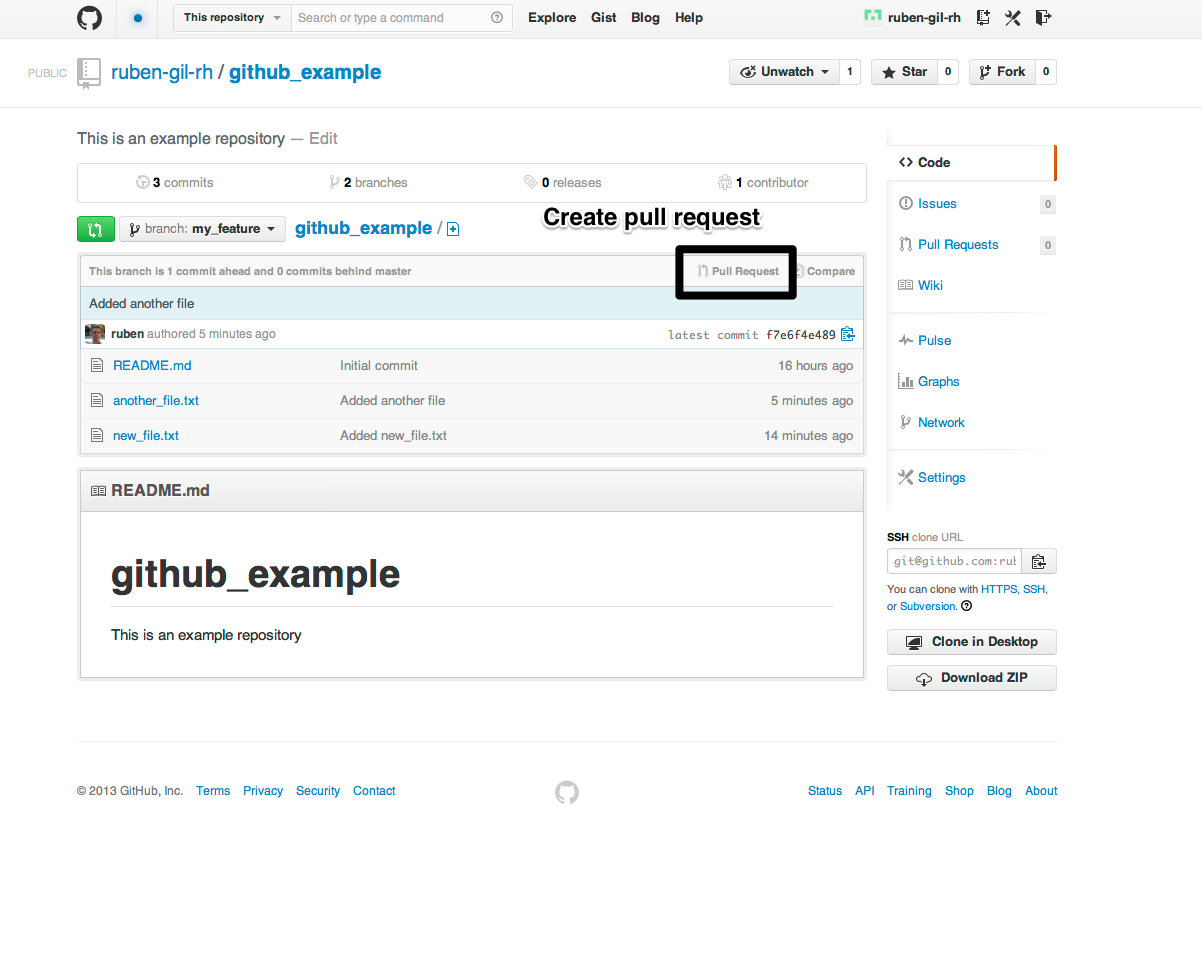
git checkout development

git checkout -b my\_new\_feature

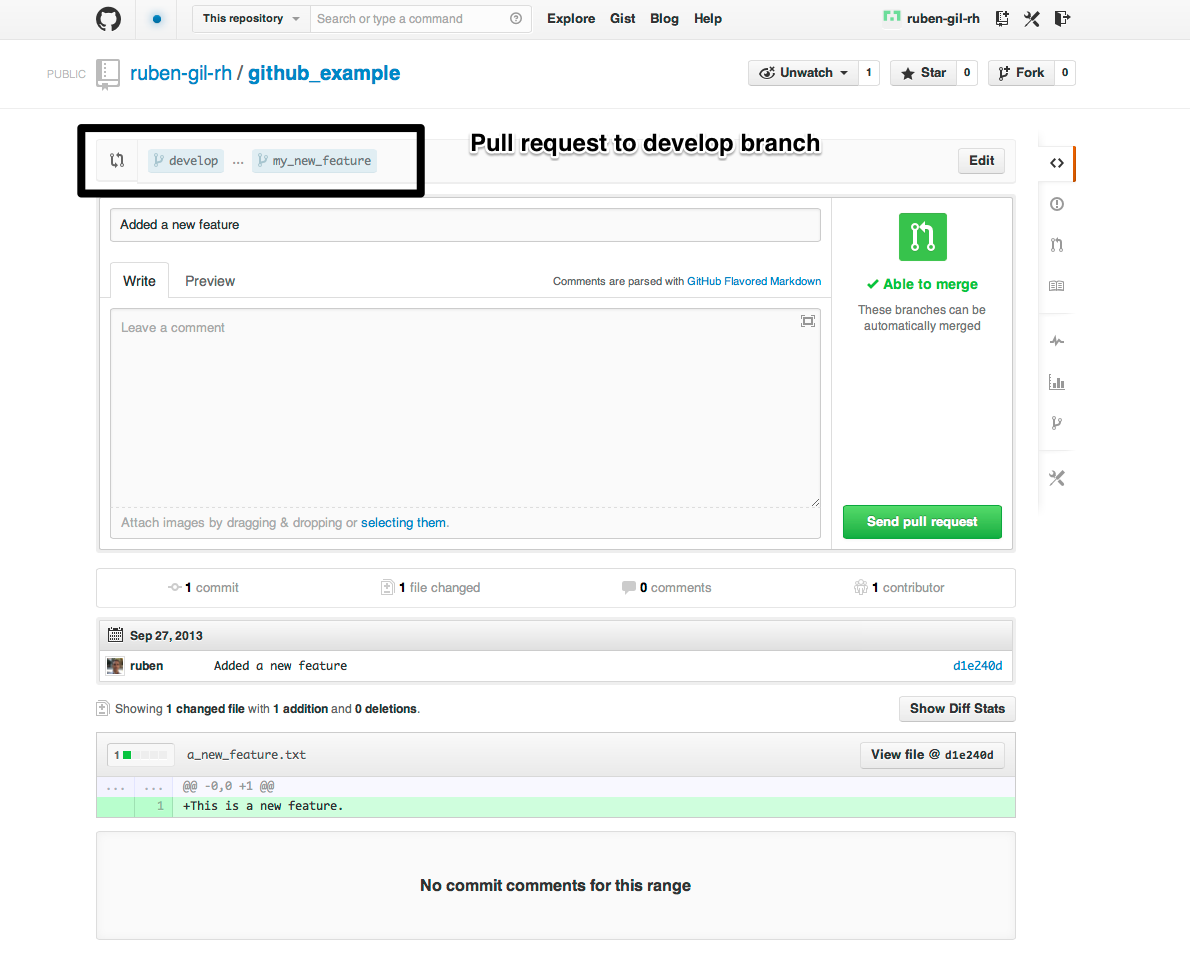
Here we will develop and test our new feature, and once is finished we will merge it back on development:

git push origin my\_new\_feature

Finally we will create a pull request of new\_feature branch into development.



It is very important that the pull request be to development branch



## Release branches

Once we are near the release, and we consider that all the features of the new version are present in the product, we will create a release branch.

This release will be deployed against a QA server. No new features related with new releases must be added into this branch. Only fixes related with the QA process.

