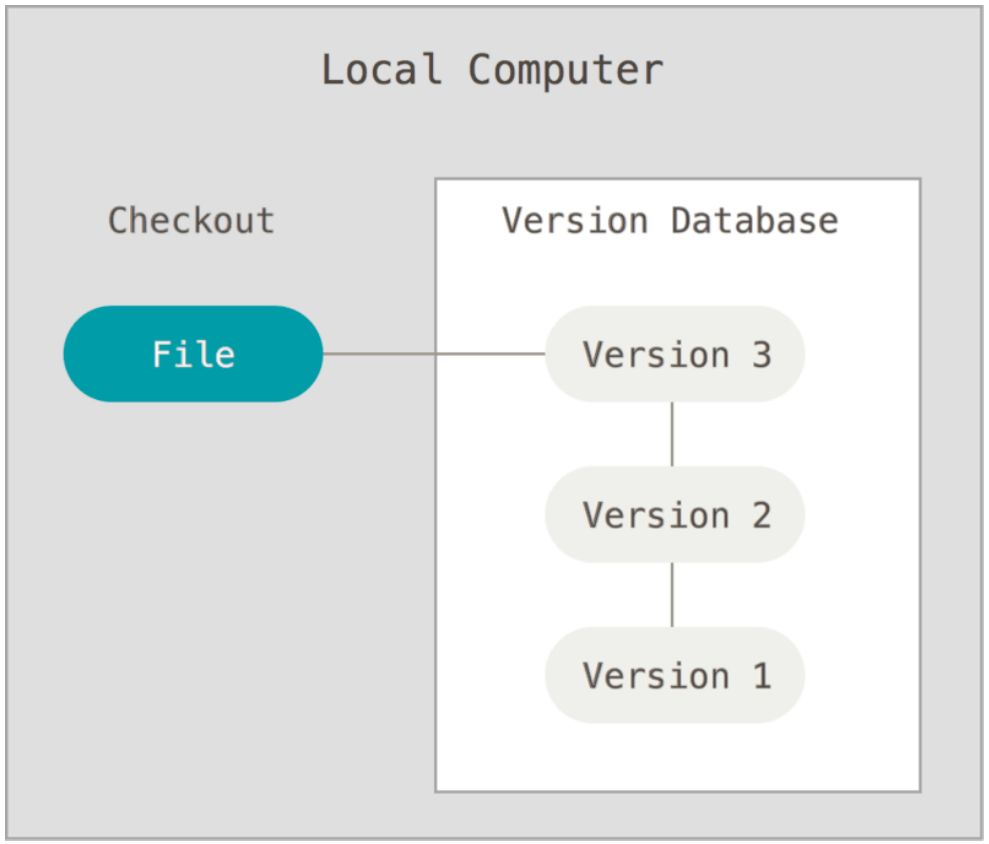
Getting started with GIT

https://git-scm.com

Illustrations in this presentation are mostly extracted from https://git-scm.com/book/en/v2

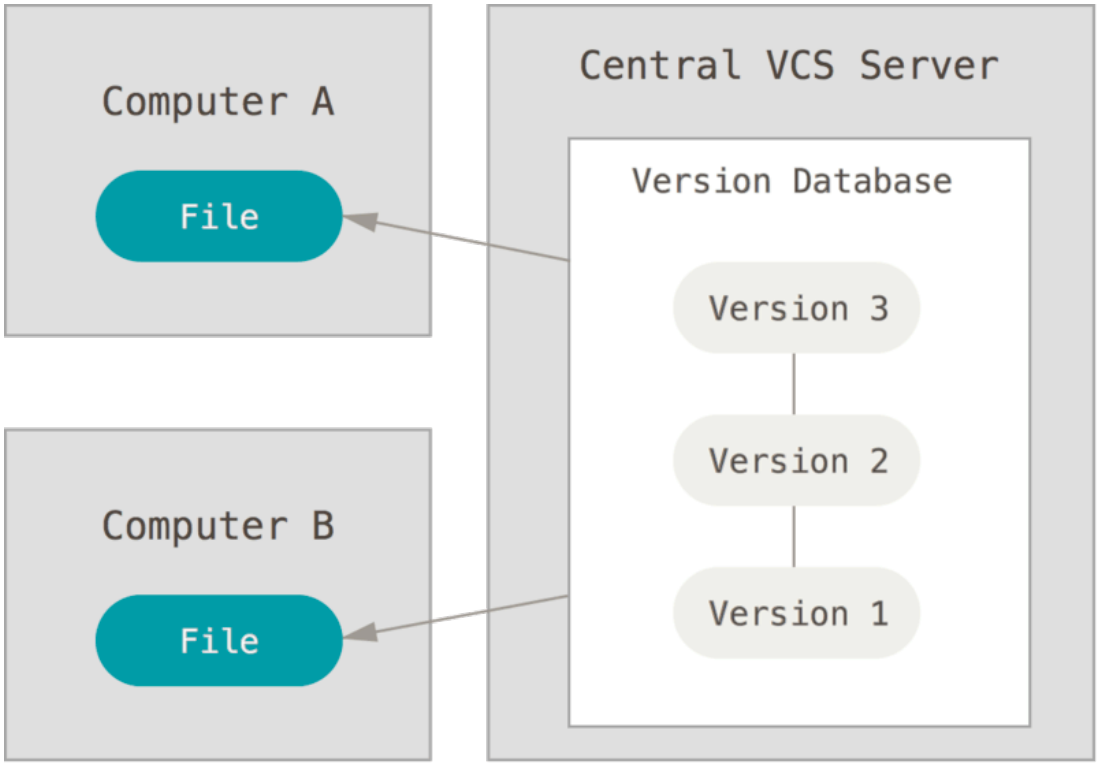
About version control

- Getting history



About version control

- Working with a team
- Collaborate with other developers
- Organizing work



Comparing files

Create both files FileV1.html and FileV2.html in your favorite editor

FileV1.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
</head>
<body>

</body>
</html>
```

FileV2.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>My page</title>
</head>
<body>
Hello everyone!
</body>
</html>
```

