Table des matières

1 Git - petit guide par Roger Dudler

1.1 créer un nouveau dépôt

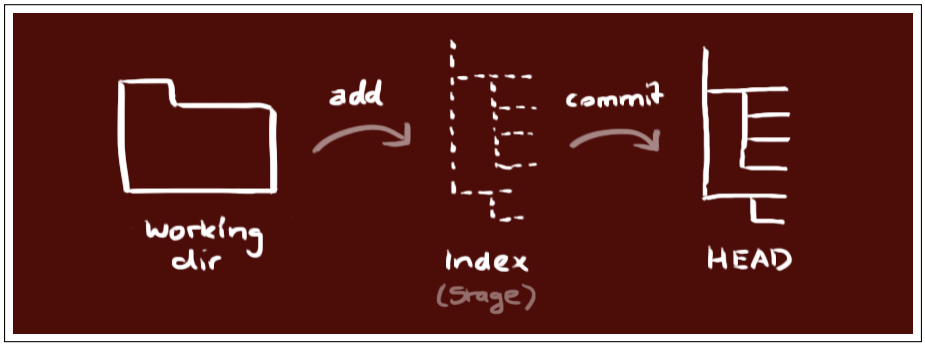
créez un nouveau dossier, ouvrez le et exécutez la commande `git init` pour créer un nouveau dépôt.

1.2 cloner un dépôt

créez une copie de votre dépôt local en exécutant la commande `git clone /path/to/repository`
si vous utilisez un serveur distant, cette commande sera `git clone username@host:/path/to/repository`

1.3 arbres

votre dépôt local est composé de trois "arbres" gérés par git. le premier est votre `espace de travail` qui contient réellement vos fichiers. le second est un `Index` qui joue un rôle d'espace de transit pour vos fichiers et enfin `HEAD` qui pointe vers la dernière validation que vous ayez fait.



1.4 ajouter & valider

Vous pouvez proposer un changement (l'ajouter à l'**Index**) en exécutant les commandes `git add <filename>` `git add *` C'est la première étape dans un workflow git basique. Pour valider ces changements, utilisez `git commit -m "Message de validation"` Le fichier est donc ajouté au **HEAD**, mais pas encore dans votre dépôt distant.

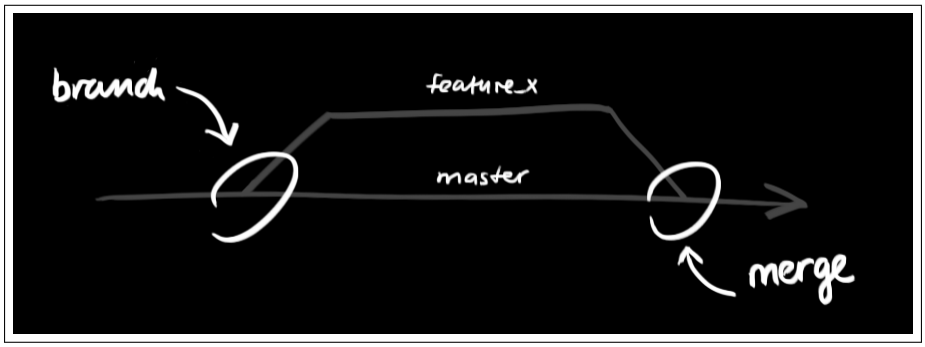
1.5 envoyer des changements

Vos changements sont maintenant dans le **HEAD** de la copie de votre dépôt local. Pour les envoyer à votre dépôt distant, exécutez la commande `git push origin master` Remplacez *master* par la branche dans laquelle vous souhaitez envoyer vos changements.

Si vous n'avez pas cloné votre dépôt existant et voulez le connecter à votre dépôt sur un serveur distant, vous devez l'ajouter avec `git remote add origin <server>` Maintenant, vous pouvez envoyer vos changements vers le serveur distant sélectionné

1.6 branches

Les branches sont utilisées pour développer des fonctionnalités isolées des autres. La branche *master* est la branche par défaut quand vous créez un dépôt. Utilisez les autres branches pour le développement et fusionnez ensuite à la branche principale quand vous avez fini.



créer une nouvelle branche nommée “feature_x” et passer dessus pour l'utiliser `git checkout -b feature_x` retourner sur la branche principale `git checkout master` et supprimer la branche `git branch -d feature_x` une branche n'est *pas disponible pour les autres* tant que vous ne l'aurez pas envoyée vers votre dépôt distant `git push origin <branch>`