The flow is like follows:

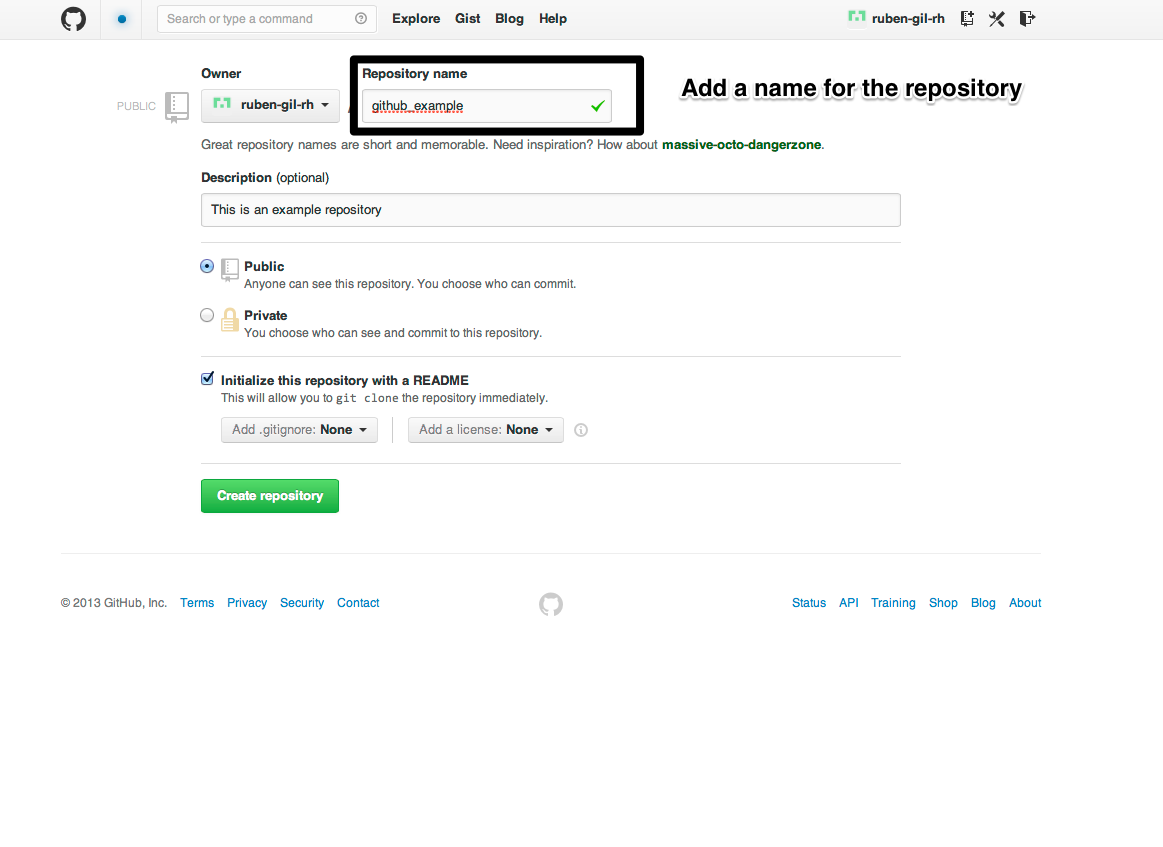
git checkout development

git checkout -b v2.0.rc1

git push origin v2.0.rc1

From this point, all feature branches must be forked and merged to this release branch.

Once the QA process has finished and we are ready to deploy to production, we will create a pull request against the master branch and the development branch.



Finally, we will tag the master version:

git tag -a 2.0

## Hotfix branches

To solve bugs in the current production release, we will create hotfix branches forked from master branch. This way we won't mess up with ongoing development. As said above, hotfix branches are forked from master, and merged into master and development.

The process is as follows:

git checkout master

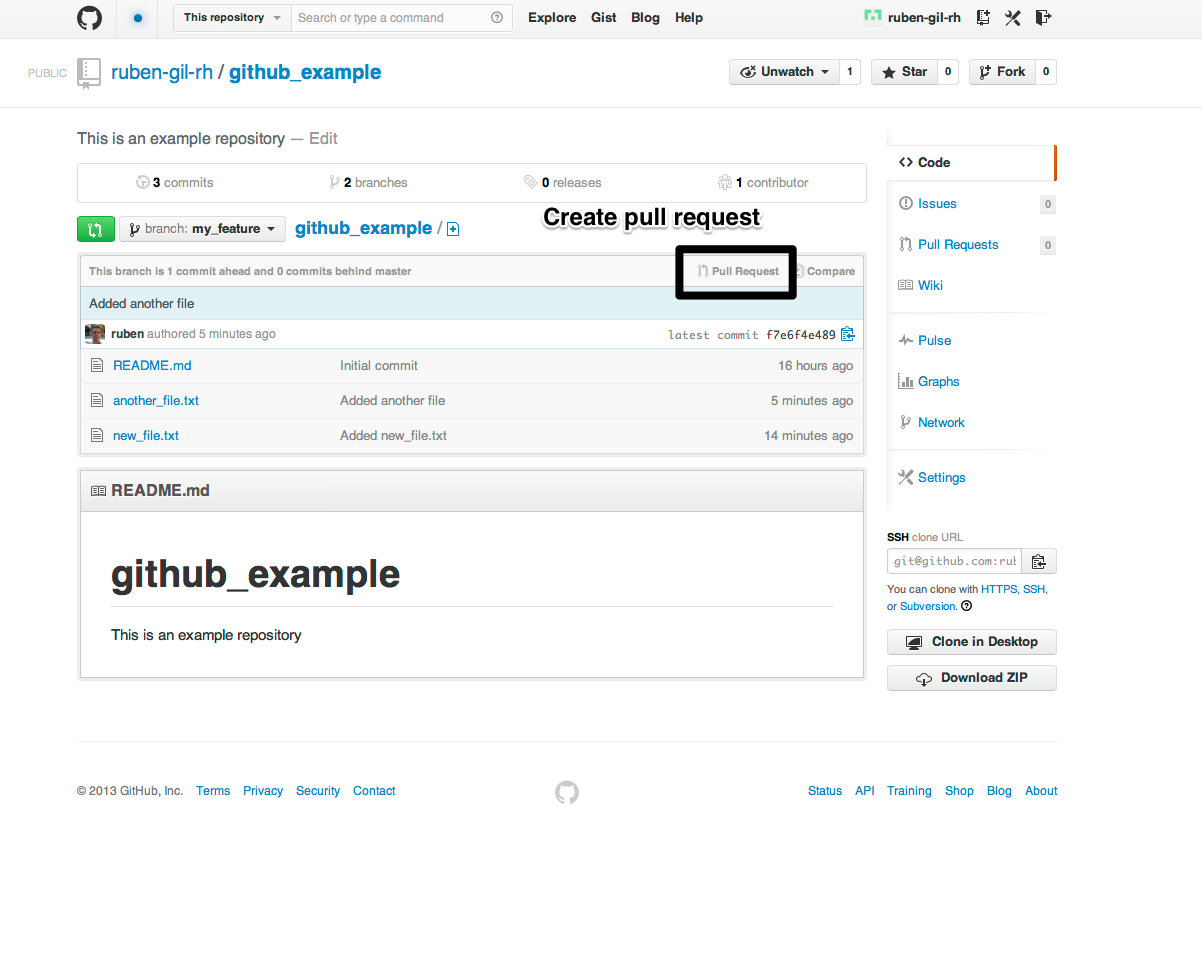
git checkout -b performance-hotfix

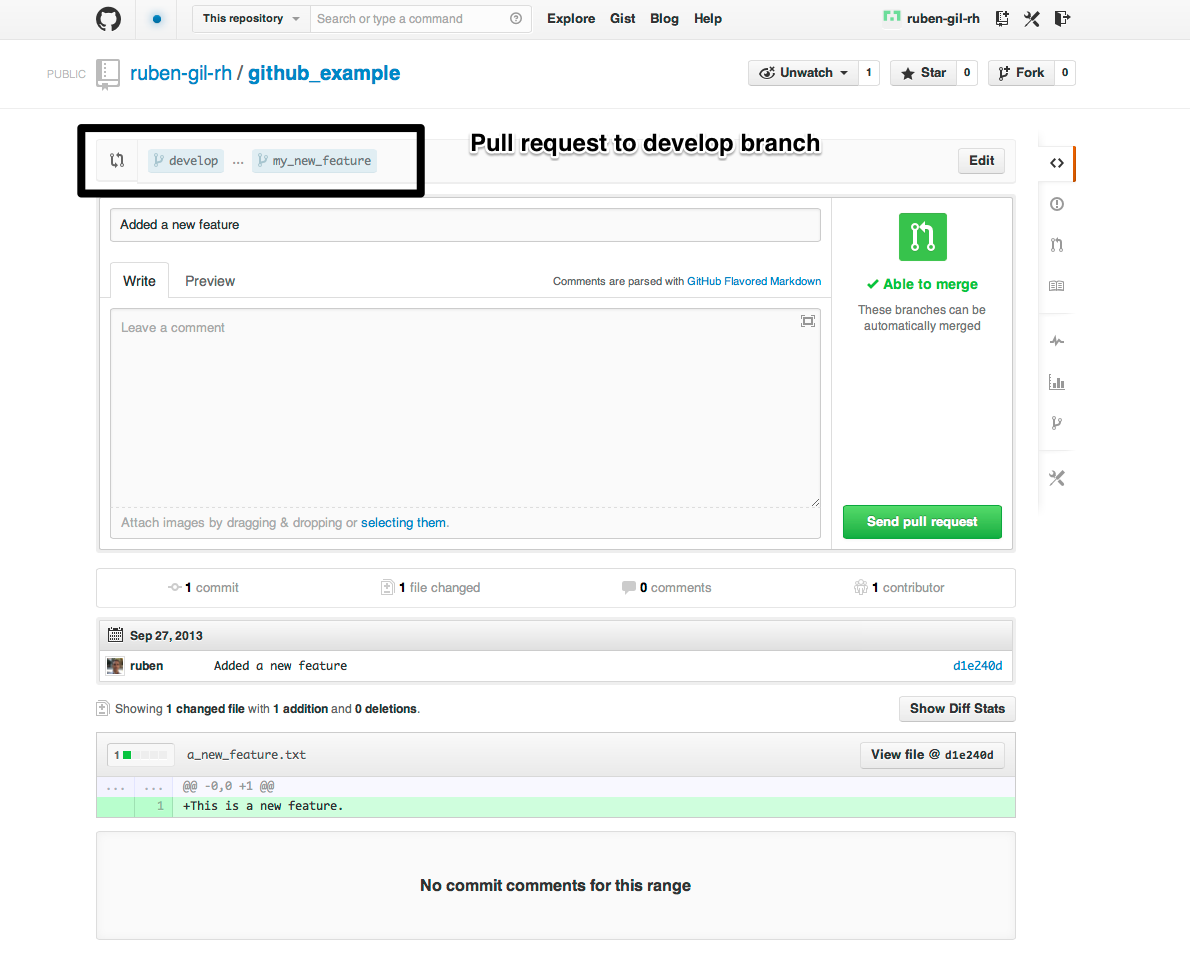
Here we will develop and test our new feature, and once is is finished we will merge it back on master:

git push origin performance-hotfix

From here, we will create a pull request against master and develop branches.

Finally we will create a pull request of performance-hotfix into master and development.





# Multiple developer workflow

In this section we will describe a more complex workflow in which we have many concurrent developers working in many features.

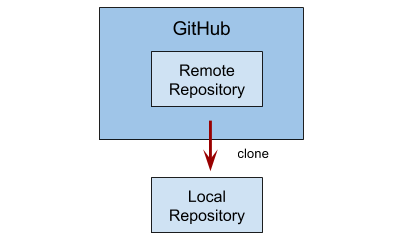
We will see how to clone or update the remote repository, working in local branches and resolving merge conflicts.

## Cloning the remote GitHub repository

Every developer will begin by cloning the remote GitHub repository to his/her local machine:

git clone git@github.com:RazorfishHealthware/github\_example.git

cd github\_example



## Ongoing development

The basic developer workflow is the following:

* Cloning remote repository or updating working directory with latest remote changes (depends if the user begins working in this project or had already cloned the remote repository).
* Creating feature branch
* Commiting work
* Updating working directory with other developers work.
* Resolving merge conflicts
* Rebasing other developers work
* Pushing to remote branch
* Issuing Pull request

### Cloning remote repository or updating working directory

Developer clones remote repository in case he is new in the project, or updates working directory with latest remote changes.

git clone git@github.com:RazorfishHealthware/github\_example.git

git checkout develop

git pull

### Creating feature branch

Developer creates local feature branch and begins working in the feature.