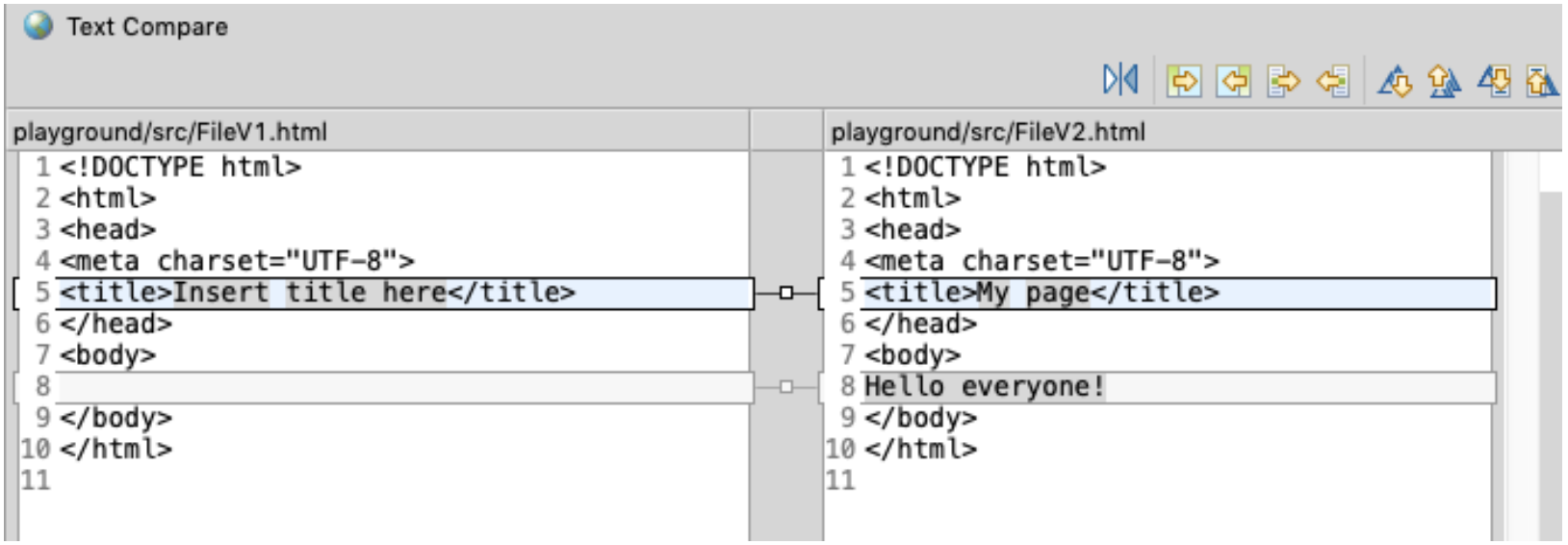
Understanding files compare

In Windows you can compare both versions of your file with the command:

```
FC FileV1.html FileV2.html
```

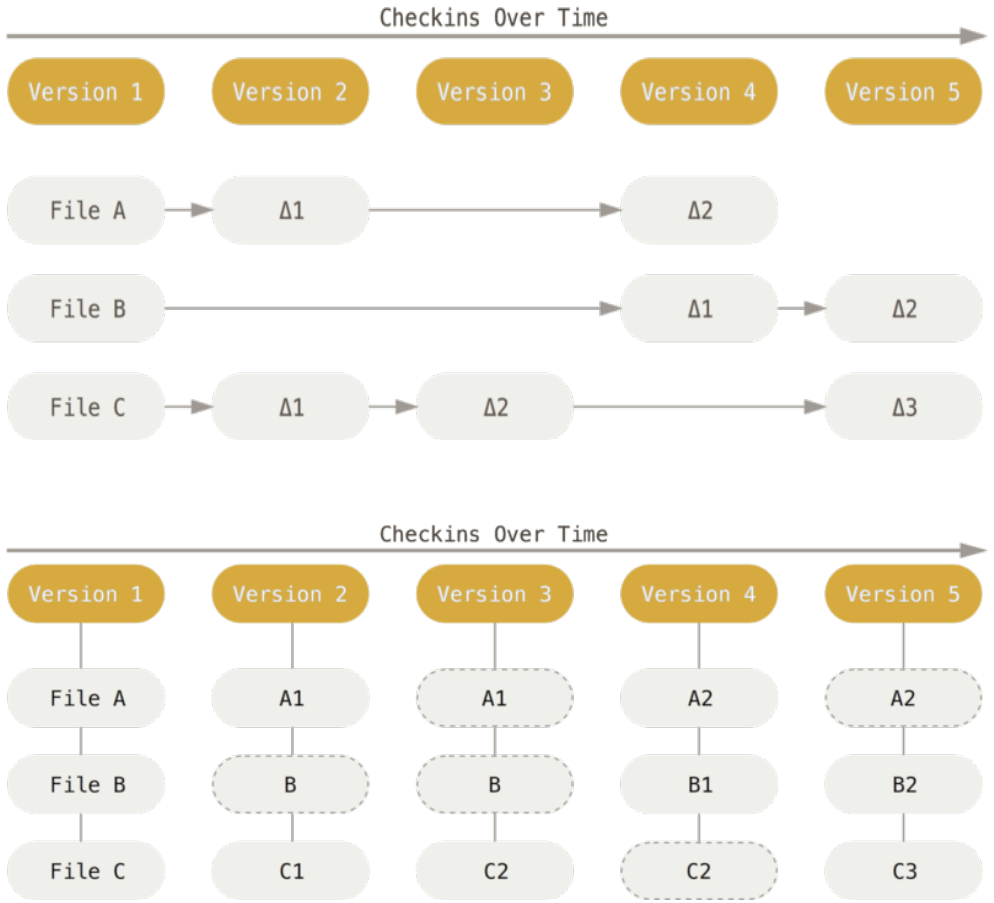
In Mac you can compare both versions of your file with the command:

```
diff FileV1.html FileV2.html
```



GIT vision of version control

“Git thinks of 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: goals and short history

- GIT is one of the most popular and mature version control tool
- Originally grown from Linux Development Community (and in particular Linus Torvalds, creator of Linux)
- Original goals were:
  - Speed
  - Simple design
  - Strong support for non-linear development (thousands of parallel branches)
  - Fully distributed
  - Able to handle large projects like the Linux kernel efficiently (speed and data size)
- Reference book: “Pro Git” (Scott Chacon and Ben Straub)
  https://git-scm.com/book/en/v2

Installing GIT

Git may be already installed in your computer (or not !)

