1. Introduction & Git

Organization of PV247

What to expect
In the middle of the course, you will have enough knowledge to create your own applications. Main target of the course is to understand the modern principles of creating FE applications in the React environment and the output is your own project.
Who knows, maybe the best student will get an invitation to work in InQool as well.

This is the schedule of the course, with main topics:

1. Introduction & Git
2. Modern Javascript & ES6
   - What is Node.js
   - Which IDE is the best to use, how and best extensions
   - Most common differences between new and old JS on examples
3. NPM
   - What is Single Page Application
   - What is NPM and NPX
   - How to create React App and all you need to know after you create your first one
   - How to run React App
   - How to build React App and what is actually running in production mode
   - ES6 Modules
   - What is JSX
   - How to use styles, images and UI frameworks
4. Typescript
   - Introduction into typed JavaScript
5. React - Basics
   - What is Function and Class component
   - Props and State
   - We will create basic structure of Tic-Tac-Toe game application
6. React - Advanced
   - Why do we need and how to use terms like these: Fragments, Conditional Rendering, React Router, Lists and keys and Form components
7. React - Hooks
   - main usage of most common hooks and light touch of component lifecycle
8. Asynchronous calls
   - How to work with Firebase
   - async / await syntax
9. React - Context
   - How and when
10. Deployment
    - Continuous integration
11. React Native, Electron
    - React beyond web

Who is a FrontEnd developer?

Front-end web development, also known as client-side development is the practice of producing HTML, CSS and JavaScript for a website or Web Application so that a user can see and interact with them directly. The challenge associated with front end development is that the tools and techniques used to create the front end of a website change constantly, so the developer needs to be constantly aware of how the field is developing.

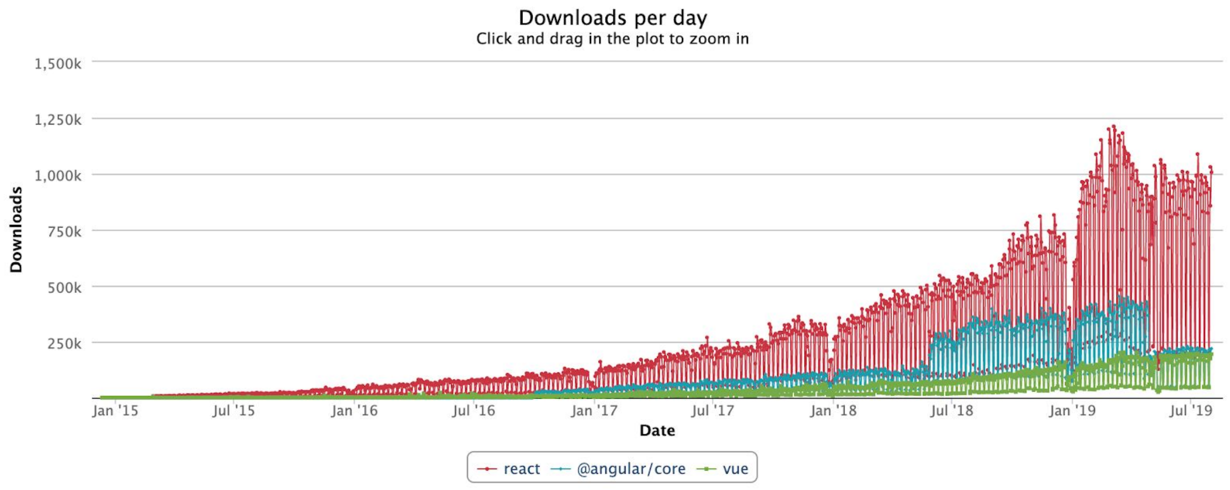
The objective of designing a website is to ensure that users see the information in a format that is easy to read and relevant when they open up the website. This is further complicated by the fact that users now use a large variety of devices with varying screen sizes and resolutions thus forcing the designer to take into consideration these aspects when designing the site. They need to ensure that their site comes up correctly in different browsers (cross-browser), different operating systems (cross-platform) and different devices (cross-device), which requires careful planning on the side of the developer.

https://en.wikipedia.org/wiki/Front-end_web_development

As the field of Front End technologies is very massive and many developers want to work with all technologies at the same time (which is not possible), we are not going to pay attention to every piece of technology. The course is created **to show you one way of work,** which we think is the best one after 5 years of practice.

FE frameworks

There are three the most popular FE frameworks in the middle of 2019. 5 years ago we decided in InQool to follow the path of React, and still, it's the most popular FE library. You can see the statistics of downloads on the following graph.



We hope we gave you enough proofs to convince you to learn React. The other argument why we are learning React is, that the author of this PDF have never worked in Angular or Vue :)

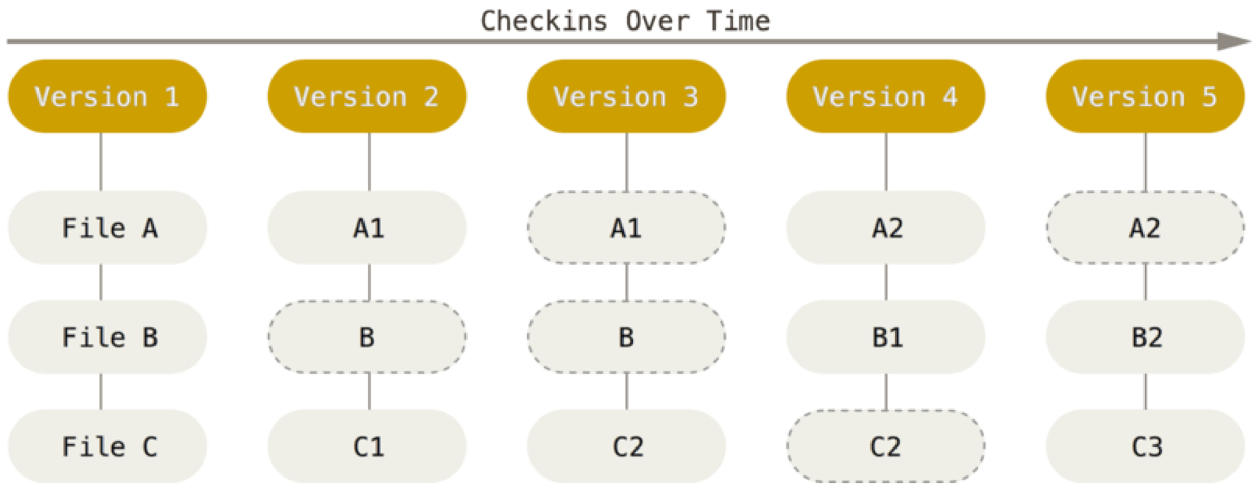
Before we start to write the first lines of source code, we need to know where to store them. And here comes source control technology, a.k.a. Git.