1.7 mettre à jour & fusionner

pour mettre à jour votre dépôt local vers les dernières validations, exécutez la commande `git pull` dans votre espace de travail pour *récupérer* et *fusionner* les changements distants. pour fusionner une autre branche avec la branche active (par exemple master), utilisez `git merge <branch>` dans les deux cas, git tente d'auto-fusionner les changements. Malheureusement, ça n'est pas toujours possible et résulte par des *conflits*. Vous devez alors régler ces *conflits* manuellement en éditant les fichiers indiqués par git. Après l'avoir fait, vous devez les marquer comme fusionnés avec `git add <filename>` après avoir fusionné les changements, vous pouvez en avoir un aperçu en utilisant `git diff <source_branch> <target_branch>`

1.8 tags

il est recommandé de créer des tags pour les releases de programmes. c'est un concept connu, qui existe aussi dans SVN. Vous pouvez créer un tag nommé *1.0.0* en exécutant la commande `git tag 1.0.0 1b2e1d63ff` le *1b2e1d63ff* désigne les 10 premiers caractères de l'identifiant du changement que vous voulez référencer avec ce tag. Vous pouvez obtenir cet identifiant avec `git log` vous pouvez utiliser moins de caractères de cet identifiant, il doit juste rester unique.

1.9 remplacer les changements locaux

Dans le cas où vous auriez fait quelque chose de travers (ce qui bien entendu n'arrive jamais ;) vous pouvez annuler les changements locaux en utilisant cette commande `git checkout -- <filename>` cela remplacera les changements dans votre arbre de travail avec le dernier contenu du HEAD. Les changements déjà ajoutés à l'index, aussi bien les nouveaux fichiers, seront gardés.

Si à la place vous voulez supprimer tous les changements et validations locaux, récupérez le dernier historique depuis le serveur et pointez la branche principale locale dessus comme ceci `git fetch origin` `git reset --hard origin/master`

1.10 conseils utiles

Interface git incluse

`‘gitk‘`

utiliser des couleurs dans la sortie de git

`‘git config color.ui true‘`

afficher le journal sur une seule ligne pour chaque validation

`‘git config format.pretty oneline‘`

utiliser l’ajout interactif

`‘git add -i‘`

2 From GitHub

2.1 Got 15 minutes and want to learn Git ?

Git allows groups of people to work on the same documents (often code) at the same time, and without stepping on each other’s toes. It’s a distributed version control system.

To initialize a Git repository here, type the following command :

```
$ git init
Initialized empty Git repository in /.git/
```

Directory : A folder used for storing multiple files.

Repository : A directory where Git has been initialized to start version controlling your files.

2.2 Checking the Status

Good job ! As Git just told us, our octobox directory now has an empty repository in /.git/. The repository is a hidden directory where Git operates.

Next up, let’s type the git status command to see what the current state of our project is :

```
$ git status
# On branch master
#
# Initial commit
#
nothing to commit (create/copy files and use "git add" to track)
```

The .git directory : you’ll notice it has all sorts of directories and files inside it. You’ll rarely ever need to do anything inside here but it’s the guts of Git, where all the magic happens.

2.3 Adding & Committing

Say you created a file called octocat.txt in the octobox repository

You should run the git status command again to see how the repository status has changed :

```
$ git status

# On branch master
#
# Initial commit
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#
# octocat.txt
nothing added to commit but untracked files present (use "git add" to track)
```

Tip : It's healthy to run git status often. Sometimes things change and you don't notice it.

2.4 Adding Changes

Good, it looks like our Git repository is working properly. Notice how Git says octocat.txt is "untracked" ? That means Git sees that octocat.txt is a new file.

To tell Git to start tracking changes made to octocat.txt, we first need to add it to the staging area by using git add.

```
$ git add octocat.txt
```

staged : Files are ready to be committed.
unstaged : Files with changes that have not been prepared to be commited.
untracked : Files aren't tracked by Git yet. This usually indicates a newly created file.
deleted : File has been deleted and is waiting to be removed from Git.

2.5 Checking for Changes

Good job ! Git is now tracking our octocat.txt file. Let's run git status again to see where we stand :

```
$ git status

# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#
```

add all : You can also type git add .. The dot represents the current directory, so everything in it, and everything beneath it gets added.
git reset : You can use git reset to remove a file or files from the staging area.

