Introduction to Git

1 Version Control Systems

Whenever you are working on a non-trivial project, keeping track of all the changes to your files and being able to undo some of them, if necessary, is vital. Even more so if you are not working alone, but in a team. You need to be able to share your work with the others, to have them share their work with you, to maintain a history of what has been done, when and by whom. The more information you have on the project's progress, the better you can control it and the more you can learn from it.

Of course, you could share you work manually, via email for example, have your mates send their work to you in the same way and maintain a log yourself, by hand. But the overhead of this process would soon hurt your productivity in more ways than one, it would annoy you more and more and, last but not least, would take up more and more of your time as the frequency of changes increases or as the team grows, and end up becoming overwhelming. Also, lets not forget that manual processes are significantly error prone. And when a process has well-defined, repetitive steps, with little variation in form and purpose, automatizing it is the logical solution to get rid of most of its overhead.

This is how version control systems came into being. A version control system does all these things for us and even more. It keeps track of our changes, it allows us to get back to an earlier, more stable version of our work if it is needed, or to compare two versions in order to understand the differences between then. It helps us share our work with others and enables us to work together in an efficient and easily to monitor way. All this happens in a transparent and user friendly way: mainstream version control systems have a command line interface, for those who want to operate the management of changes in a direct manner, but they also have dedicated plugins for most popular IDEs such as Intellij or Eclipse.

Let us list below some of the best known and most widely used version control systems:

- CVS (Concurrent Versions System) – one of the first version control systems, easy to learn and use, based on a centralized, client-server model
- SVN (Subversion) – very popular, centralized model (versions are kept in a single, shared repository, on a remote server)
- Git – rapidly emerging, based on a distributed model, versions are kept both on a local and a central, shared repository
- Mercurial – also distributed, fast and scalable, with a simpler command set than Git
- Bazaar – decentralized (but can also be deployed in a centralized mode), very versatile, supporting many different workflows
- and many more... (both open source and proprietary, centralized of distributed)

2 Git: concept, architecture and advantages

Git is one of the most well-spread version control systems, nowadays, and its popularity is continuously increasing across both public and private software development enterprises. It is distributed, non-linear and open source, it has been implemented in C and designed by the one and only Linus Torvalds under the purpose of efficiently managing the code base for the development of the Linux kernel.

2.1 Distributed vs centralized vs local

What does it mean that Git is a distributed version control system? Well, lets try to understand this by examining the differences from the other types of version control systems.

The first ever version control systems, such as RCS (Revision Control Systems) or SCCS (Source Code Control System) were local, on the user's own file system. The changes performed on the tracked files were simply

recorded to a local database. These stored differences could be retrieved at any time and used to recreate the state of the tracked files for any previous moment. This was and still is a good way to maintain a local history for your project, however it almost completely excludes collaboration between team members as long as they are not working on the same file system (which is a quite restrictive precondition).

Then, centralized version control systems were developed, which allowed for collaboration between people working on different machines. These version control systems (see CVS, SVN, etc.) rely on a client-server architecture, where the changes are stored on a remote database, on a single, central server, from where they are retrieved by a number of clients. So people actually get to work together, on the same set of files whose consistency is ensured by the version control system, they can all see the whole history of the project. The workflow is quite simple: one has to make sure, before uploading any changes, that one has, on his local workspace, the latest state from the server. If the same file has been modified both locally and remotely, the versioning tool attempts to automatically create a version of the file containing both sets of changes, i.e. merge the concurrent changes. Sometimes this is not possible, for example when both sets of changes hit the same line/s in the file, in which case the merge operation is abandoned, the user is notified of the conflict and has to resolve it manually (or with the assistance of a merge tool). The file/s remain marked as being under conflict until that conflict is explicitly marked as resolved, and only then, when the state of the local workspace is up to date and consistent, can the local changes be committed to the single, central repository.