How to use Git

What is Git?

So, what is Git in a nutshell? This is an important section to absorb, because if you understand what Git is and the fundamentals of how it works, then using Git effectively will probably be much easier for you. As you learn Git, try to clear your mind of the things you may know about other VCSs, such as CVS, Subversion or Perforce — doing so will help you avoid subtle confusion when using the tool. Even though Git's user interface is fairly similar to these other VCSs, Git stores and thinks about information in a very different way, and understanding these differences will help you avoid becoming confused while using it.

Snapshots, Not Differences
The major difference between Git and other VCS (Subversion and friends included) is the way Git thinks about its data. Conceptually, most other systems store information as a list of file-based changes. These other systems (CVS, Subversion, Perforce, Bazaar, and so on) think of the information they store as a set of files and the changes made to each file over time (this is commonly described as delta-based version control).Git doesn't think of or store its data this way. Instead, Git thinks about its data more like a series of snapshots of a miniature filesystem. With Git, every time you commit or save the state of your project, Git basically takes a picture of what all your files look like at that moment and stores a reference to that snapshot. To be efficient, if files have not changed, Git doesn't store the file again, just a link to the previous identical file it has already stored. Git thinks about its data more like a stream of snapshots.



Git Has Integrity

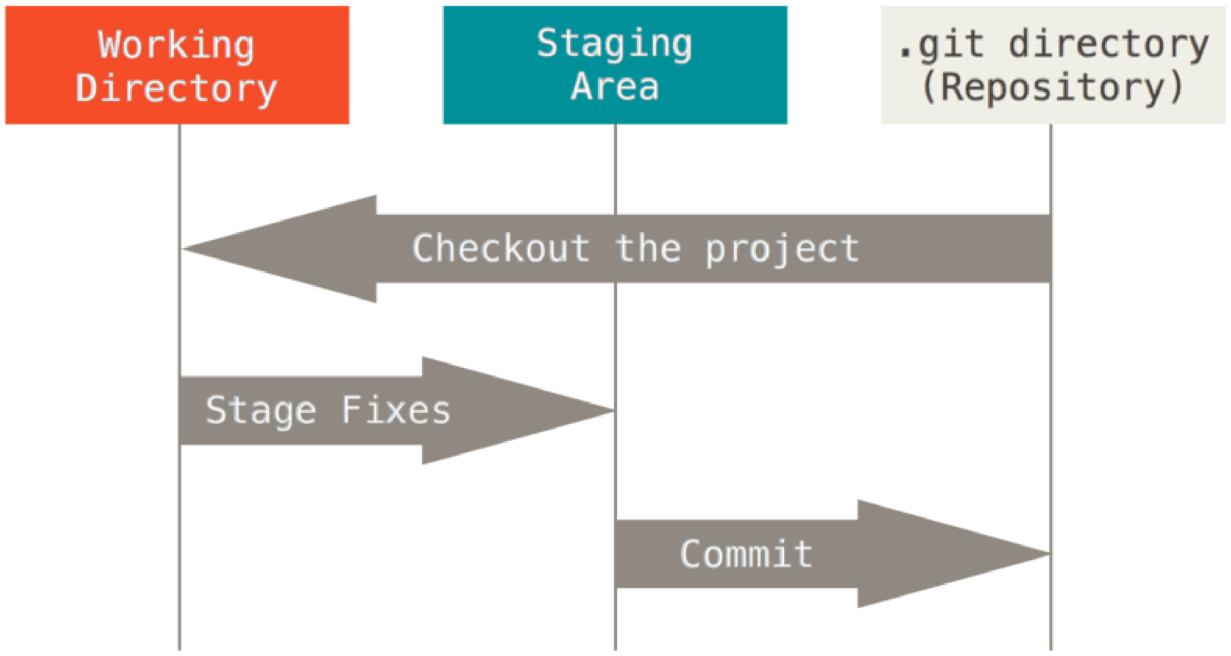
Everything in Git is checksummed before it is stored and is then referred to by that checksum. This means it's impossible to change the contents of any file or directory without Git knowing about it. This functionality is built into Git at the lowest levels and is integral to its philosophy. You can't lose information in transit or get file corruption without Git being able to detect it.
The mechanism that Git uses for this checksumming is called a SHA-1 hash. This is a 40-character string composed of hexadecimal characters (0–9 and a–f) and calculated based on the contents of a file or directory structure in Git. A SHA-1 hash looks something like this: 24b9da6552252987aa493b52f8696cd6d3b00373 You will see these hash values all over the place in Git because it uses them so much. In fact, Git stores everything in its database not by file name but by the hash value of its contents.

The Three States

Pay attention now — here is the main thing to remember about Git if you want the rest of your learning process to go smoothly. Git has three main states that your files can reside in: **modified**, **staged**, and **committed**:

- Modified means that you have changed the file but have not committed it to your database yet.
- Staged means that you have marked a modified file in its current version to go into your next commit snapshot.
- Committed means that the data is safely stored in your local database.

This leads us to the three main sections of a Git project: the working tree, the staging area, and the Git directory.
Working tree, staging area, and Git directory.