Windows:

1. Navigate to https://git-scm.com/ downloads with your favorite browser.
2. Click Windows and wait for the download to start.
3. Once the file has been downloaded, open it and follow the install procedure.
4. Git will now be available in the machine along with Git Bash, allowing one to make use of git just as one would on any unix machine.
5. To get colored Git output, run git config --global color.ui auto

Mac OS X:

Option 1 (Official Git .dmg)

1. Navigate to https://git-scm.com/downloads with your favorite browser
2. Click Mac OS X and wait for the download to start
3. Once the file has been downloaded, open it and follow the prompts
4. To get colored Git output, run git config --global color.ui auto

Option 2 (With Homebrew already installed)

1. Run the command brew install git
2. To get colored Git output, run git config --global color.ui auto

GIT in action (1)

Once Git is in place, one may now create repositories out of any folder and easily manage the versioning of any project one may develop in place. In order to have similar results as before with diff and FC we must run through a somewhat similar process. However, Git has some quite significant advantages that make the process of managing a codebase far easier in real use-cases. Let us now run through the previous example with Git so that these differences make themselves manifest.

First **create** a folder where you will store your codebase.
This will now become your local repository and will be where Git will be set up.
You must now **initialize** your repository with Git, so that any files or folders contained within can be tracked.
This can be done as follows:

```
$ cd /some/parent/directory/
$ mkdir myrepo
$ cd myrepo
$ git init
```

Then use your favorite editor and create file **`file.html`**

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
</head>
<body>
</body>
</html>
```

GIT in action (2)

Now check the status of the repository and you will see this output:

```
$ git status
On branch master

No commits yet

Untracked files:
(use "git add <file>..." to include in what will be committed)

    file.html

nothing added to commit but untracked files present (use "git add" to track)
```

This means that Git is aware that new elements are present within your repository but it has not yet been asked to keep track of them.
With that in mind, proceed to add the file to the staging area (Git's cache saved in the binary index file) and then commit the changes, essentially telling Git to create a snapshot of the state of the repository (local repository).

GIT in action (3)

Run the following commands:

```
$ git add file.html
$ git commit -m "added file.html"
```

Once this has been done, Git is now conscious of both the existence of **`file.html`** and of the state it was in during this commit.
Now, open **`file.html`** with your preferred editor and write “Hello world!” between the <body></body> tags.
Once done, please query Git about the status of your repository.

```
$ 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:   file.html

no changes added to commit (use "git add" and/or "git commit -a")
```

GIT in action (4)

We can see that Git is very much aware of the changes made to the file (comparing it with its last commit) and, crucially, there is no need to duplicate the file as Git will enable on to keep a continuous stream of changes to a single file as its progression through the commits where it is changed.
This becomes evident when we ask Git to show us what changes it has identified.

```
$ git diff file.html
diff --git a/file.html b/file.html
index 1441a85..b76435f 100644
--- a/file.html
+++ b/file.html
@@ -5,6 +5,6 @@
 <title>Insert title here</title>
 </head>
 <body>
-
+Hello world!
 </body>
 </html>
```

GIT in action (5)

If we then add the file to our next commit and then do the commit the state of Git's snapshot of the directory will now be brought up to date.

We can do this just as before (with one simple difference that is worth mentioning: if we give . as the argument to git, it will add every file in the main directory's underlying folder structure, thus removing the need to list out every file we want to commit):

```
$ git add .
$ git commit -m "added a body to the file"
```

GIT in action (6)

Finally we can now inspect how Git is tracking the changes made to the repository:

```
$ git log
commit a9b50797ee1c0f9847c47e659e990a57409abba4 (HEAD -> master)
Author: user <user@email.com>
Date:   Sun Jul 28 12:07:59 2019 +0200

    added a body to the file

commit d3e90205d172f1505286bbe6fbd49f7eff1b8590
Author: user <user@email.com>
Date:   Sun Jul 28 11:45:19 2019 +0200

    added file.html
```

And you can even query Git for the changes in your repository from two different versions (notice that each version has an identifier and git log gives the versions in order, from newest to oldest):

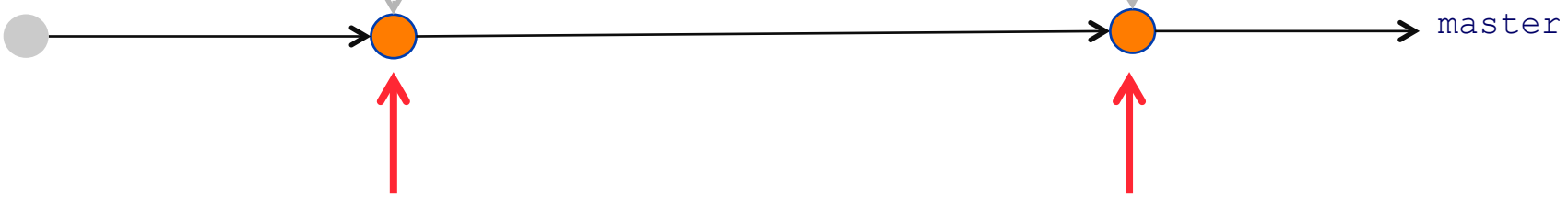
```
$ git diff d3e90205d172f1505286bbe6fbd49f7eff1b8590
a9b50797ee1c0f9847c47e659e990a57409abba4
```

a9b50797ee1c0f9847c47e659e990a57409abba4

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
</head>
<body>
</body>
</html>
```

d3e90205d172f1505286bbe6fbd49f7eff1b8590

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
</head>
<body>
Hello world!
</body>
</html>
```

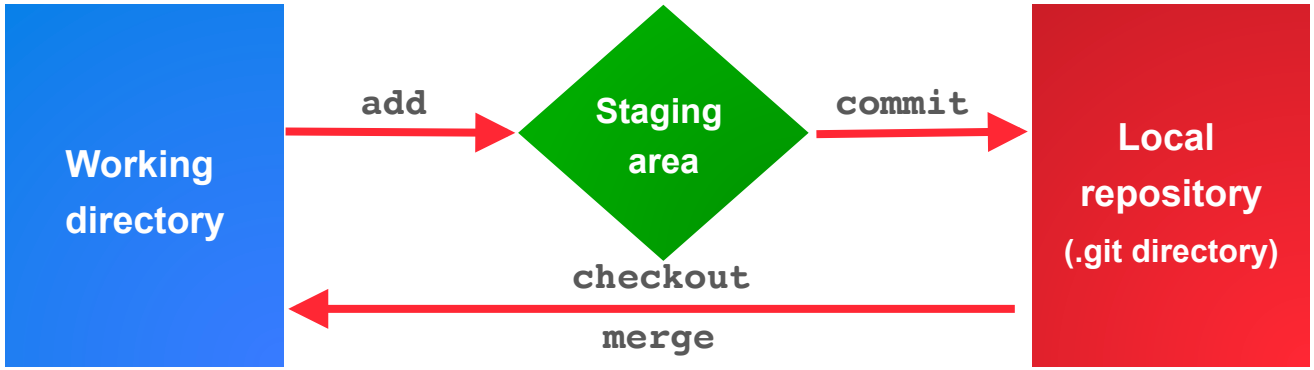