2.6 Committing

Notice how Git says changes to be committed ? The files listed here are in the Staging Area, and they are not in our repository yet. We could add or remove files from the stage before we store them in the repository.

To store our staged changes we run the commit command with a message describing what we've changed. Let's do that now by typing :

```
$ git commit -m "Add cute octocat story"

[master (root-commit) 20b5ccd] Add cute octocat story
1 file changed, 1 insertion(+)
create mode 100644 octocat.txt
```

Staging Area : A place where we can group files together before we "commit" them to Git.

Commit A "commit" is a snapshot of our repository. This way if we ever need to look back at the changes we've made (or if someone else does), we will see a nice timeline of all changes.

2.7 Adding All Changes

Great ! You also can use wildcards if you want to add many files of the same type. Notice that I've added a bunch of .txt files into your directory below.

I put some in an octofamily directory and some others ended up in the root of our octobox. Luckily, we can add all the new files using a wildcard with git add. Don't forget the quotes !

```
git add '*.txt'
```

Wildcards : We need quotes so that Git will receive the wildcard before our shell can interfere with it. Without quotes our shell will only execute the wildcard search within the current directory. Git will receive the list of files the shell found instead of the wildcard and it will not be able to add the files inside of the octofamily directory.

2.8 Committing All Changes

Okay, you've added all the text files to the staging area. Feel free to run git status to see what you're about to commit.

If it looks good, go ahead and run :

```
git commit -m 'Add all the octocat txt files'

[master 3852b4d] Add all the octocat txt files
 4 files changed, 4 insertions(+)
 create mode 100644 blue_octocat.txt
 create mode 100644 octofamily/baby_octocat.txt
 create mode 100644 octofamily/momma_octocat.txt
 create mode 100644 red_octocat.txt


 Check all the things!
```

~ When using wildcards you want to be extra careful when doing commits. Make sure to check what files and folders are staged by using git status before you do the actual commit. This way you can be sure you're committing only the things you want.

2.9 History

So we've made a few commits. Now let's browse them to see what we changed.

Fortunately for us, there's git log. Think of Git's log as a journal that remembers all the changes we've committed so far, in the order we committed them. Try running it now :

```
$ git log

commit 3852b4db1634463d0bb4d267edb7b3f9cd02ace1
Author: Try Git <try_git@github.com>
Date:   Sat Oct 10 08:30:00 2020 -0500

    Add all the octocat txt files

commit b652edfd888cd3d5e7fcb857d0dabc5a0fcb5e28
Author: Try Git <try_git@github.com>
Date:   Sat Oct 10 08:30:00 2020 -0500

Added cute octocat story
```

More useful logs : Use git log –summary to see more information for each commit. You can see where
new files were added for the first time or where files were deleted. It's a good overview of what's going on in the project.

2.10 Remote Repositories

Great job ! We've gone ahead and created a new empty GitHub repository for you to use with Try Git at https ://github.com/try-git/try_git.git. To push our local repo to the GitHub server we'll need to add a remote repository.

This command takes a remote name and a repository URL, which in your case is https ://github.com/try-git/try_git.git.

Go ahead and run git remote add with the options below :

git remote add origin https ://github.com/try-git/try_git.git

git remote : Git doesn't care what you name your remotes, but it's typical to name your main
one origin.
It's also a good idea for your main repository to be on a remote server like GitHub in case your machine is lost at sea during a transatlantic boat cruise or crushed by three monkey statues during an earthquake.

2.11 Pushing Remotely

The push command tells Git where to put our commits when we're ready, and boy we're ready. So let's push our local changes to our origin repo (on GitHub).

The name of our remote is origin and the default local branch name is master. The -u tells Git to remember the parameters, so that next time we can simply run git push and Git will know what to do. Go ahead and push it!

```
$ git push -u origin master

Branch master set up to track remote branch master from origin.
```

> *Cool Stuff* : When you start to get the hang of git you can do some really cool things with hooks when you push.
> For example, you can upload directly to a webserver whenever you push to your master remote instead of having to upload your site with an ftp client. Check out Customizing Git - Git Hooks for more information.

2.12 Pulling Remotely

Let's pretend some time has passed. We've invited other people to our github project who have pulled your changes, made their own commits, and pushed them.