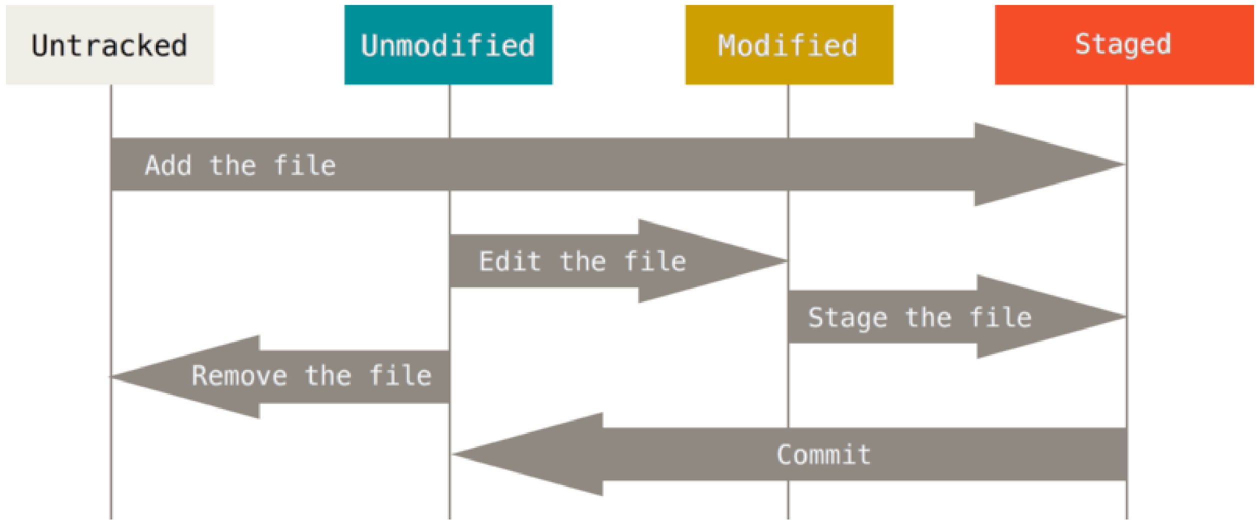
The working tree is a single checkout of one version of the project. These files are pulled out of the compressed database in the Git directory and placed on disk for you to use or modify.
The staging area is a file, generally contained in your Git directory, that stores information about what will go into your next commit. Its technical name in Git parlance is the “index”, but the phrase “staging area” works just as well.
The Git directory is where Git stores the metadata and object database for your project. This is the most important part of Git, and it is what is copied when you clone a repository from another computer.
The basic Git workflow goes something like this:

1. You modify files in your working tree.
2. You selectively stage just those changes you want to be part of your next commit, which adds only those changes to the staging area.
3. You do a commit, which takes the files as they are in the staging area and stores that snapshot permanently to your Git directory.

If a particular version of a file is in the Git directory, it’s considered committed. If it has been modified and was added to the staging area, it is staged. And if it was changed since it was checked out but has not been staged, it is modified. In Git Basics, you’ll learn more about these states and how you can either take advantage of them or skip the staged part entirely.

Recording Changes to the Repository

Remember that each file in your working directory can be in one of two states: **tracked** or **untracked**. Tracked files are files that were in the last snapshot. They can be **unmodified**, **modified**, or **staged**. In short, tracked files are files that Git knows about.
Untracked files are everything else — any files in your working directory that were not in your last snapshot and are not in your staging area. When you first clone a repository, all of your files will be tracked and unmodified because Git just checked them out and you haven't edited anything.
As you edit files, Git sees them as modified, because you've changed them since your last commit. As you work, you selectively stage these modified files and then commit all those staged changes, and the cycle repeats.



Ignoring Files

Often, you'll have a class of files that you don't want Git to automatically add or even show you as being untracked. These are generally automatically generated files such as log files or files produced by your build system. In such cases, you can create a file listing patterns to match them named .gitignore.
You can use this website gitingore.io to generate .gitignore .