```
$ git add file.html
$ git commit -m "added file.html"
```

```
$ git add .
$ git commit -m "added a body to the file"
```

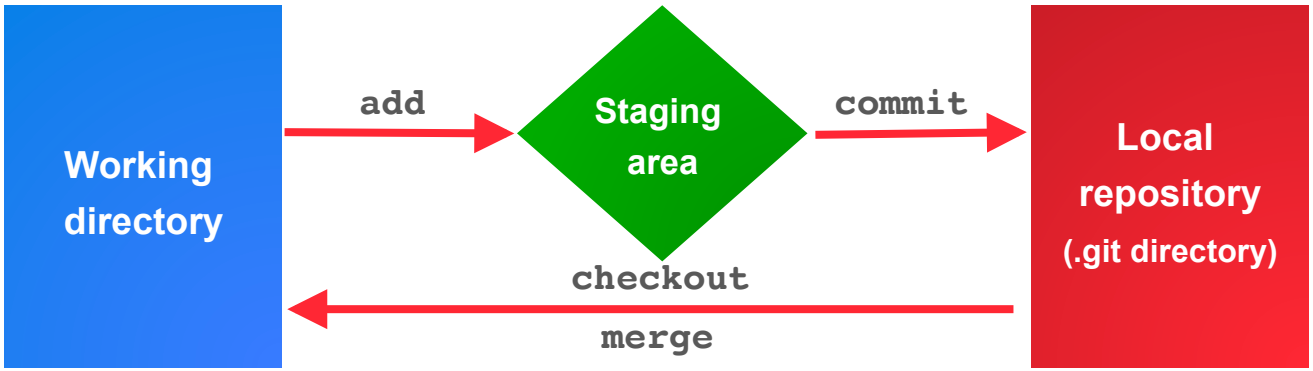
- **`Modified`** means that you have changed the file but have not committed it to your database yet.
- **`Staged`** means that you have marked a modified file in its current version to go into your next commit snapshot.
- **`Committed`** means that the data is safely stored in your local database.

Git has three main states that your files can reside in: **`modified`**, **`staged`**, and **`committed`**:

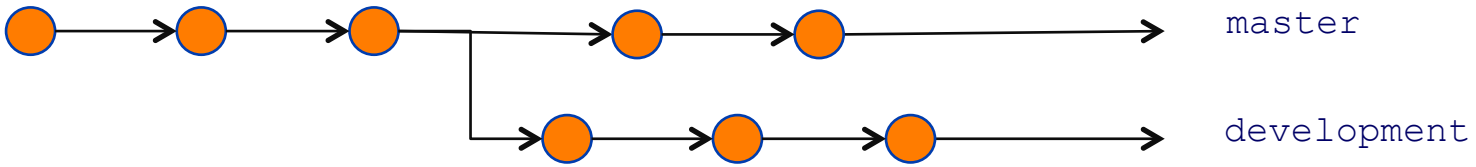


Basic commands

```
git add
git commit
git branch
git checkout
git merge
```



Branching (1)



Now, suppose you would like to add an experimental feature to your program, but would rather test it out first before adding it to your codebase (for example, you could have two possible solutions to a problem and would like to test them both out before committing to one).

This can be accomplished by making use of several of Git's features.

The first of these is its ability to create branches. To create a new branch you must run the following command (please note that you may use any name, except that of other branches that already exist):

```
$ git branch development
```

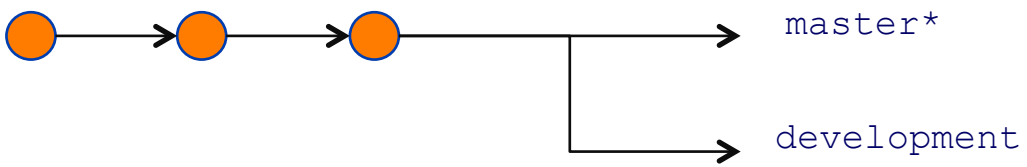
Branching (2)

You can now check that this new branch exists by running the **`git branch command`**.
The output should look like this:

```
$ git branch
  development
* master
```

This would indicate that there are two available branches in the local repository and shows a star (*) besides the currently active branch.

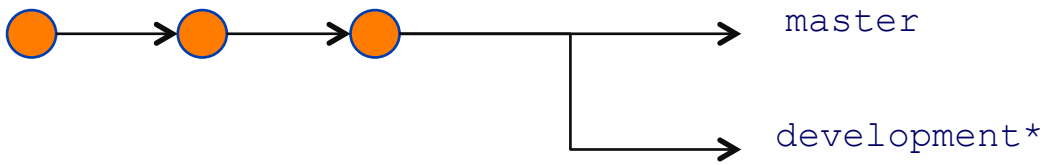
It is worth notice that branches are created in such a way that they start out at the active branch's currently active commit and as such allow you to diverge from the current state of the directory while allowing the other branch to evolve separately if need be.



Branching (3)

With all this said and done, in order **to switch to the newly created branch** run the following command:

```
$ git checkout development
```



This command will, thus, change the currently active branch and make it so that any new commits that are added will now belong to the newly selected branch.

Evidently one may switch back and forth between branches at will by changing the parameter of the git checkout command. It is also worth noting that in practice one usually desires **to both create a branch and immediately switch to it**, and, fortunately, git provides a convenient command that does exactly that:

```
$ git checkout -b <new_branch_name>
```

Branching (4)

We can now proceed to make any changes desired to the software.

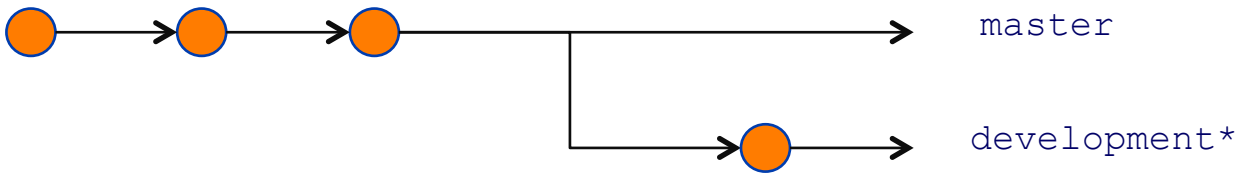
In this case, let's assume you would like to **test out the change of bolding the phrase that is to be displayed**. Now proceed to change the file so that it looks as follows with your favorite editor.

With this done you may now commit this change just as you did before, but this time this change will belong to the development branch.

Run the following commands to do so:

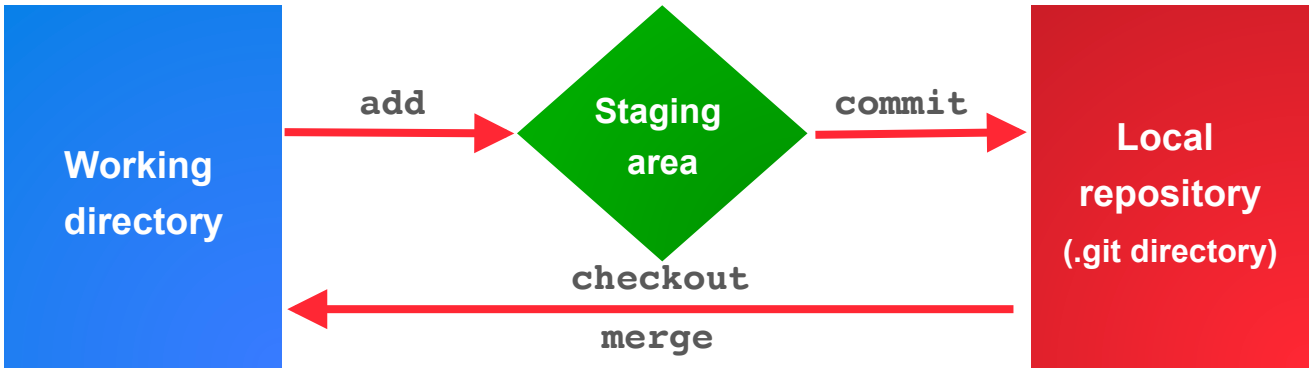
```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
</head>
<body>
<b>
Hello world!
</b>
</body>
</html>
```

```
$ git add .
$ git commit -m "added bold tags"
```



Basic commands

```
git add
git commit
git branch
git checkout
git merge
```



Merging (1)

Now, assuming you have tested these changes and have decided that they are worthy of adding to your main (master) branch, you will need to do a merge of your changes.

First change back to your master branch by running the following command:

```
$ git checkout master
```

You may notice that **the changes you have made to your files are no longer present. However, there is no need to worry, these changes are stored as the commits that you have made on your development branch**. Since these changes were not made on the master branch, you will not see them reflected on the files. In order to bring these changes to your master branch, you will need then to merge your development branch into your master branch.

In other words, **you will take all the commits that have been made to the development branch since it was created and add them to the master branch so that it now reflects the changes that were made on the other branch**.

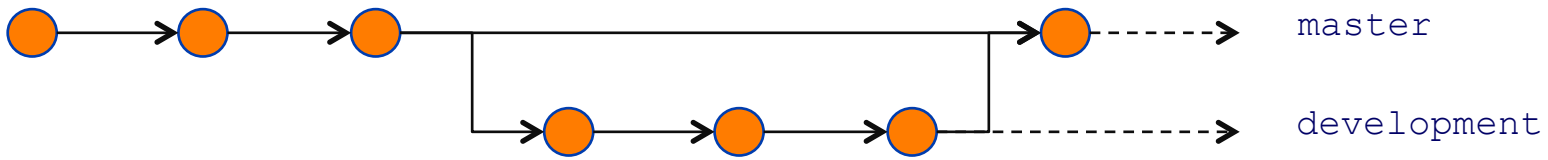
To do this run the following command:

```
$ git merge development
```

Merging (2)

You should see output similar to this:

```
$ git merge development
Updating 3372db0..b1a764f
Fast-forward
 file.html | 2 ++
 1 file changed, 2 insertions(+)
```



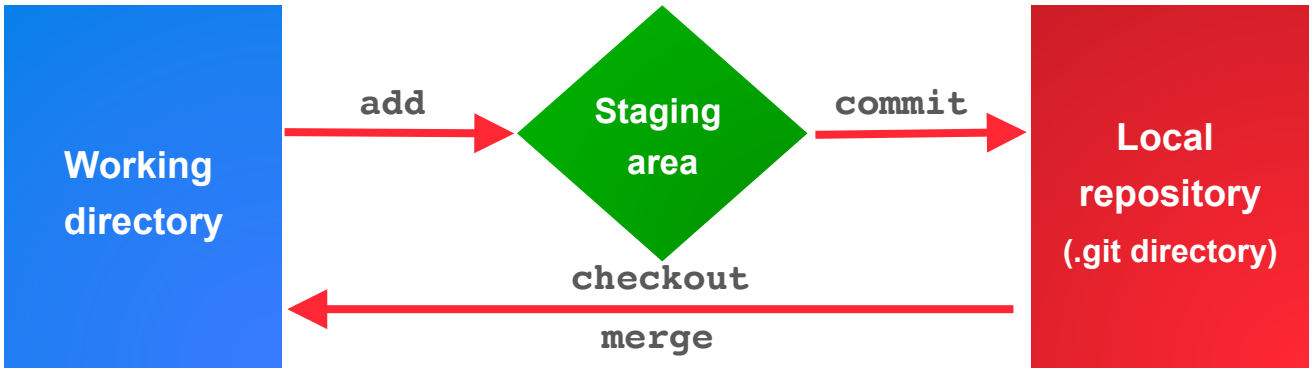
This indicates that Git was able to successfully bring your master branch forward to where the development branch was.

It will not always be that simple and on occasion **conflicts** will emerge such that Git will be unable to easily merge your two branches and you will need to manually make the changes and create a new commit that reflects the changes made so that the two branches are now up to date.

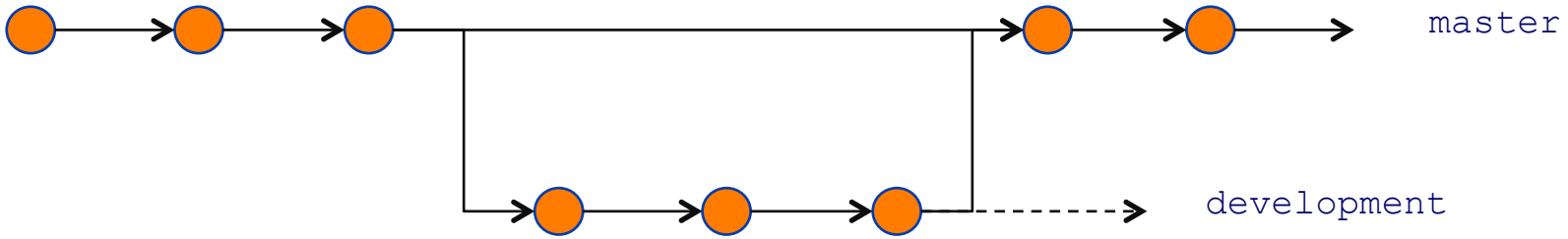
However, in this case, given the nature of the changes and the branches, it conformed to the simplest of cases.

Basic commands