git checkout -b my\_new\_feature

### Commiting new work

Developer adds and commits files to the local git branch.

git add .

git commit -m "My new feature"

### Updating working directory

Developer updates working directory with the latest remote changes.

git checkout develop

git pull --rebase

And applies those changes to the local branch

git checkout my\_new\_feature

git rebase develop

### Resolving merge conflicts

If case there are conflicts during the rebase, the conflicts will be tagged in the files with:

<<<<<<< HEAD

# Commits from Develop

=======

# Commits from <my\_new\_feature>

>>>>>>> "My New feature"

The developer correct these conflict in your code text editor, then add them and continue the rebase:

git add -u

git rebase --continue

### Pushing to remote repository

Developer pushes new branch to remote. The branch gets created in the remote repository and is ready for a pull request to remote develop branch.

git checkout my\_new\_feature

git push

### Issuing pull request

From GitHub interface, developer issues a Pull Request to develop branch.

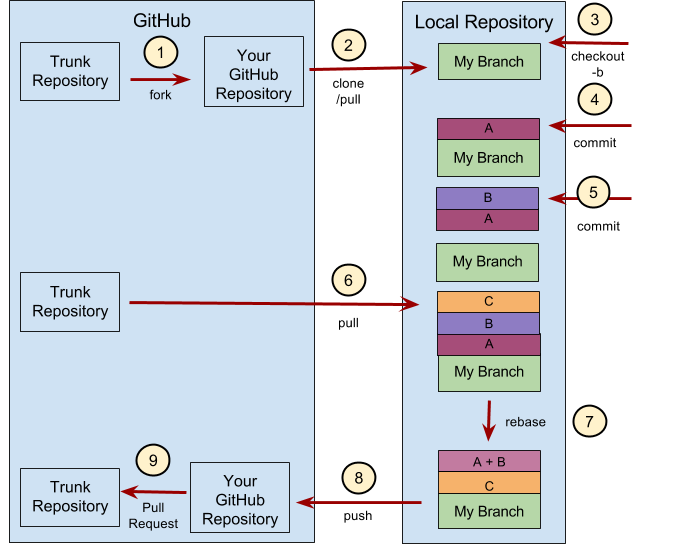


Diagram showing a complex developer workflow.

# Advanced and useful Git tools

Many times, when working with Git, you may want to revise your commit history for some reason. One of the great things about Git is that it allows you to make decisions at the last possible moment. You can decide what files go into which commits right before you commit with the staging area, you can decide that you didn’t mean to be working on something yet with the stash command, and you can rewrite commits that already happened so they look like they happened in a different way. This can involve changing the order of the commits, changing messages or modifying files in a commit, squashing together or splitting apart commits, or removing commits entirely — all before you share your work with others.

In this section, you’ll cover how to accomplish these very useful tasks so that you can make your commit history look the way you want before you share it with others.

IMPORTANT NOTE: You need to be careful with these techniques because them can change the SHA-1 of the commit, the commit’s order, etc. So, as general rule, you shouldn’t apply them to a branch if you’ve already pushed it. Or if you have pushed it, but no one is working with the same branch, you can do and then force a push to update the remote branch:

git push origin my\_branch –force

or

git push origin +my\_branch

## Changing the Last Commit

Changing your last commit is probably the most common rewriting of history that you’ll do. You’ll often want to do two basic things to your last commit: change the commit message, or change the snapshot you just recorded by adding, changing and removing files.

If you only want to modify your last commit message, it’s very simple:

git commit --amend

That drops you into your text editor, which has your last commit message in it, ready for you to modify the message. When you save and close the editor, the editor writes a new commit containing that message and makes it your new last commit.

If you’ve committed and then you want to change the snapshot you committed by adding or changing files, possibly because you forgot to add a newly created file when you originally committed, the process works basically the same way. You stage the changes you want by editing a file and running git add on it or git rm to a tracked file, and the subsequent git commit --amend takes your current staging area and makes it the snapshot for the new commit.

You need to be careful with this technique because amending changes the SHA-1 of the commit. It’s like a very small rebase — don’t amend your last commit if you’ve already pushed it.