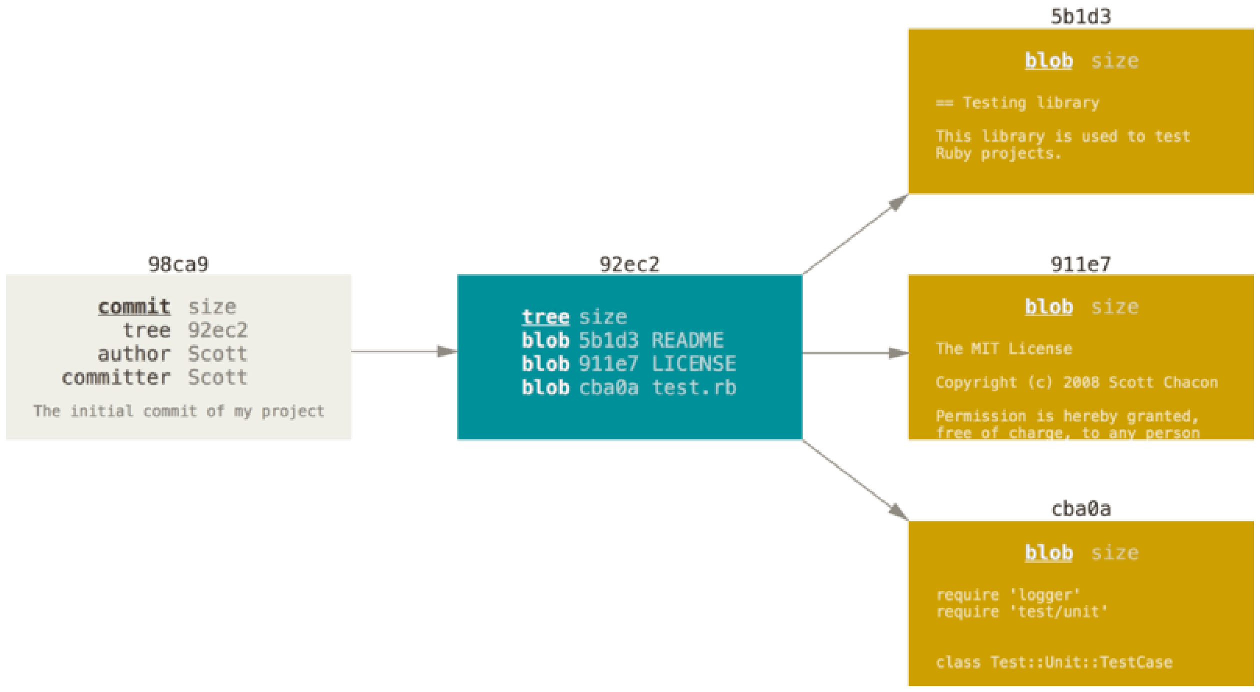
Branches

Git doesn't store data as a series of changesets or differences, but instead as a series of **snapshots**. When you make a commit, Git stores a commit object that contains a pointer to the snapshot of the content you staged. This object also contains the author's name and email address, the message that you typed, and pointers to the commit or commits that directly came before this commit (its parent or parents): zero parents for the initial commit, one parent for a normal commit, and multiple parents for a commit that results from a merge of two or more branches.
To visualize this, let's assume that you have a directory containing three files, and you stage them all and commit. Staging the files computes a checksum for each one (the SHA-1 hash we mentioned),

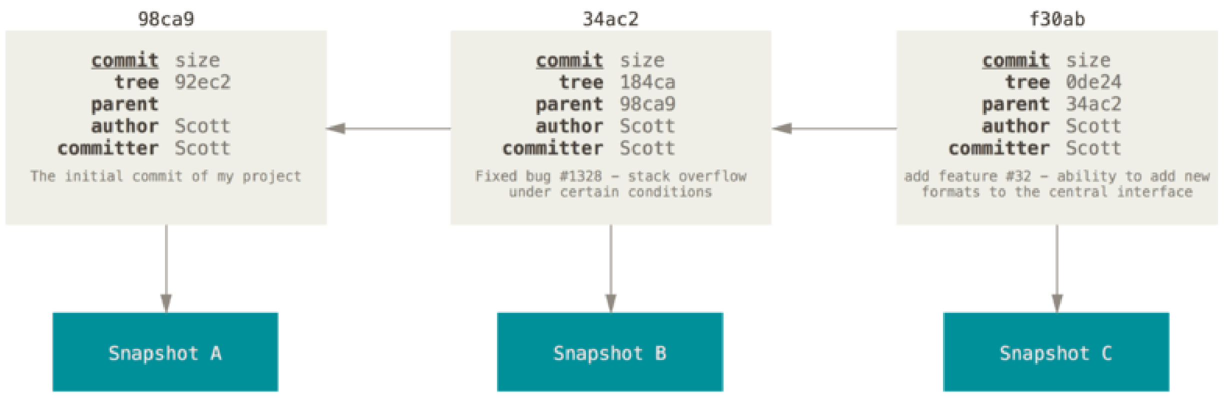
stores that version of the file in the Git repository (Git refers to them as blobs), and adds that checksum to the staging area:

```
$ git add README test.rb LICENSE
$ git commit -m 'The initial commit of my project'
```

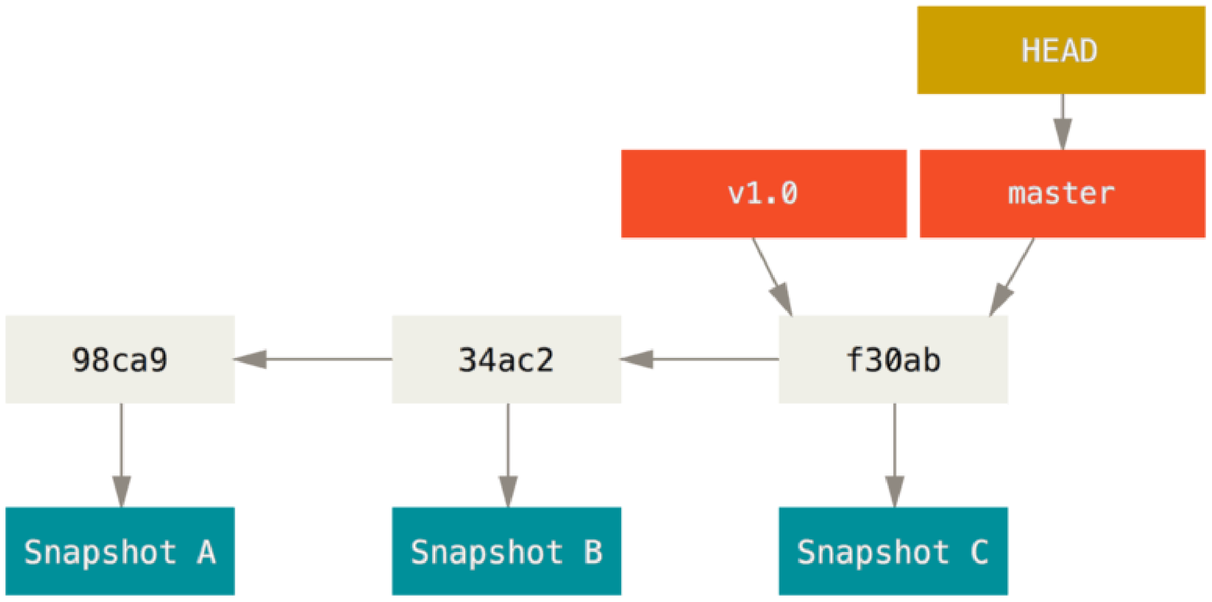
When you create the commit by running git commit, Git checksums each subdirectory (in this case, just the root project directory) and stores them as a tree object in the Git repository. Git then creates a commit object that has the metadata and a pointer to the root project tree so it can re-create that snapshot when needed.
Your Git repository now contains five objects: three blobs (each representing the contents of one of the three files), one tree that lists the contents of the directory and specifies which file names are stored as which blobs, and one commit with the pointer to that root tree and all the commit metadata.