```
git add
git commit
git branch
git checkout
git merge
```



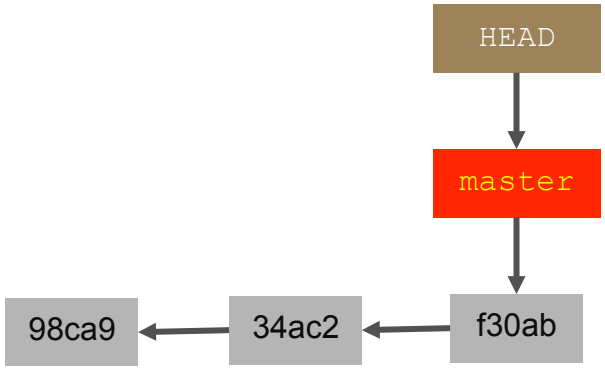
About branching and merge



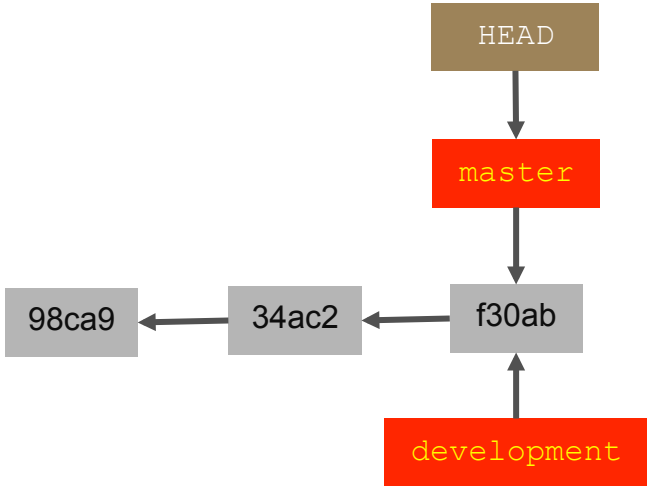
Classical vision of branching and merge: a way to represent what we just have done.

GIT vision of branching: simple branch (1)

```
$ git branch development
```



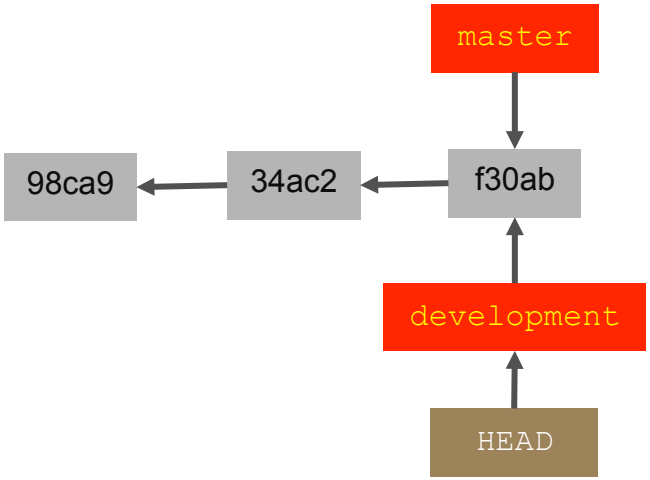
Three commits on **master** branch.
Working directory on **master** branch.



Branch **development** was created.
Working directory is still on **master** branch.

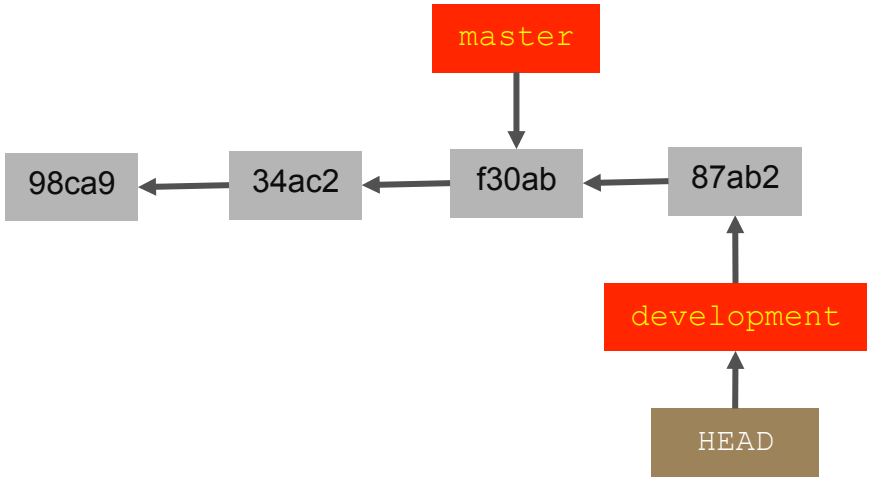
GIT vision of branching: simple branch (2)