## Changing Multiple Commit Messages

To modify a commit that is farther back in your history, you must move to more complex tools. Git doesn’t have a modify-history tool, but you can use the rebase tool to rebase a series of commits onto the HEAD they were originally based on instead of moving them to another one. With the interactive rebase tool, you can then stop after each commit you want to modify and change the message, add files, or do whatever you wish. You can run rebase interactively by adding the –i option to git rebase. You must indicate how far back you want to rewrite commits by telling the command which commit to rebase onto.

For example, if you want to change the last three commit messages, or any of the commit messages in that group, you supply as an argument to git rebase -i the parent of the last commit you want to edit, which is HEAD~2^ or HEAD~3. It may be easier to remember the ~3 because you’re trying to edit the last three commits; but keep in mind that you’re actually designating four commits ago, the parent of the last commit you want to edit:

git rebase -i HEAD~3

Remember again that this is a rebasing command — every commit included in the range HEAD~3..HEAD will be rewritten, whether you change the message or not. Don’t include any commit you’ve already pushed to a central server — doing so will confuse other developers by providing an alternate version of the same change.

Running this command gives you a list of commits in your text editor that looks something like this:

pick f7f3f6d changed my name a bit

pick 310154e updated README formatting and added blame

pick a5f4a0d added cat-file

# Rebase 710f0f8..a5f4a0d onto 710f0f8

#

# Commands:

# p, pick = use commit

# e, edit = use commit, but stop for amending

# s, squash = use commit, but meld into previous commit

#

# If you remove a line here THAT COMMIT WILL BE LOST.

# However, if you remove everything, the rebase will be aborted.

#

It’s important to note that these commits are listed in the opposite order than you normally see them using the log command. If you run a log, you see something like this:

git log --pretty=format:"%h %s" HEAD~3..HEAD

a5f4a0d added cat-file

310154e updated README formatting and added blame

f7f3f6d changed my name a bit

Notice the reverse order. The interactive rebase gives you a script that it’s going to run. It will start at the commit you specify on the command line (HEAD~3) and replay the changes introduced in each of these commits from top to bottom. It lists the oldest at the top, rather than the newest, because that’s the first one it will replay.

You need to edit the script so that it stops at the commit you want to edit. To do so, change the word pick to the word edit for each of the commits you want the script to stop after. For example, to modify only the third commit message, you change the file to look like this:

edit f7f3f6d changed my name a bit

pick 310154e updated README formatting and added blame

pick a5f4a0d added cat-file

When you save and exit the editor, Git rewinds you back to the last commit in that list and drops you on the command line with the following message:

git rebase -i HEAD~3

Stopped at 7482e0d... updated the gemspec to hopefully work better You can amend the commit now, with

git commit --amend Once you’re satisfied with your changes, run git rebase --continue

These instructions tell you exactly what to do. Type

git commit --amend

Change the commit message, and exit the editor. Then, run

git rebase --continue

This command will apply the other two commits automatically, and then you’re done. If you change pick to edit on more lines, you can repeat these steps for each commit you change to edit. Each time, Git will stop, let you amend the commit, and continue when you’re finished.

## Reordering Commits

You can also use interactive rebases to reorder or remove commits entirely. If you want to remove the "added cat-file" commit and change the order in which the other two commits are introduced, you can change the rebase script from this

pick f7f3f6d changed my name a bit

pick 310154e updated README formatting and added blame

pick a5f4a0d added cat-file

to this:

pick 310154e updated README formatting and added blame

pick f7f3f6d changed my name a bit

When you save and exit the editor, Git rewinds your branch to the parent of these commits, applies 310154e and then f7f3f6d, and then stops. You effectively change the order of those commits and remove the "added cat-file" commit completely.