We can check for changes on our GitHub repository and pull down any new changes by running :

```
$ git pull origin master

Updating 3852b4d..3e70b0f
Fast-forward
 yellow_octocat.txt |    1+
 1 file changed, 1 insertion(+)
 create mode 100644 yellow_octocat.txt
```

> *git stash* : Sometimes when you go to pull you may have changes you don't want to commit just yet. One option you have, other than commiting, is to stash the changes. Use the command 'git stash' to stash your changes, and 'git stash apply' to re-apply your changes after your pull.

2.13 Differences

Uh oh, looks like there has been some additions and changes to the octocat family. Let's take a look at what is different from our last commit by using the git diff command.

In this case we want the diff of our most recent commit, which we can refer to using the HEAD pointer.

```
$ git diff HEAD

diff --git a/octocat.txt b/octocat.txt
index 7d8d808..e725ef6 100644
-------- a/octocat.txt
+++ b/octocat.txt
@@ -1 +1 @@
-A Tale of Two Octocats
 Tale of Two Octocats and an Octodog
```

HEAD The HEAD is a pointer that holds your position within all your different Tale of Two Octocats commits. By default HEAD points to your most recent commit, so it can be used as a quick way to reference that commit without having to look up the SHA.

2.14 Staged Differences

Another great use for diff is looking at changes within files that have already been staged. Remember, staged files are files we have told git that are ready to be committed.

Let's use git add to stage octofamily/octodog.txt, which I just added to the family for you.

```
$ git add octofamily/octodog.txt
```

Commit Etiquette : You want to try to keep related changes together in separate commits. Using 'git diff' gives you a good overview of changes you have made and lets you add files or directories one at a time and commit them separately.

2.15 Staged Differences (cont'd)

Good, now go ahead and run git diff with the –staged option to see the changes you just staged. You should see that octodog.txt was created.

```
$ git diff --staged

diff --git a/octofamily/octodog.txt b/octofamily/octodog.txt
new file mode 100644
index 0000000..cfbc74a
-------- /dev/null
+++ b/octofamily/octodog.txt
@@ -0,0 +1 @@
+[mwoof
```

2.16 Resetting the Stage

So now that octodog is part of the family, octocat is all depressed. Since we love octocat more than octodog, we'll turn his frown around by removing octodog.txt.

You can unstage files by using the git reset command. Go ahead and remove octofamily/octodog.txt.

```
$ git reset octofamily/octodog.txt
```

2.17 Undo

git reset did a great job of unstaging octodog.txt, but you'll notice that he's still there. He's just not staged anymore. It would be great if we could go back to how things were before octodog came around and ruined the party.

Files can be changed back to how they were at the last commit by using the command : git checkout – . Go ahead and get rid of all the changes since the last commit for octocat.txt

```
$ git checkout -- octocat.txt
```

The '–' So you may be wondering, why do I have to use this '–' thing ? git checkout seems to work fine without it. It's simply promising the command line that there are no more options after the '–'. This way if you happen to have a branch named octocat.txt, it will still revert the file, instead of switching to the branch of the same name.

2.18 Branching Out

When developers are working on a feature or bug they'll often create a copy (aka. branch) of their code they can make separate commits to. Then when they're done they can merge this branch back into their main master branch.

We want to remove all these pesky octocats, so let's create a branch called clean_up, where we'll do all the work :

```
$ git branch clean_up
```

Branching : Branches are what naturally happens when you want to work on multiple features at the same time. You wouldn't want to end up with a master branch which has Feature A half done and Feature B half done.

Rather you'd separate the code base into two "snapshots" (branches) and work on and commit to them separately. As soon as one was ready, you might merge this branch back into the master branch and push it to the remote server.

2.19 Switching Branches

Great ! Now if you type git branch you'll see two local branches : a main branch named master and your new branch named clean_up.

You can switch branches using the git checkout command. Try it now to switch to the clean_up branch :

```
$ git checkout clean_up

Switched to branch 'clean_up'
```

All at Once You can use :

`git checkout -b new_branch`

to checkout and create a branch at the same time. This is the same thing as doing :

```
git branch new_branch
git checkout new_branch
```

2.20 Removing All The Things

Ok, so you're in the clean_up branch. You can finally remove all those pesky octocats by using the git rm command which will not only remove the actual files from disk, but will also stage the removal of the files for us.