Centralized version control systems have several downsides, though. Due to the fact that everyone is using the same central repository, working in parallel is limited, or conflicts will frequently arise. If you want to commit your changes, you need the server to be online, so you depend heavily on your connection to the network. Furthermore, the central repository works as a single point of failure: if the server is somehow compromised (hardware failure, DDoS attack, etc.) the whole history of the project, with its past snapshots and changes, is lost and cannot be recovered. So the need for a more robust, high availability architecture is quite obvious.

And so distributed version control systems stepped in. Here, the central repository is not alone, it has a full copy, with its whole history, etc., on each client side. Thus, we have one central repository, on the server, and several local repositories, on each local machine. Each local repository acts as a full backup of the central repository, and can be used to restore it in case of need. Whenever some changes are committed, they are committed to the local repository, and from there they can be pushed to the central repository when necessary, or they can be kept as local, private drafts if the user chooses so. This also implies a significantly reduced dependence on the central repository, since we only need it to be available when we push the local repository's state to it, and not continuously. It also allows the teams to employ more complex workflows than the standard alternation of update/commit available for centralized version control systems. As there's no such thing as a free lunch, the advantages of distributed version control come with a larger storage footprint and a slower initial checkout, since the centralized repository is cloned to the user's local machine. However, due to the usage of specific compression algorithms for both the files on the local and central repositories, this drawback is significantly reduced.

In Git, there are 4 well-defined areas for our changes:

- the working directory: here we have the current state of the repository + changes staged for commit, but not yet commited + ongoing changes (work in progress)
- the staging area: those changes that have been added to the next commit, but have not yet been commited to the local repository
- the local repository
- the central (remote) repository

The files on the local and central repository are compressed for a more efficient storage, while being uncompressed in the working directory and in the staging area, for an easier, more straightforward manipulation. Whenever conflicts arise, they will still be fixed in the working directory, just like for centralized version control systems, with the resolved file versions being then passed to the staging area and, via a commit operation, to the local repository.

2.2 Non-linear version control

We have seen above why is Git a distributed version control system and what do we get out of this. But what does it mean that Git is non-linear?

In order for different features to be developed in parallel, which is almost always the case for software development, one would like to be able to isolate the state of his repository from the rest of the development process, in order to avoid conflicts. This is partially enabled by the local repository, but in a way that is too

restrictive: we might still want to share our work with other people, we might still want to have a version of it safely on the central repository, only a bit aside. What we want, actually, is the possibility of having not only consecutive, but also parallel versions of our files. And this is what branches are, in a version control system: parallel snapshots of the tracked files, located in the same repository. There will always be a main branch of the repository, the master branch, which is supposed to be the most stable. We can create a branch either from the master branch or from other existing branches, develop our feature there, in parallel with other work, and then, when we are done, we can merge our branch into the master branch, or into another branch, as we wish. Being able to do so gives us a simpler, more efficient and more independent development process, as better task parallelism can be employed.

While centralized version control systems like CVS or SVN also offer some support for branch creation and merge, it is done so in a shallow, unassuming way: in SVN a commit has exactly one parent and at most one child, while in Git it may have several parents and children. This means that in SVN a branch exists mostly by convention, as a separate set of commits done in a separate directory, on a copy of the main branch. Meanwhile, GIT offers specific support for branch management, which allows us to create and merge local branches upon will, while only pushing them to the remote repository if necessary.

The development process with Git is thus specifically tailored as non-linear, due to the branching support provided. The commits in a repository with several branches will have a directed acyclic graph structure, describing the actual change history of the repository. This lets us have more complex workflows, that employ the power of branching by keeping the ongoing work on dedicated, feature-specific branches, and only allowing for well-reviewed, high quality code to be merged into the main development branch, sometimes after passing through several phases/filters (code review, acceptance testing, quality analysis, etc.). A common practice in many enterprises is to implement each feature or bugfix on a separate branch and then, when implementation is complete, to open a so called "pull request", basically a proposal consisting of a set of changes which you want to merge to the master branch, but which should be first discussed with, reviewed and approved by your collaborators. Several modifications can be required from the pull request owner before the pull request is approved or rejected, thus increasing both team awareness and accountability with respect to the entire code base and not just one's own features.

