Source code management

What is source code management?

It allows a developer to organize code iterations chronologically, and version it for an application. The most powerful features of source code management systems are in how they allow teams of very diverse sizes to work together on the same application simultaneously.

Some terms that are common to all of them:

- A project will be called **repository**, it's representing the index/the filesystem root of your project.
- Developers might create **branches** of the codebase, that they will iterate on separately to other developers. For instance, a branch can be meant to be used for a given feature, or a given bug fix. One can create however many branches they need.
- Once a branch is ready for it (it's been tested, peer-reviewed, etc.), it can be **merged** back to the main branch. The main branch may be called differently: in Git it's called **master** , in SVN it's called **trunk**.
- While coding on their branch, developers are meant to work in small, atomic iterations, called **commits**. All commits have a commit message describing in one sentence what's in there.
- All commits together are called the **history** , and it's a big deal to write meaningful commits and commit messages in order to keep the project's history clean at a glance, to understand what has been going on and who did what.
- Some people might be modifying the same pieces on the codebase on different branches, and this could create **conflicts** when one merges those branches together. Some of those conflicts can obviously only be fixed by a human, and each system has a different way to manage merge conflicts.

Which systems exist?

I'll put each specific wording between quotes. The same words may be used for differing notions across the various products.

SourceSafe was an early source code management system from Microsoft, which didn't handle branches, merges, or conflicts. You could "check out" a file, which meant no one else was allowed to "check it out" and modify it at the same time. When you were done with it, you could "check in" the file, making it editable again to the others. Not very suitable for large teams that may work on the same file, and longer iterations in a given file. SourceSafe is discontinued today.

CVS was among the first open-source source code management systems in the industry, and used to be wildly popular, but is barely seen anymore. It didn't handle branches, but two people could modify the same file at the same time. People would "update" their whole directory to get everybody else's work before starting, and "commit" their code to the server when they're done. When they would "commit" a file that had been "committed" by someone else since last time they "updated", the system was not able to merge, so it would consider it a conflict every time, that you would have to manually fix (even if the changes were not on the same lines, for instance).

SVN was built upon CVS to handle branches, so a lot of terminology is the same. At some point, it was the most used system, and it is still seen in the industry, even though people are walking away from it. When you'd create a branch, it would actually copy-paste the whole codebase into another directory in the code repository; then, you'd try to merge, and it would massively compare each file one by one. Merging algorithms were smarter than the CVS ones, but you still had conflicts on most merges, even those that shouldn't necessarily require a

human. When you'd create a "tag" (which is a set version of your code, like "1.2.0"), then the whole codebase would simply be massively copy-pasted into another directory too, but without the intention to merge it back later.

Git doesn't copy-paste the whole codebase when branching and tagging, but stores each commit as a code iteration, in an organized structure that resembles a tree. This allows it to have a much smarter merging algorithm, and it almost never bothers you with conflicts, except for those that really need a human decision. As a result, the cost of branching/merging is very low, and people typically branch/merge a lot, therefore one should never directly work on the "master" branch, if they're not the only developer on the project. Also: unlike its predecessors, Git allows to work and commit without needing to talk with a server, which allows to work on planes, for instance; and it also can work as a decentralized (peer-to-peer) system, although it's very rarely done that way.

Mercurial is very similar to Git in its concepts (although the syntax of its command-line tool is often different). It is more rarely seen in the industry than Git, but is still very relevant. It is the one used by Facebook, for instance, for its main application.

The lowdown on Git

A particularity about Git, is that it's designed to be useable without a central repository (you can **pull** code from your friend's computer, and **push** you work back there, for instance), but not many people use it that way. There is usually a central Git server that the whole team **pushes** code to and **pulls** code from; however, that explains why it is often referred as a "decentralized" system.

So that you can work without a server, the **commit** operation is local, no one other than your computer knows you committed something. You can make several commits however you want, but when you want the server to know about it, you must **push** them there.
You want to be **pulling** from the repository often if other people may be working on the same branch as you, because each **pull** performs a merge operation between the code you didn't have, and the code you recently committed locally. Therefore, in order to let you **push** , Git will sometimes demand that you **pull** first, so that the **merge** can be done on your computer, and you take care of potential **conflicts**.

Sometimes, you may have modified 3 files, but there are only two that you wish to include in the **commit** you're about the make. Therefore, Git has a notion of " **index**", in which you **add** your modified files so that they're included in the next **commit** you register.

How to configure the remote servers your local repository is talking to? They're called **remotes** , and you can configure however many you need. The main one is typically named **origin** (that's the name that is setup by default when you **clone** a project from a remote location in the first place). You can also configure however many **branches** you need, and name them as you please, and the default one is usually called **master**.