You're going to want to use a wildcard again to get all the octocats in one sweep, go ahead and run :

```
$ git rm '*.txt'

rm 'blue_octocat.txt'
rm 'octocat.txt'
rm 'octofamily/baby_octocat.txt'
rm 'octofamily/momma_octocat.txt'
rm 'red_octocat.txt'
```

Remove all the things ! Removing one file is great and all, but what if you want to remove an entire folder ? You can use the recursive option on git rm :

`git rm -r folder_of_cats`

This will recursively remove all folders and files from the given directory.

2.21 Commiting Branch Changes

Now that you've removed all the cats you'll need to commit your changes.

Feel free to run git status to check the changes you're about to commit.

```
$ git commit -m "Remove all the cats"

[clean_up 63540fe] Remove all the cats
 5 files changed, 5 deletions(-)
 delete mode 100644 blue_octocat.txt
 delete mode 100644 octocat.txt
 delete mode 100644 octofamily/baby_octocat.txt
 delete mode 100644 octofamily/momma_octocat.txt
 delete mode 100644 red_octocat.txt
```

The '-a' option If you happen to delete a file without using `git rm` you'll find that you still have to `git rm` the deleted files from the working tree. You can save this step by using the `-a` option on 'git commit', which auto removes deleted files with the commit.

`git commit -am "Delete stuff"`

2.22 Switching Back to master

Great, you're almost finished with the cat... er the bug fix, you just need to switch back to the master branch so you can copy (or merge) your changes from the clean_up branch back into the master branch.

Go ahead and checkout the master branch :

```
$ git checkout master

Switched to branch 'master'
```

Pull Requests If you're hosting your repo on GitHub, you can do something called a pull request.

A pull request allows the boss of the project to look through your changes and make comments before deciding to merge in the change. It's a really great feature that is used all the time for remote workers and open-source projects.

Check out out the pull request help page for more information.

2.23 Preparing to Merge

Alrighty, the moment has come when you have to merge your changes from the clean_up branch into the master branch. Take a deep breath, it's not that scary.

We're already on the master branch, so we just need to tell Git to merge the clean_up branch into it :

```
$ git merge clean_up

Updating 3852b4d..ec6888b
Fast-forward
 blue_octocat.txt              |    1 -
 octocat.txt                   |    1 -
 octofamily/baby_octocat.txt   |    1 -
 octofamily/momma_octocat.txt  |    1 -
 red_octocat.txt               |    1 -
 5 files changed, 5 deletions(-)
 delete mode 100644 blue_octocat.txt
 delete mode 100644 octocat.txt
 delete mode 100644 octofamily/baby_octocat.txt
 delete mode 100644 octofamily/momma_octocat.txt
 delete mode 100644 red_octocat.txt
```

> Merge Conflicts
>
> Merge Conflicts can occur when changes are made to a file at the same time. A lot of people get really scared when a conflict happens, but fear not ! They aren't that scary, you just need to decide which code to keep.
>
> Merge conflicts are beyond the scope of this course, but if you're interested in reading more, take a look the section of the Pro Git book on how conflicts are presented.

2.24 Keeping Things Clean

Congratulations ! You just accomplished your first successful bugfix and merge. All that's left to do is clean up after yourself. Since you're done with the clean_up branch you don't need it anymore.

You can use `git branch -d <branch name>` to delete a branch. Go ahead and delete the clean_up branch now :

```
$ git branch -d clean_up

Deleted branch clean_up (was ec6888b).
```

> What if you aae been working on a feature branch and you decide you really don't want this feature anymore ? You might decide to delete the branch since you're scrapping the idea. You'll notice that `git branch -d bad_feature` doesn't
>
> work. This is because -d won't let you delete something that hasn't been merged. You can either add the `--force (-f)` option or use `-D` which combines `-d -f` together into one command.

2.25 The Final Push

Here we are, at the last step. I'm proud that you've made it this far, and it's been great learning Git with you. All that's left for you to do now is to push everything you've been working on to your remote repository, and you're done !

```
$git push

To https://github.com/try-git/try_git.git
   3e70b0f..b07babc  master -> master
```

3 5 things I frequently do and forget with git

Here are few things I used to do a lot in my starting days with git and then forgot to re-search them when needed next time. These tips are from my notes.

3.1 How to undo the last Git commit ?

There are two scenarios in this case, and hence two ways to achieve our goal.