If you make some changes and commit again, the next commit stores a pointer to the commit that came immediately before it.



A **branch** in Git is simply a lightweight movable pointer to one of these commits. The default branch name in Git is master. As you start making commits, you're given a master branch that points to the last commit you made. Every time you commit, the master branch pointer moves forward automatically.

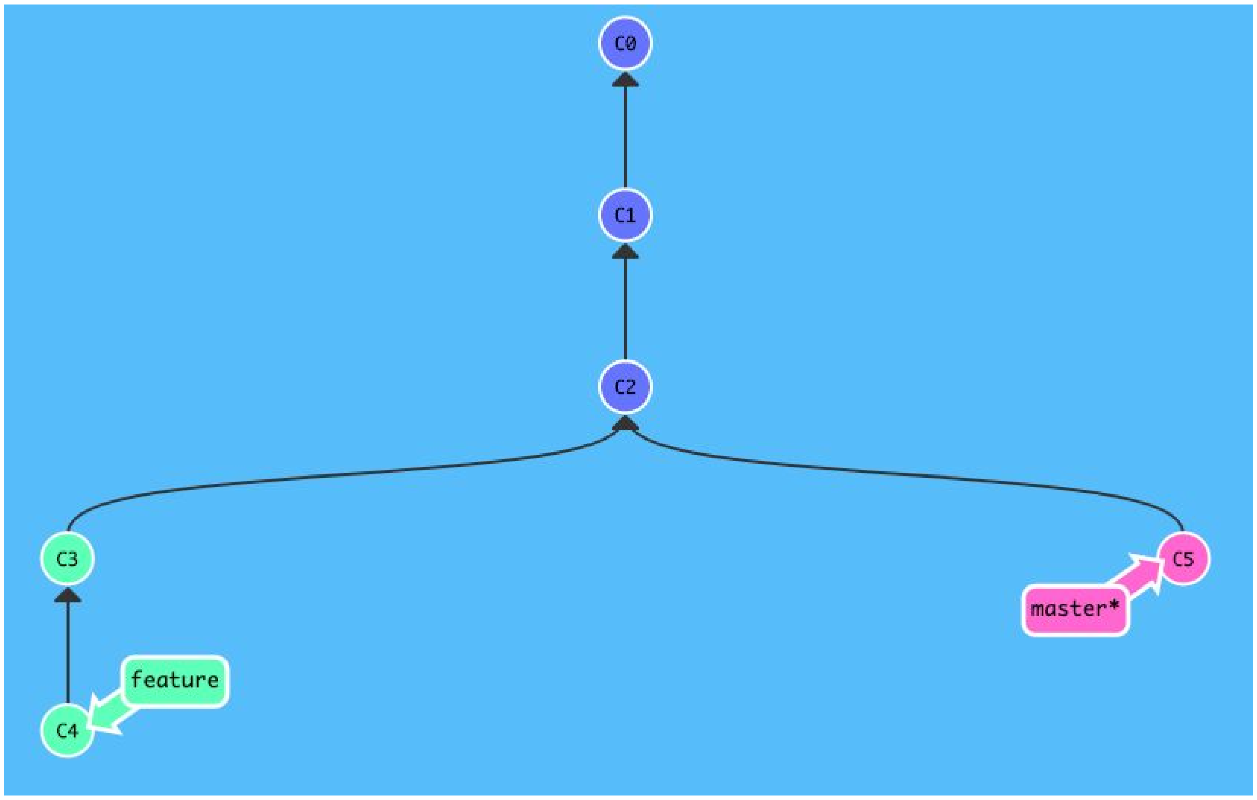


How does Git know what branch you're currently on? It keeps a special pointer called **HEAD**. In Git, this is a pointer to the local branch you're currently on. In this case, you're still on master.

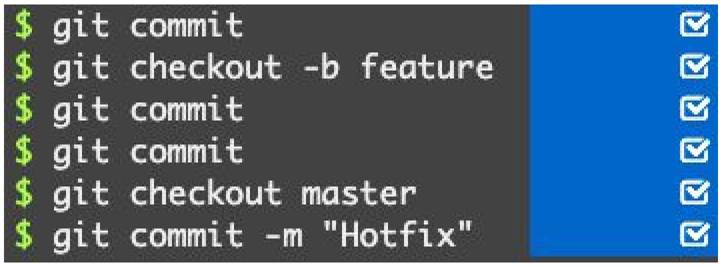
Rebasing vs Merging

The first thing to understand about git rebase is that it solves the same problem as git merge. Both of these commands are designed to integrate changes from one branch into another branch—they just do it in very different ways.
Consider what happens when you start working on a new feature in a dedicated branch, then another team member updates the master branch with new commits. This results in a forked history, which should be familiar to anyone who has used Git as a collaboration tool.

Visualization of the process:



Commands:



```
$ git commit
$ git checkout -b feature
$ git commit
$ git commit
$ git checkout master
$ git commit -m "Hotfix"
```

Now, let’s say that the new commits are in master and you want to add new feature to your master. To incorporate the new commits into your master branch, you have two options: merging or rebasing.