3 Some public Git repositories

One does not necessarily have to set up a server for the central repositories himself, since many providers of version control as a service already exist, some of them even offering source code hosting facilities for free.

Thus, several well known Git public repositories exist, which anyone can use to create his own repositories and share his files with his collaborators.

So far, the most popular are:

- GitHub: https://github.com
- BitBucket: https://bitbucket.org/
- SourceForge : https://sourceforge.net/
- ...and many more

4 Git commands and workflow

In this section we will take a more in-depth look to several important git commands and see how we can perform the operations we need from a distributed version control system in a more or less basic workflow. Git has a rich command set, containing both high-level operations and low-level ones (allowing for a full control of the tool's internals).

4.1 The Git reference manual

You can learn more about Git from the manual pages accessible via the `man` interface.

```
$ man git
```

In the same way, you can use `man` to find out more about a certain Git command, like, for example, `commit`.

```
$ man git-commit
```

Or, without the hyphenation:

```
$ man git commit
```

One can also access the command's manual page via `git help`:

```
$ git help commit
```

If you want a list of all available Git commands, it's easy to get:

```
$ git help -a
```

And, finally, if you are interested in diving deeper, you can try to browse the available concept guides from the Git reference manual:

```
$ git help -g
```

Which will lead you to the following output:

```
The common Git guides are:

   attributes   Defining attributes per path
   glossary     A Git glossary
   ignore       Specifies intentionally untracked files to ignore
   modules      Defining submodule properties
   revisions    Specifying revisions and ranges for Git
   tutorial     A tutorial introduction to Git (for version 1.5.1 or newer)
   workflows    An overview of recommended workflows with Git
```

If you want to further explore one of these guides, you can follow up with the following command:

```
$ git help workflows
```

4.2 Basic configuration: git config

In order to be able to track your changes in the logs, across time, you need to provide some basic information about yourself in the Git's configuration. Your name and email would be a minimum minimorum for all projects. It's best to configure them in Git from the very beginning.

```
$ git config --global user.name "Name Surname"
$ git config --global user.email name.surname@domain.com
```

Now, there are many more Git configurations options, either global or with the scope limited to your current repository. You can read more about Git configuration and the associated configuration files by accessing its dedicated manual page:

```
$ git help config
```

4.3 Create a local repository: git init

Lets assume we have a `project` folder somewhere on our disk, either obtained by unzipping an existing project archive or created by ourselves. We want to place this folder and at least some of its content under Git version control.

First we change the current directory to the one we want to add to version control:

```
$ cd project
```

Then, we initialize the Git repository:

```
$ git init
```

If everything goes well, we should get a short, diplomatic response:

```
Initialized empty Git repository in /home/casandra/project/.git/
```

When initializing the local Git repository a new `.git` folder is created in the current folder, and we can take a look at its structure with the UNIX command `ls`:

```
$ ls .git
branches  config  description  HEAD  hooks  info  objects  refs
```

Beside the `config` and `description` files we notice the `HEAD` file, which represents a reference to the last commit on the current branch. We can easily see its content with the UNIX command `cat`. Initially, the HEAD file will reference the tip of the master branch.

```
$ cat .git/HEAD
ref: refs/heads/master
```

Executing the `git init` command in an existing repository is a safe operation, as we won't override existing configurations or other data, etc.

Keep in mind that, after running the initialization, our current `project` folder becomes the working directory associated to the local repository. This doesn't mean, however, that its content is already tracked. If we want to track the changes performed to a file in the working directory, we have to explicitly add that file to the index (the staging area).

4.4 Checking the current status of the repository: git status

In order to see the current state of the working directory, staging area and local repository, you can use the `status` command:

```
$ git status
```

Thus, you will see all the paths that have differences between the working directory and the staging area (the index), between the staging area and the current HEAD commit, and also the untracked files that are not ignored via `git ignore`.