## Squashing Commits

It’s also possible to take a series of commits and squash them down into a single commit with the interactive rebasing tool. The script puts helpful instructions in the rebase message:

#

# Commands:

# p, pick = use commit

# e, edit = use commit, but stop for amending

# s, squash = use commit, but meld into previous commit

#

# If you remove a line here THAT COMMIT WILL BE LOST.

# However, if you remove everything, the rebase will be aborted.

#

If, instead of "pick" or "edit", you specify "squash", Git applies both that change and the change directly before it and makes you merge the commit messages together. So, if you want to make a single commit from these three commits, you make the script look like this:

pick f7f3f6d changed my name a bit

squash 310154e updated README formatting and added blame

squash a5f4a0d added cat-file

When you save and exit the editor, Git applies all three changes and then puts you back into the editor to merge the three commit messages:

# This is a combination of 3 commits.

# The first commit's message is:

changed my name a bit

# This is the 2nd commit message:

updated README formatting and added blame

# This is the 3rd commit message:

added cat-file

When you save that, you have a single commit that introduces the changes of all three previous commits.

## Splitting a Commit

Splitting a commit undoes a commit and then partially stages and commits as many times as commits you want to end up with. For example, suppose you want to split the middle commit of your three commits. Instead of "updated README formatting and added blame", you want to split it into two commits: "updated README formatting" for the first, and "added blame" for the second. You can do that in the rebase -i script by changing the instruction on the commit you want to split to "edit":

pick f7f3f6d changed my name a bit

edit 310154e updated README formatting and added blame

pick a5f4a0d added cat-file

When you save and exit the editor, Git rewinds to the parent of the first commit in your list, applies the first commit (f7f3f6d), applies the second (310154e), and drops you to the console. There, you can do a mixed reset of that commit with git reset HEAD^, which effectively undoes that commit and leaves the modified files unstaged. Now you can take the changes that have been reset, and create multiple commits out of them. Simply stage and commit files until you have several commits, and run git rebase --continue when you’re done:

git reset HEAD^

git add README

git commit -m 'updated README formatting'

git add lib/simplegit.rb

git commit -m 'added blame'

git rebase --continue

Git applies the last commit (a5f4a0d) in the script, and your history looks like this:

git log -4 --pretty=format:"%h %s"

1c002dd added cat-file

9b29157 added blame

35cfb2b updated README formatting

f3cc40e changed my name a bit

Once again, this changes the SHAs of all the commits in your list, so make sure no commit shows up in that list that you’ve already pushed to a shared repository.

# Appendix 1 – Glossary

In this appendix we will include a glossary of Git and GitHub related terms.

| Term | Description |
| --- | --- |
| Branch | A "branch" is an active line of development.  A single git [repository](https://www.kernel.org/pub/software/scm/git/docs/gitglossary.html#def_repository) can track an arbitrary number of branches, but your [working tree](https://www.kernel.org/pub/software/scm/git/docs/gitglossary.html#def_working_tree) is associated with just one of them (the "current" or "checked out" branch. |
| Checkout | The action of updating the working tree with and object stored in the Git database. Often this object is another branch. |
| Changeset | Synonym for "commit" |
| Commit | As a noun: A single point in the git history; the entire history of a project is represented as a set of interrelated commits. The word "commit" is often used by git in the same places other revision control systems use the words "revision" or "version". Also used as a short hand for commit object.  As a verb: The action of storing a new snapshot of the project’s state in the git history, by creating a new commit representing the current state of the index and advancing HEAD to point at the new commit. |
| Clone | A clone of a repository ("just another repository") or the act of doing so ("to clone a repository (creates a new clone)") |
| Merge | To bring the contents of another branch (possibly from an external repository) into the current branch. In the case where the merged-in branch is from a different repository, this is done by first fetching the remote branch and then merging the result into the current branch. This combination of fetch and merge operations is called a pull. |
| Head | The current branch. In more detail: Your working tree is normally derived from the state of the tree referred to by HEAD. |
| Origin | The default upstream repository. Most projects have at least one upstream project which they track. By default origin is used for that purpose. New upstream updates will be fetched into remote remote-tracking branches named origin/name-of-upstream-branch, which you can see using git branch -r. |
| Pull | Pulling a branch means to fetch it and merge it. |
| Push | Pushing a branch means putting all objects that have been modified into the remote object database. |
| Repository | This is the object database were your history and configuration is stored. May contain several branches. |
| Tag | A descriptive name given to one of your commits (or trees, or blobs). Can also contain a message (eg. changelog). Tags can be cryptographically signed with GPG. |
| Working Tree | The tree of actual checked out files. |
| Upstream | After cloning a repository you often call that "original" repository "upstream". In git it's aliased to origin |