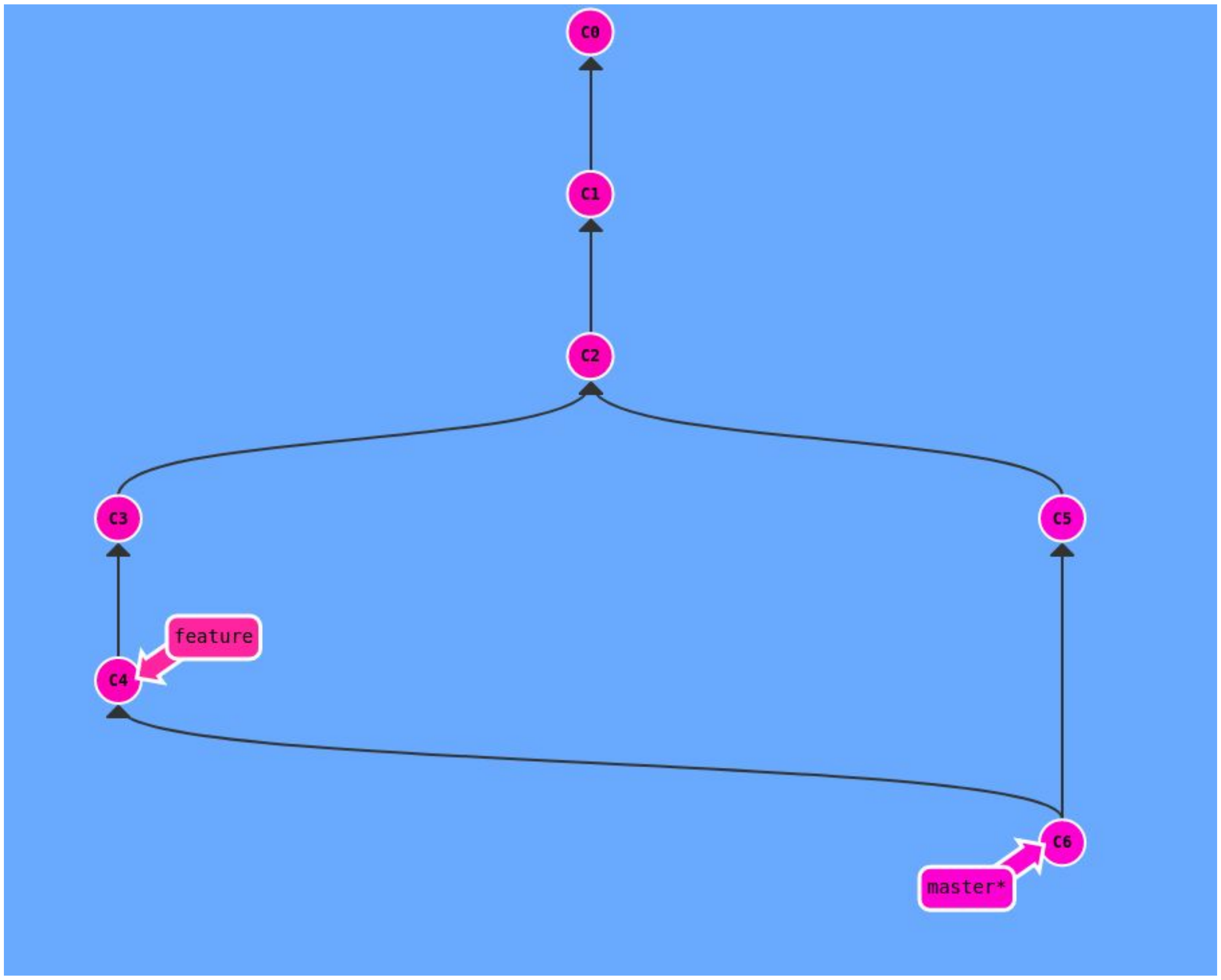
Merge

The easiest option is to merge the **feature** branch into the **master** branch using something like the following:

git checkout master

git merge feature

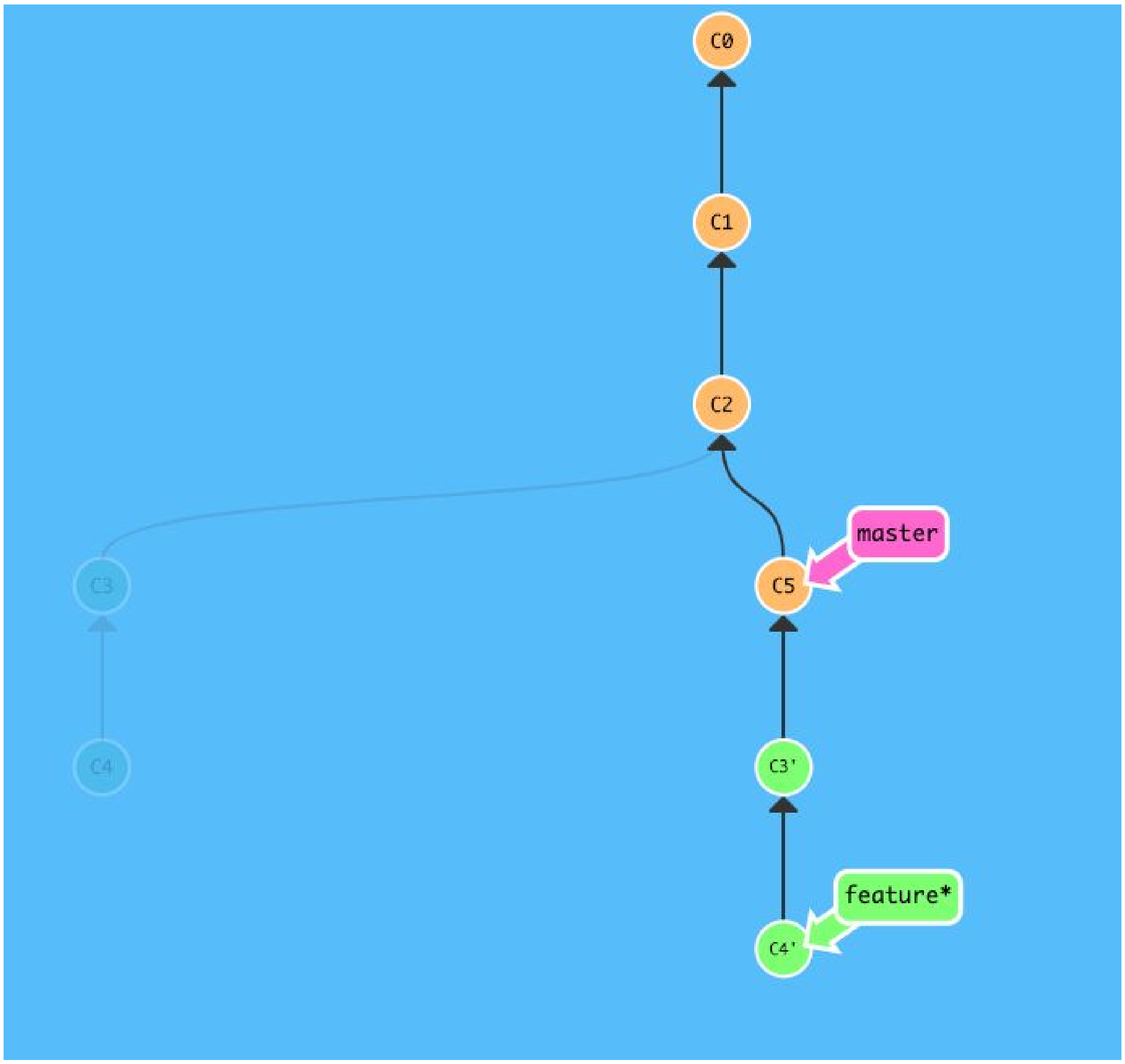
This creates a new "merge commit" in the feature branch that ties together the histories of both branches, giving you a branch structure that looks like this:



Merging is nice because it's a non-destructive operation. The existing branches are not changed in any way. This avoids all of the potential pitfalls of rebasing.
On the other hand, this also means that the **master** branch will have an extraneous merge commit every time you need to incorporate upstream changes.

Rebase

As an alternative to merging, you can rebase the feature branch onto master branch using the following commands:
git checkout feature
git rebase master
This moves the entire feature branch to begin on the tip of the master branch, effectively incorporating all of the new commits in master. But, instead of using a merge commit, rebasing re-writes the project history by creating brand new commits for each commit in the original branch.



The major benefit of rebasing is that you get a much cleaner project history. First, it eliminates the unnecessary merge commits required by git merge. Second, as you can see in the above diagram, rebasing also results in a perfectly linear project history—you can follow the tip of feature all the way to the beginning of the project without any forks. This makes it easier to navigate your project with commands like git log, git bisect, and gitk.
Now we want that master will become the same as the feature branch because we want our new feature in master. We can use following commands:
git checkout master
git merge feature
You'll notice the phrase "fast-forward" in that merge. Because the commit C4' pointed to by the branch feature you merged in was directly ahead of the commit C5 you're on, Git simply moves the pointer forward. To phrase that another way, when you try to merge one commit with a commit that can be reached by following the first commit's history, Git simplifies things by moving the

pointer forward because there is no divergent work to merge together — this is called a "fast-forward."
This gif shows the whole process of rebasing.

Resolve conflict

Let's have a repository with just one file that was changed in commit C2 and C3 based on the next picture.
Merging and rebasing will end in a merge conflict. And file Main.java will be changed to this:
Your next progress should look like this:

1. Edit all files that are in conflict. You should erase all lines and special characters (that git added) according to your knowledge of code. And what code you want to retain.
2. Use git add <filePath> to ensure git that your conflict is resolved.
3. Use git merge/rebase --continue to finish merging or rebasing

Note: Use git status if you don't know your next move. Use merge/rebase --abort to abort merging.

Remote repository

Git collaborative approach to development depends on publishing commits from your local repository for other people to view, fetch, and update.
A remote URL is Git's fancy way of saying "the place where your code is stored." That URL could be your repository on GitHub, or another user's fork, or even on a completely different server.
You can only push to two types of URL addresses:

- An HTTPS URL like https://github.com/user/repo.git

- An SSH URL, like git@github.com:user/repo.git

Git associates a remote URL with a name, and your default remote is usually called origin.

Local git commands

Git init

CMD: git init
Summary: This command creates an empty Git repository - basically a .git directory with subdirectories for objects, refs/heads, refs/tags, and template files. An initial HEAD file that references the HEAD of the master branch is also created.
Documentation: https://git-scm.com/docs/git-init

Git add

CMD: git add [<pathspec>…]
Summary: This command updates the index using the current content found in the working tree, to prepare the content staged for the next commit. It typically adds the current content of existing paths as a whole, but with some options it can also be used to add content with only part of the changes made to the working tree files applied, or remove paths that do not exist in the working tree anymore.
The "index" holds a snapshot of the content of the working tree, and it is this snapshot that is taken as the contents of the next commit. Thus after making any changes to the working tree, and before running the commit command, you must use the add command to add any new or modified files to the index.
This command can be performed multiple times before a commit. It only adds the content of the specified file(s) at the time the add command is run; if you want subsequent changes included in the next commit, then you must run git add again to add the new content to the index.
Documentation: https://git-scm.com/docs/git-add
Useful options:

- git add -A
  - Update the index not only where the working tree has a file matching but also where the index already has an entry. This adds, modifies, and removes index entries to match the working tree.
- git add -u
  - Update the index just where it already has an entry matching . This removes as well as modifies index entries to match the working tree, but adds no new files.

Git status

CMD: git status
Summary: Displays paths that have differences between the index file and the current HEAD commit, paths that have differences between the working tree and the index file, and paths in the working tree that are not tracked by Git. The first are what you would commit by running git commit; the second and third are what you could commit by running git add before running git commit.
Documentation: https://git-scm.com/docs/git-init
Useful options:

- -s --short
  - Give the output in the short-format.

Git commit

CMD: git commit
Summary: Create a new commit containing the current contents of the index and the given log message describing the changes. The new commit is a direct child of HEAD, usually the tip of the current branch, and the branch is updated to point to it (unless no branch is associated with the working tree, in which case HEAD is "detached").
Documentation: https://git-scm.com/docs/git-commit
Useful options:

- git status -m <msg>
  - Use the given as the commit message. If multiple -m options are given, their values are concatenated as separate paragraphs.