4.5 Add files to index: git add

Lets assume now that we want to add a file, or several, to the staging area, i.e. the index. We can do this with the `add` command.

This is how we can add a previously untracked file to the staging area:

```
$ git add firstFile
```

Or several files:

```
$ git add firstFile secondFile thirdFile
```

We can then use the `status` command to check whether our files have been added to index. For our freshly initialized repository, the output of `git status` will look as below:

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

Similarly, we can add a folder to the staging area:

```
$ git add firstFolder
```

Or the whole content of the working directory:

```
$ git add .
```

It is important to note here the Git tracks content, not files, so if the files added to the staging area will be further modified afterwards, the new changes will not be added to the index by default. On the contrary, they will have to be added explicitly, via a new `git add` command.

4.6 Commiting to the local repository: git commit

The `command` simply records the performed changes to the repository. Only the changes already added to the staging area, i.e. the index are considered when the `commit` operation is performed.

Basically, what happens is that the current content of the staging area is recorded in the local repository as a new commit. In order for the commit to be easily identifiable by anyone using the codebase, the user has to provide a commit message.

```
$ git commit -m "Initial commit"
[master (root-commit) 9229a2e] Initial commit
3 files changed, 1 insertion(+)
 create mode 100644 firstFile
 create mode 100644 secondFile
 create mode 100644 thirdFile
```

Once the changes are commited to the local repository the previous differences between the index and the HEAD are gone, which can be seen with a `git status` operation:

```
$ git status
# On branch master
nothing to commit, working directory clean
```

4.7 Viewing the commit history: git log

At any point in time, you can view the whole history of your current branch by using the `log` command. The output produced will look as below.

```
$ git log
commit 9229a2e6ab54b83149c224648e32917d7651ca70
Author: Casandra Holotescu <casandra@localhost.localdomain>
Date:   Sun Mar 17 19:40:53 2019 +0200

    Initial commit
```

4.8 Viewing the differences: git diff

Lets now assume we have been making some changes (added two lines, with the message "Some random changes / were performed...") to one of the files in the working directory and the current status looks as following:

```
$ git status
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
# modified:   secondFile
#
no changes added to commit (use "git add" and/or "git commit -a")
```

We are no longer sure what exactly we have changed, and would like to be able to see the actual differences between the current state of the working tree and the staging area, before we add anything to the index. This is where the `diff` operation comes in handy, and it will show us which files differ between the two versions, at which lines and what was added and/or removed:

```
$ git diff
diff --git a/secondFile b/secondFile
index e69de29..30e292d 100644
--- a/secondFile
+++ b/secondFile
@@ -0,0 +1,2 @@
+Some random changes
+were performed...
```

So, this is how we use `diff` to compare the working directory with the staging area. This is not, however, the only way in which we can play with `diff`. We can also use it to see the differences between the working directory and HEAD, between two different commits, etc. See `git help diff` for further information on `diff`.

4.9 Exploring the commit history: git show

Sometimes we want to know more about past commits than what can be seen in the log (i.e., the code of the commit, the author, the timestamp and the commit message). We want to see exactly what has been changed by a certain commit, and this is what the `show` command is for. When looking at a certain commit (identified by its unique code) with `show`, we see its main associated information together with all the differences between the previous commit and the examined one, computed using the `diff` command. The full output of `show` for our initial commit looks as below.

```
$ git show 9229a2e6ab54b83149c224648e32917d7651ca70
commit 9229a2e6ab54b83149c224648e32917d7651ca70
Author: Casandra Holotescu <casandra@localhost.localdomain>
Date:   Sun Mar 17 19:40:53 2019 +0200

    Initial commit

diff --git a/firstFile b/firstFile
new file mode 100644
index 0000000..a72c745
--- /dev/null
+++ b/firstFile
@@ -0,0 +1 @@
+ikhcfuihid kjdujdbj  kjjujguyjfh
diff --git a/secondFile b/secondFile
new file mode 100644
index 0000000..e69de29
diff --git a/thirdFile b/thirdFile
new file mode 100644
index 0000000..e69de29
```

4.10 When nostalgia wins: git reset

Sometimes, we really really want to go back to an earlier state of things. It's a bit more complicated in life, but, fortunately, version control systems have a command for this: `reset`. There are several ways in which we can use `reset` to turn back not time, but the flow of changes in our repository.