Hard Reset - Completely undo the last commit and all the changes it made.

```
git reset --hard HEAD\~1
```

Imagine the state of the tree is as following :

```
A-B-C
```

where `B` is the current state of files and `C` is the commit we want to undo. The above command completely removes the last commit along with all the changes involved in the commit. The new state of the tree is like this :

```
A-B
```

> **What is** `HEAD?`
> We can think of the HEAD as the "current branch". HEAD is basically a pointer which points to the latest commit. When we switch branches with git checkout, the HEAD revision changes to point to the tip of the new branch.
> **What is** `HEAD\~1` **?**
> It's kind of complicated to explain `HEAD\~1` without explaining `HEAD\^1`. `HEAD\^1` means the first parent of the commit object. `^` means the th parent. `HEAD\~1` means the commit object that is the first generation grand-parent of the named commit object, following only the first parents. e.g `\~3` is equivalent to `\^\^\^` which is equivalent to `\^1\^1\^1`.

Undo the commit but keep the changes

```
git reset HEAD\~1
```

If we want to undo the last commit for some reason (like if we entered the wrong commit message or the commit was incomplete), we can use a soft reset on HEAD. In this case, the tree structure changes to something like :

```
A-B-C
```

What's happening?
In both cases, `HEAD` is just a pointer to the latest commit. When we do a git reset `HEAD\~1`, we tell Git to move the `HEAD` pointer back one commit. But (unless we use –hard) we leave your files as they were. So now git status shows the changes you had checked into C. You haven't lost a thing!

3.2 How to change the commit message of last commit

```
git reset --soft HEAD\~1
```

Using this command we undo the last commit but keep the files as well as the index untouched. This means we can just recommit with no extra effort as a new commit with a new commit message.

What's happening here?
Same as first point. But it leaves the files as well as the index unchanged, so we just need to 'commit' again with a new commit message.

3.3 How to get a commit back after 'reset –hard'

In the first point, we undid the previous commit permanently(?). Sometimes (read often) it happens that we need the destroyed commit back. Following commands can be used to resurrect the destroyed commit.

```
git reflog
```

This command shows a list of partial commit shas. We need to choose the commit we want to restore from this list and use following command.

```
git checkout -b someNewBranchName shaOfDestroyedCommit
```

This will create a new branch and restore the destroyed commit in it, which we can re-merge as required.

What's happening here?
Commits don't actually get destroyed in git for about 90 days. So usually we can go back and restore commits with method explained above.

3.4 How to remove a git submodule?

```
Git submodule?
```

Submodules allow foreign repositories to be embedded within a dedicated subdirectory of the source tree, always pointed at a particular commit. Submodules are meant for different projects you would like to make part of your source tree, while the history of the two projects still stays completely independent and you cannot modify the contents of the submodule from within the main project.

Removing a git submodule Git submodules can simply be removed with 'git rm', but that keeps the submodule entry intact in the .git/config and .gitmodules. There is actually an easier way to remove submodules with a single command.

```
git submodule deinit
```

This is in comprehension to 'git submodule init' command and does all the submodule removal work itself.

3.5 How to delete a remote git branch ?

This one is easy. We just need one command to delete a remote branch.

```
git push origin --delete
```

To remove a local branch, we can of course use

```
git branch -d
```

4 Notes Nico

Autre très bon tuto : http ://www.miximum.fr/tutos/1546-enfin-comprendre-git

REMARQUE : The branches, the tags, the HEAD are just fancy aliases for commits (more on these in some other post (next may be)). A commit is basically the snapshot of present working tree.

4.1 Créer un repo drupal (TODO : réécrire)

Pour le moment nous limitons à un dépôt par vhost. Voici un petit exemple à exécuter localement sur votre machine :

```
$ mkdir {vhost}
$ cd {vhost}
$ git init


$ git clone http://git.drupal.org/project/drupal.git fooproject
$ cd fooproject
$ git checkout 7.0
$ git remote rename origin drupalOrg
$ git remote add dev   /var/wwww/dev
$ git remote add stage /var/www/stage
$ git remote add origin ssh+git://{login}@git.{datacenter_location}.gpaas.net/{vhost}.git
$ mkdir htdocs
$ echo "Hello world" > htdocs/index.html
$ git add htdocs
$ git commit -m "first version of index.html" htdocs
$ git push origin master
```