Git branch

CMD: git branch git branch <branchname>
Summary:
If --list is given, or if there are no non-option arguments, existing branches are listed; the current branch will be highlighted in green and marked with an asterisk.
The commands second form creates a new branch head named which points to the current HEAD. Note that this will create the new branch, but it will not switch the working tree to it; use "git switch " to switch to the new branch.
Documentation: https://git-scm.com/docs/git-branch
Useful options:

- git branch -d git branch --delete
  - Delete a branch. The branch must be fully merged in its upstream branch, or in HEAD if no upstream was set.
- git branch -a git branch --all
  - List both remote-tracking branches and local branches.

Git checkout

CMD: git checkout [<branch>]
Summary: To prepare for working on , switch to it by updating the index and the files in the working tree, and by pointing HEAD at the branch. Local modifications to the files in the working tree are kept, so that they can be committed to the .

CMD: git checkout <commit>
Summary: Prepare to work on top of , by detaching HEAD at it (see "DETACHED HEAD" section), and updating the index and the files in the working tree. Local modifications to the files in the working tree are kept, so that the resulting working tree will be the state recorded in the commit plus the local modifications.
CMD: git checkout -b <branch>
Summary: Specifying -b causes a new branch to be created as if git branch was called and then checked out. This is the transactional equivalent of :
$ git branch <branch>
$ git checkout <branch>
Documentation: https://git-scm.com/docs/git-checkout
Useful options:

- git checkout -f <branch> git checkout --force <branch>
  - When switching branches, proceed even if the index or the working tree differs from HEAD. This is used to throw away local changes. When checking out paths from the index, do not fail upon unmerged entries; instead, unmerged entries are ignored.

Git merge

CMD: git merge <branch>
Summary: Incorporates changes from the named commits (since the time their histories diverged from the current branch) into the current branch. This command is used by git pull to incorporate changes from another repository and can be used by hand to merge changes from one branch into another.
The syntax git merge --abort can only be run after the merge has resulted in conflicts. git merge --abort will abort the merge process and try to reconstruct the pre-merge state. However, if there were uncommitted changes when the merge started (and especially if those changes were further modified after the merge was started), git merge --abort will in some cases be unable to reconstruct the original (pre-merge) changes. Therefore:
Warning: Running git merge with non-trivial uncommitted changes is discouraged: while possible, it may leave you in a state that is hard to back out of in the case of a conflict.
The syntax (git merge --continue) can only be run after the merge has resulted in conflicts.
Documentation: https://git-scm.com/docs/git-merge
Useful options:

- --ff
  - When the merge resolves as a fast-forward, only update the branch pointer, without creating a merge commit. This is the default behavior.

Git rebase

CMD: git rebase <branch> <upstream>
Summary: <upstream> is branch to compare against. May be any valid commit, not just an existing branch name. Defaults to the configured upstream for the current branch. <branch> is Working branch; defaults to HEAD. All changes made by commits in the current branch but that are not in are saved to a temporary area. The commits that were previously saved into the temporary area are then reapplied to the current branch, one by one, in order.
Documentation: https://git-scm.com/docs/git-rebase
Useful options:

- --continue
  - Restart the rebasing process after having resolved a merge conflict.
- --abort
  - Abort the rebase operation and reset HEAD to the original branch. If was provided when the rebase operation was started, then HEAD will be reset to . Otherwise HEAD will be reset to where it was when the rebase operation was started.
- --quit
  - Abort the rebase operation but HEAD is not reset back to the original branch. The index and working tree are also left unchanged as a result.

Git reset

CMD: git reset [<mode>] [<commit>]
Summary: This form resets the current branch head to and possibly updates the index (resetting it to the tree of <commit>) and the working tree depending on <mode>. If <mode> is omitted, defaults to --mixed. The <mode> must be one of the following:

- --soft
  - Does not touch the index file or the working tree at all (but resets the head to <commit>, just like all modes do). This leaves all your changed files "Changes to be committed", as git status would put it.
- --mixed
  - Resets the index but not the working tree (i.e., the changed files are preserved but not marked for commit) and reports what has not been updated. This is the default action.

- --hard
    - Resets the index and working tree. Any changes to tracked files in the working tree since are discarded.

Documentation: https://git-scm.com/docs/git-reset

Git revert

CMD: git revert <commit>
Summary: Given one or more existing commits, revert the changes that the related patches introduce, and record some new commits that record them. This requires your working tree to be clean (no modifications from the HEAD commit). git revert is used to record some new commits to reverse the effect of some earlier commits (often only a faulty one).
Documentation: https://git-scm.com/docs/git-revert
Useful options:

Remote commands

Git clone

CMD: git clone <repository>
Summary: Clones a repository into a newly created directory, creates remote-tracking branches for each branch in the cloned repository (visible using git branch --remotes), and creates and checks out an initial branch that is forked from the cloned repository's currently active branch.
After the clone, a plain git fetch without arguments will update all the remote-tracking branches, and a git pull without arguments will in addition merge the remote master branch into the current master branch, if any (this is untrue when "--single-branch" is given; see below).
This default configuration is achieved by creating references to the remote branch heads under refs/remotes/origin and by initializing remote.origin.url and remote.origin.fetch configuration variables.
Documentation: https://git-scm.com/docs/git-clone

Git fetch

CMD: git fetch
Summary: Fetch branches and/or tags (collectively, "refs") from one or more other repositories, along with the objects necessary to complete their histories. Remote-tracking branches are updated (see the description of below for ways to control this behavior).
When no remote is specified, by default the origin remote will be used, unless there's an upstream branch configured for the current branch. **Documentation:** https://git-scm.com/docs/git-fetch

Git pull

CMD: git pull
Summary: Incorporates changes from a remote repository into the current branch. In its default mode, git pull is shorthand for git fetch followed by git merge FETCH_HEAD.
More precisely, git pull runs git fetch with the given parameters and calls git merge to merge the retrieved branch heads into the current branch.
Documentation: https://git-scm.com/docs/git-pull

Git push

CMD: git push <remote>
Summary: Updates remote refs using local refs, while sending objects necessary to complete the given refs.
Documentation: https://git-scm.com/docs/git-push