First, assume we have added a certain file to the index, but we've changed our mind about it.

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
# modified:   secondFile
#
```

We can easily remove the file from the staging area with `reset`.

```
$ git reset HEAD secondFile
Unstaged changes after reset:
M secondFile
```

Also, if we have actually gone further and commited those changes, and now we have second thoughts about it, we can use `reset` to return to the previous commit. Take the following example:

```
$ git log
commit 6c256127832e45bb559b8f3fdd04d7ef31d8b5e4
Author: Casandra Holotescu <casandra@localhost.localdomain>
Date:   Sun Mar 17 21:01:55 2019 +0200

    this commit shall be reverted

commit 9229a2e6ab54b83149c224648e32917d7651ca70
Author: Casandra Holotescu <casandra@localhost.localdomain>
Date:   Sun Mar 17 19:40:53 2019 +0200
```

```
    Initial commit
```

We use the code of the previous commit to pass as parameter to `reset`. The current HEAD will then be set to this commit. There are several modes for reset (`soft`, `hard`, `mixed`, etc.), the difference between them residing in what else takes place besides the HEAD being set to the previous commit (Is the staging area reset too? Are the changes also discarded form the working directory?). The default mode, seen in the example below, is `mixed`, where the staging area is also reset, but the differences between the two commits are not discarded, but kept in the working directory as unstaged changes.

```
$ git reset 9229a2e6ab54b83149c224648e32917d7651ca70
Unstaged changes after reset:
M secondFile
```

If, however, we want to undo other commit than the latest on our branch, it is recommended that we use `git revert`. While `git reset` simply throws away existing commits (especially when used with the `--hard` option), `git revert` adds new commits, that reverse the effects of the undesirable ones. In order to use `git revert`, however, the working directory has to be clean, with no changes from the HEAD.

4.11 Linking to a remote repository: git remote

So far we have explored the tips and tricks of local version control, but, as we have stated before, this doesn't bring us enough value. We need to share our code with others, and for this our local repository is no longer enough, we need the central, remote repository to publish our commits to. And once we have it, of course, we want to link it to our local repository and publish the changes from our local repository to the central one.

Let's assume that we have created a remote repository on some public git repository provider, such as GitHub or BitBucket, on which we are the happy owners of a free account. Let's also assume that our remote repository has the following URL: `https://github.com/CasandraHolotescu/remoteRepository.git`.

We will use the `remote` command, designed to manage the set of remote tracked repositories, more specifically we will use `git remote add` to link our remote repository to the local one, under the name "origin".

```
$ git remote add origin https://github.com/CasandraHolotescu/remoteRepository.git
```

Then, `git remote` with no parameters will show us the list of tracked remote repositories, which, in our case, contains only "origin".

```
$ git remote
origin
```

4.11.1 Interacting with the remote repository over SSH

If you would experience trouble while interacting with GitHub over HTTPS, you can change the protocol by editing the `config` file from the `.git` folder. For example, you can choose to perform remote read (like `fetch`, etc.) operations via the `git` protocol, while performing remote write operations via `ssh`. In this case, the lines from your `config` file referring to the `origin` remote repository will look as following:

```
[remote "origin"]
url = git://github.com/CasandraHolotescu/remoteRepository.git
pushUrl = git@github.com:CasandraHolotescu/remoteRepository.git
```

You can check the details of the associated remote repository with `git remote show`:

```
git remote show origin
```

However, in order to be able to interact with GitHub over `ssh`, you need to add an `ssh` public key to your GitHub account. First, you need to generate this key using the `ssh-keygen` command:

```
$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/casandra/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/casandra/.ssh/id_rsa.
Your public key has been saved in /home/casandra/.ssh/id_rsa.pub.
```

We can see the public `ssh` key by using the UNIX command `cat` (key has been shortened for obvious reasons):

```
$ cat /home/casandra/.ssh/id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAAA...8CPYflGqBmyFoXd casandra@localhost.localdomain
```

Then, you copy this key and go to you GitHub account, to the `Settings` section of your profile (`https://github.com/settings/profile`). Here, you go to the `SSH and GPD keys` section, and you add a new SSH key: in the `Title` field you introduce a chosen name for your public key, and in the `Key` field you paste the public key you have generated with the `ssh-keygen` command (the whole key, that is, in our case "ssh-rsa AAAAB3NzaC1yc2EAAAA...8CPYflGqBmyFoXd casandra@localhost.localdomain").

And now you should be able to publish you commits to the remote repository via `ssh`.

4.12 Publishing our local commits: git push

We will now push our master branch to the remote repository:

```
$ git push origin master

Warning: Permanently added the RSA host key for IP address '140.82.118.4' to the list of known hosts.
Counting objects: 8, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (7/7), 672 bytes | 0 bytes/s, done.
Total 7 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), done.
To git@github.com:CasandraHolotescu/remoteRepository.git
   a014ff4..81f2cee  master -> master
```

In the same way, we can publish any local commits, from any local branch, to the remote repository, by using `git push remote-name branch-name`.

Furthermore, if we want to set our local branch to track, from this point on, the newly pushed, remote branch, then we can use:

```
$ git push -u origin newBranch
Total 0 (delta 0), reused 0 (delta 0)
remote:
remote: Create a pull request for 'newBranch' on GitHub by visiting:
remote:      https://github.com/CasandraHolotescu/remoteRepository/pull/new/newBranch
remote:
To git@github.com:CasandraHolotescu/remoteRepository.git
 * [new branch]      newBranch -> newBranch
Branch newBranch set up to track remote branch newBranch from origin.
```

We need, however, to pay attention to one important aspect: before being allowed to push our changes to the remote repository we need to make sure that our current branch from our local repository is already up to date, otherwise the `push` operation will be rejected by the remote repository and we will be asked to update first (see example below).

```
$ git push origin master
To git@github.com:CasandraHolotescu/remoteRepository.git
 ! [rejected]        master -> master (fetch first)
error: failed to push some refs to 'git@github.com:CasandraHolotescu/remoteRepository.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first merge the remote changes (e.g.,
hint: 'git pull') before pushing again.
```

4.13 Retrieving remote objects: git fetch

The `git fetch` command is used to download objects and references from the remote repository (or repositories). The retrieved data, however, is not incorporated into neither the local repository, nor the staging area or the working directory. It is, instead, stored in `.git/FETCH_HEAD` and it can be incorporated into the state of the current repository later on, via a `git merge` or a `git rebase` command.

A `fetch` operation to retrieve all objects and references from our remote repository can be performed as below.

```
$ git fetch origin
```

If, however, we wish to download all objects and references from all remote repositories we are linked to, then we will perform

```
$ git fetch --all
```

4.14 Incorporating remote changes: git pull

The `pull` command is used when we want to both retrieve and incorporate the changes from a remote repository into our current branch. For our remote repository and our master branch, we can perform the `pull` operation as following:

```
$ git pull origin master
```

If, however, we want to be able to use the simpler `git pull` command, and if we know that we will always pull the changes from the same remote branch to our local branch, then we can set our local branch `master` to track the remote branch `origin/master`. This will cause all the subsequent `git push`, `git pull` and `git rebase` actions to happened between the tracking local branch and the tracked remote one.

```
$ git branch --set-upstream-to=origin/master
Branch master set up to track remote branch master from origin.

$ git pull
Already up-to-date.
```

It is worth noting here that the default `git pull` operation actually consists from a `git fetch`, which runs for the specified branch and remote repository, followed by a `git merge`, which merges the HEAD of the retrieved branch into the current branch. There is also the possibility of performing `pull` with the option `--rebase`.