Some UI tools for Git exist, but Git is able to do so many things, that they don't represent the magnitude of cases for which you need Git. You really want to learn to use it with the command line. Here are some commands:

```
$ git clone url_of_your_remote_repository   # Clone a repository from a remote reposito
ry
$ git add file1 file2    # will add those two files to the index if they were modified
$ git commit -m "Meaningful commit message"   # will commit those two files (locally)
$ git add .   # will add all of the modified files to the index at once
$ git commit -m "Other meaningful commit message"   # will commit all of those files to
gether
$ git push origin master   # send all commit to the remote server
```

Now, let'd do this again, but by on a branch:

```
$ git branch my_feature   # Creating the branch
$ git checkout my_feature   # Changing the codebase so that we're on that branch now
$ git checkout -b my_feature   # This does the two previous operations in one ;)
$ git add file1 file2
$ git commit -m "Meaningful commit message"   # We didn't just commit this on the maste
r branch like last time, but on the my_feature one
$ git add .
$ git commit -m "Other meaningful commit message"
$ git push origin my_feature   # Notice: we're not pushing master anymore, you just cre
ate a new remote branch
```

Next time you want to work on that branch, you should probably do this first:

```
$ git checkout my_feature   # Just making sure you're currently on the right branch!
$ git pull origin my_feature   # Pulling what your coworkers have done so far.
```

And when you're done with the whole feature and want to merge it to master:

```
$ git checkout master
$ git merge my_feature
```

One other pretty neat thing: if you have a non-Git directory in your computer, and you want to turn it into a Git repository, it's that easy:

```
$ git init   # You're done!
$ git remote add origin url_of_your_git_server   # So that you can push your code somew
here.
```

When you're in a Git repository, and want to know which files are modified but not in the index, and those that are modified and are in the index, you can run:

```
$ git status
```

Git provides many more abilities, such as rewriting pieces of the **history** of the project if you feel the **commits** were not meaningful enough, displaying the **history** in visually meaningful ways, …

For instance, you should run this right now, and see how a complex history can be viewed really nicely:

```
$ git clone [https://github.com/loverajoel/jstips.git](https://github.com/loverajoel/js
tips.git)   # You will need a GitHub account for this to work
$ cd jstips   # changing your directory into the one you just downloaded
$ git log --graph --pretty=tformat:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%an
%cr)%Creset' --abbrev-commit --date=relative
```

Cool, right?

What is the difference between Git and GitHub?

Git is everything we've covered so far: a source code management tool, that comes with a command-line tool for its users.

GitHub is one of many services that provide at the same time:

- a Git repository server to push your code to
- a web UI to that view your repositories, with their files and commits
- a number of extra features (managing your team and accesses, …) Two GitHub features you have to get familiarized with are:

- **Forks** : you can **fork** any repository on GitHub, and it will duplicate the repository's codebase into repository that you own. For instance, if you **fork** twbs/bootstrap, and your GitHub username is "my_username", then it will create the my_username/bootstrap repository, and it will remember where it was forked from. Usually, you aren't allowed to **push** on other people's repositories, so that will give you a repository that you can push to, since you own it.

- **Pull requests:** once you've pushed your code to your repository (or sometimes to a branch of the main repository, if you're allowed), then you can create a **pull request** towards the main repository's **master branch**. Somebody in charge of the main repository will review your pull request (potentially asking you to change a couple of things), and **merge** it if it's suitable to be in the main product.

GitHub has many competitors, two of the main ones being GitLab and BitBucket (which provide very similar services). We chose to make you use GitHub because that's where most of the industry is (that way, you'll be able to interact with them on their open-source projects), and it's also where tech recruiters typically go check out to see what you've been up to.

Some interesting links about Git

- https://try.github.io (/rltoken/Yo5sfG-qfPTQ5wkJ3mh4TA): an interactive tutorial for beginners.
- https://help.github.com/articles/good-resources-for-learning-git-and-github/ (/rltoken/qaMLmt-Ly9w4b63v68l94g): a list of resources about Git, curated by GitHub.
- http://nvie.com/posts/a-successful-git-branching-model/ (/rltoken/Xv4OUmdgaAqZc6xNIbj5Qw): once you master the technical tool, you have many ways to organize your branches according to your project. This very notorious article from 2010 introduces **git-flow** , a detailed proposal for organizing collective work with Git that is still the most common today. You should talk about that each time you start a collaborative project using Git.
- http://semver.org (/rltoken/oxqz34z-08QZECcmpR9JdQ): now that you can give version numbers to your code iterations, how should you number them? Semantic versioning is the most used versioning scheme.
- Git from the inside out (/rltoken/AGmLa9Evzc9Ml8dP9SFOcQ)
- Learn git branching (/rltoken/OAvUt-4nF_R3xb_cC20fqw)

All of your school projects must make proper use of Git, and must be pushed to your GitHub account.