```
$ git checkout development
```



Working directory is now on **development** branch.

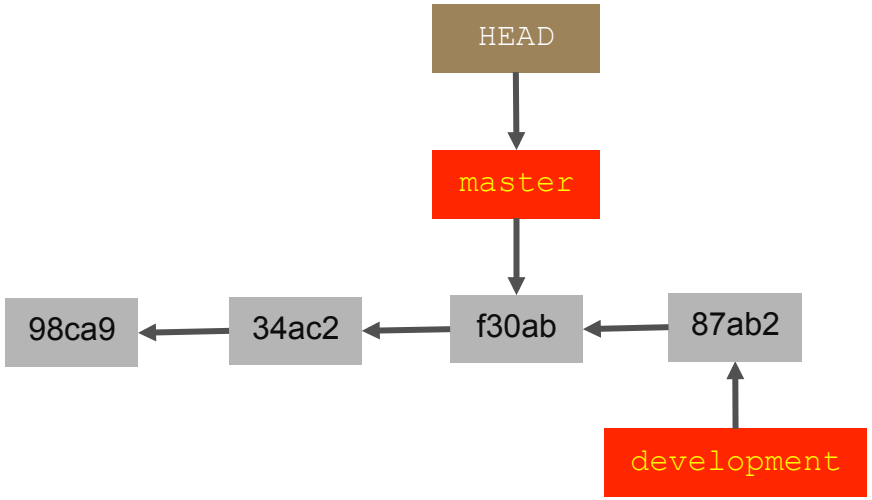
```
$ nano file.html
$ git commit -m 'made a change file.html'
```



development branch is now one commit ahead

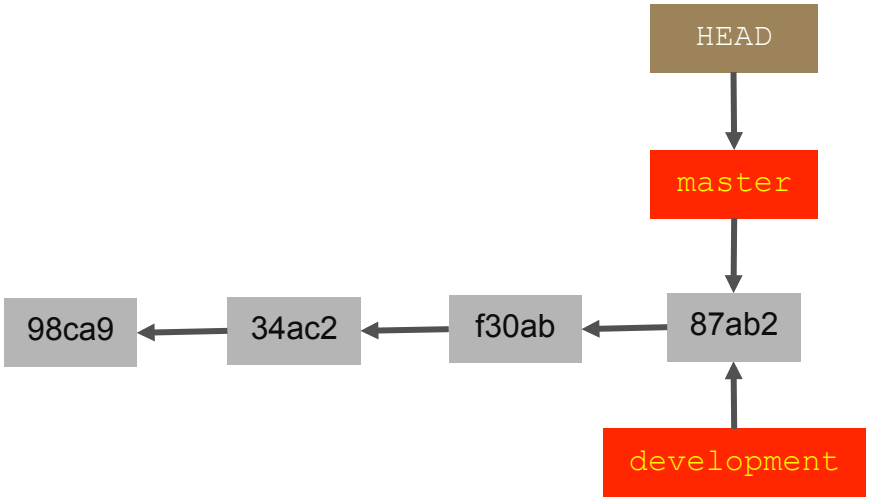
GIT vision of branching: fast-forward merge

```
$ git checkout master
```



Switched back to **`master`** branch.

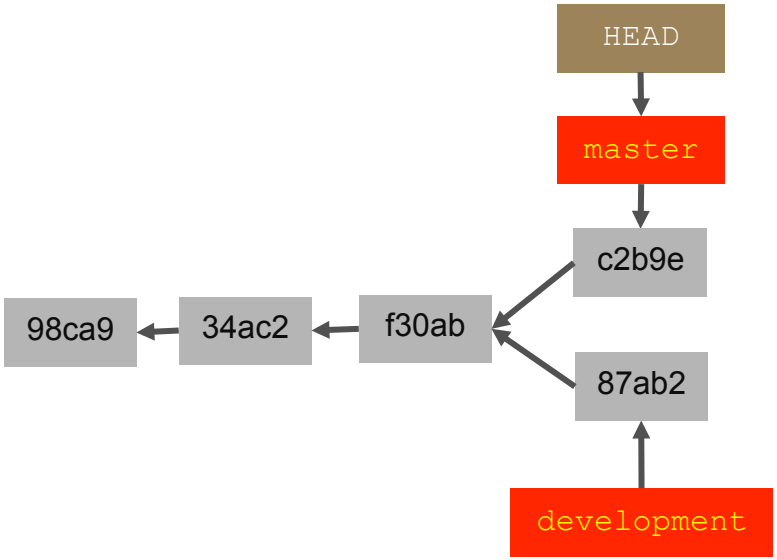
```
$ git merge development
```



Merging is trivial since it's a basic pointer move.
We call it fast forward.

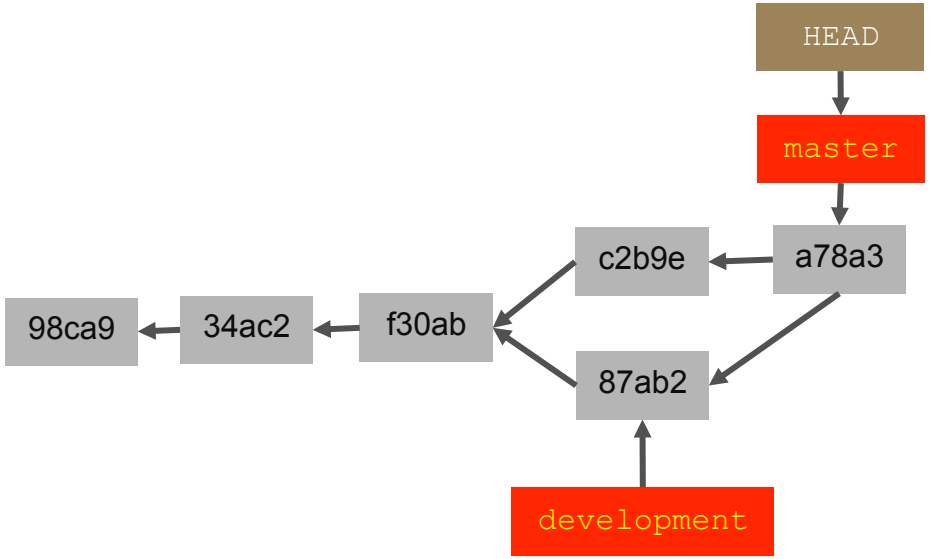
GIT vision of branching: basic three-way merge

```
$ nano file2.html
$ git commit -a -m 'added a new file'
```



Imagine now we have made a concurrent edit on **`master`** branch: the two branches are now divergent.

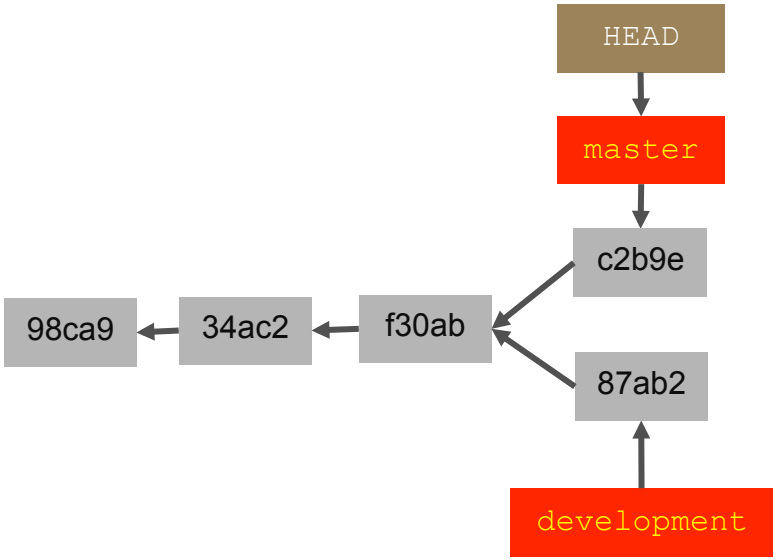
```
$ git merge development
Merge made by the 'recursive' strategy.
file2.html |    1 +
1 file changed, 1 insertion(+)
```



In this case, Git does a simple three-way merge, using the two snapshots pointed to by the branch tips and the common ancestor of the two.

GIT vision of branching: conflicting merge

```
$ nano file.html
$ git commit -a -m 'modified file.html'
```



We have edited the same file **`file.html`** on both branches. Merging is no more trivial anymore.