```
$ git pull --rebase
```

In this case, the retrieved changes will be rebased, instead of merged, into the current branch.

4.15 Branch management: git branch

We have seen before that non-linearity and more specifically branching is one of the strong points of Git. So, how can we manage branches in Git? Unexpectedly, with the `git branch` command.

We can create a new local branch from the HEAD commit of the current branch

```
$ git branch newBranch
```

And we can set an upstream branch for it at creation, that means we can set it to track a specific branch from the remote repository.

```
$ git branch newBranch --set-upstream-to=origin/master
Branch newBranch set up to track remote branch master from origin.
```

We can use `git branch`, with no parameters, to see all local branches (current branch is marked with "*").

```
$ git branch
* master
  newBranch
```

And with parameter "-r" to see all remote branches:

```
$ git branch -r
  origin/master
  origin/newBranch
```

And, finally, we can use it with parameter "-d" to delete a branch, be it local or remote ("-d -r").

```
$ git branch -d newBranch
Deleted branch newBranch (was 81f2cee).
$ git branch -d -r origin/newBranch
Deleted remote branch origin/newBranch (was 81f2cee).
```

4.16 Switch between branches: git checkout

After creating a new branch in our local repository, we can use the `checkout` command to switch to it and make it our new current branch (we notice that the subsequent `git branch` shows `newBranch`).

```
$ git checkout newBranch
Switched to branch 'newBranch'
$ git branch
  master
* newBranch
```

We also have the possibility to create a new branch and then switch to it, all in one command, by using `checkout` with parameter "-b".

```
$ git checkout -b newestBranch
Switched to a new branch 'newestBranch'
$ git branch
  master
  newBranch
* newestBranch
```

4.17 Incorporating changes: git merge

What the `git merge` command does is to actually join two or more development histories together. Simply put, it merges the changes from one branch into another, the current branch.

In the example below, we merge branch `newBranch` into branch `newestBranch` (the current branch).

```
$ git merge newBranch
Updating 81f2cee..65c5301
Fast-forward
 4thFile | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 4thFile
```

If, as above, one branch is another's ancestor and there are no new commits on the current branch, then the merge operation can be forwarded, since a new commit is not needed to store the combined history. This means that no new merge commit is created and that simply the HEAD (along with the index) is updated to point at the tip of the merged branch (i.e., its latest commit).

Otherwise, however, the merged branches are tied together via a new commit, known as the merge commit, which has both branches as parents. The HEAD, the index, and the working directory are all updated to the merge commit.

4.17.1 Merge conflicts

Not always, however, is it possible to automatically replay the changes from one branch on top of the other and create a version that will reconcile them all. Sometimes, due to changes that happen to hit the same file or files, at the same line/s, the tool cannot perform the automated merge operation and what is known as a merge conflict arises.

In this situation, the paths that can be merged cleanly are merged, while the others are marked as conflicted and up to 3 versions of the index files are recorded: the common ancestor, the HEAD (of the current branch) and the MERGE_HEAD (the HEAD of the branch to be merged). Meanwhile, the files from the working tree display the conflict markers resulting from the failed merge operation.

There are several ways to address a merge conflict:

- fix it automatically, by giving up the conflicting changes from one branch by merging using the "-ours" or "-theirs" strategy

  ```
  $ git merge -s ours newBranch
  ```

- fix it via a mergetool, by launching a graphical conflict visualization and merge tool, which will guide us through the process

  ```
  $ git mergetool
  ```

- manually, by examining the differences between the three versions from the staging area, performing the needed editing and then marking the conflict as resolved by adding the conflicted paths manually to the merge commit, followed by a `commit` operation which concludes the merge

4.18 Incorporating changes: git rebase

There is another possibility of integrating the changes from one branch to another, besides `merge`, and that is the `rebase` operation.

In the example below we are rebasing `newestBranch` on the `master` branch, in the local repository.

```
$ git rebase newestBranch
First, rewinding head to replay your work on top of it...
Fast-forwarded master to newestBranch.
```

Rebase is different from merge in the fact that it does not simply integrate the differences from one branch with the ones from the other and creates a final, reconciliating commit. Instead, the `rebase` operation creates a linear structure on the destination branch.

Rebasing works by going back to the common ancestor of the two branches and then, from there, it takes all the differences introduced by each commit from that point on, it saves them to temporary files, and then replays them on the top of the destination branch. The end result is the same as for merge, we can have conflicts here too (rebase conflicts) if the changes cannot be automatically reconciled. The difference is that we are missing the final merge commit and that, when viewing the logs, the development history appears linear even if the development took place in parallel, on different branches. Thus, we end up with a cleaner, easier to follow history and this is why `rebase` is preferred over `merge` in many situations.

There is, however, a potential problem arising from `rebase`: one should never rebase published commits, especially commits on which someone else might have based his work upon. This is because when rebasing one actually abandons existing commits and creates new ones, with the same content, of course, but with different identifiers. If those commits have already been published and someone has already done some work on top of them, they will have to merge again, and things can get very messy.

4.19 Cloning a repository: git clone

There are many situations in which we get to start our work from an already existing repository, be it local or remote. In this case, we will use the `clone` command to clone the existing repository into an empty directory of our choice. This creates remote-tracking branches for each branch in the cloned repository.

Below we have cloned out remote repository into a new folder.

```
$ git clone git://github.com/CasandraHolotescu/remoteRepository.git
Cloning into 'remoteRepository'...
remote: Enumerating objects: 11, done.
remote: Counting objects: 100% (11/11), done.
remote: Compressing objects: 100% (6/6), done.
remote: Total 11 (delta 2), reused 7 (delta 1), pack-reused 0
Receiving objects: 100% (11/11), done.
Resolving deltas: 100% (2/2), done.
```

After `git clone`, we can use `git fetch` (with no arguments) to update all the remote-tracking branches, and a subsequent `git pull` (again with no arguments) to merge the remote master branch into the current master branch.

5 IDE Git plugins

Most widely adopted IDEs, such as Eclipse, Intellij or NetBeans have dedicated plugins developed for all major version control systems. Git (and not only) plugins for these IDEs offer a visual interface for interacting with the version control tool, that help us perform the desired version control operations in a more intuitive, user friendly way than via the command line interface. Thus, one trades the low level control and finer options one has when working directly with the command line tool, for a simpler, more intuitive interface that allows us to perform the high level actions without requiring an in-depth knowledge of the version control tool itself.

6 Git practice exercise

In order to further deepen you understanding of Git, you are required to do the following, using the Git commands and concepts that have been detailed before:

1. Go to this repository, which is publicly available (but read-only): `https://github.com/fis-2019/lab1`, and download the code as `.zip`

2. Unpack the `.zip` on your computer, on a local folder
3. From the command line, create a new local repository in that folder
4. Make a first commit in your local repository (don't forget about the commit message)
5. Create an account on GitHub or BitBucket (if you don't have one already), and use that account to create a remote repository
6. Link your remote repository with your local repository and push your local master branch to the remote repository
7. Create a new branch on your local repository, and switch to it
8. Here, start making some relevant changes: create a new class (in a new file) and modify an existing one. Add these changes to the staging area (the index) and then make a commit on your branch, with a easily to comprehend and descriptive message.
9. Push the new branch on the remote repository, making sure that the local branch will be tracking the upstream branch after the push
10. Now, open your project in Intellij and start using the Git client provided by Intellij.
11. Create a new branch from the master branch and start working on it. Here, create a new file, with some content, and also modify the same file you have modified on the first branch, at exactly the same line.
12. Go back to the command line and merge the first branch into master. Then, delete the branch, since it's no longer needed.
13. Using the Git client provided by Intellij, checkout the master branch and then merge the second branch into the master branch.
14. We should have conflicts, since we have changed the same file, at the same line/s. We will use the graphical tool provided by Intellij for conflict resolution.