```
$ git merge development
Auto-merging file.html
CONFLICT (content): Merge conflict in
file.html
Automatic merge failed; fix conflicts
and then commit the result.
```

Auto-merging failed, and you have to resolve the conflict by hand.

```
$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")
Unmerged paths:
  (use "git add <file>..." to mark
resolution)
    both modified:      file.html
no changes added to commit (use "git
add" and/or "git commit -a")
```

GIT vision of branching: fix conflicting merge

Anything that has merge conflicts and hasn't been resolved is listed as unmerged. Git adds standard conflict-resolution markers to the files that have conflicts, so you can open them manually and resolve those conflicts. Your file contains a section that looks something like this:

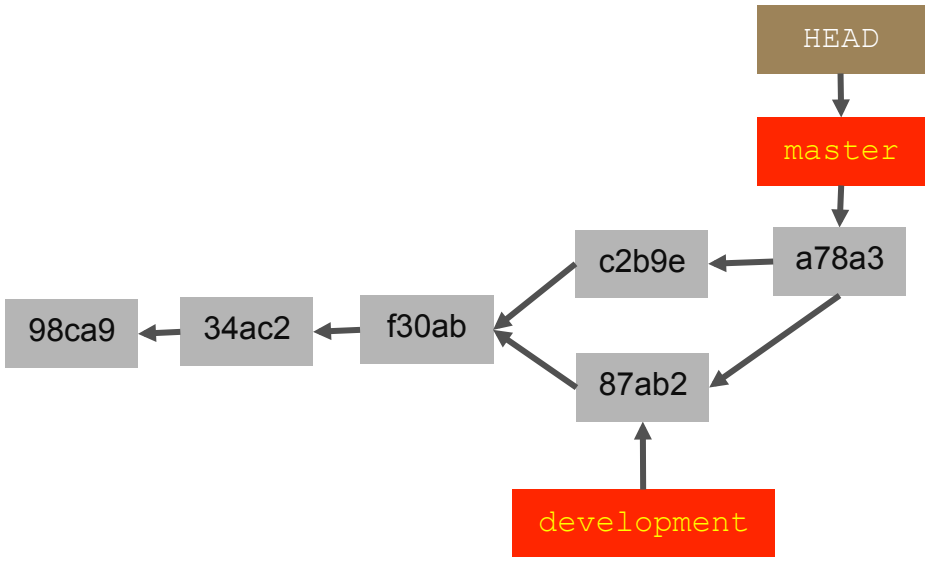
```
<<<<<<< HEAD:file.html
<div id="footer">contact :
email.support@github.com</div>
=======
<div id="footer">
 please contact us at support@github.com
</div>
>>>>>>> development:file.html
```

You might resolve this conflict by replacing the entire block with this:

```
<div id="footer">
please contact us at email.support@github.com
</div>
```

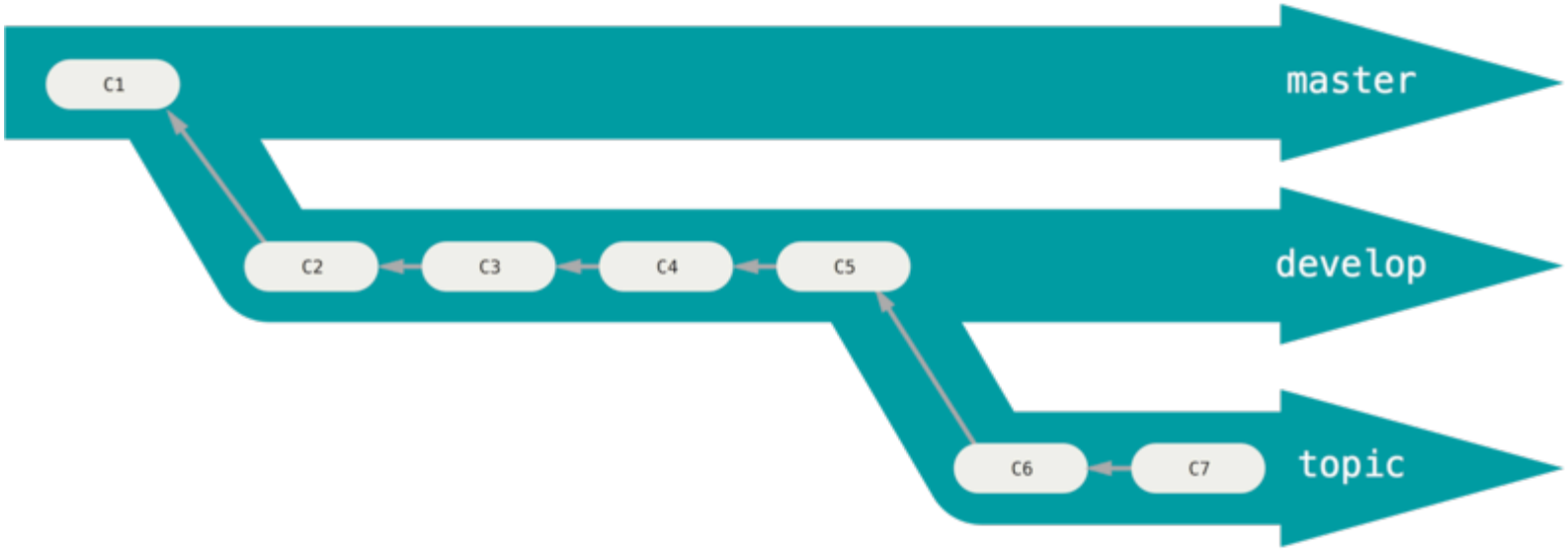
Conclude merge by committing.

```
$ git add file.html
$ git commit -a -m 'Merge with development'
```



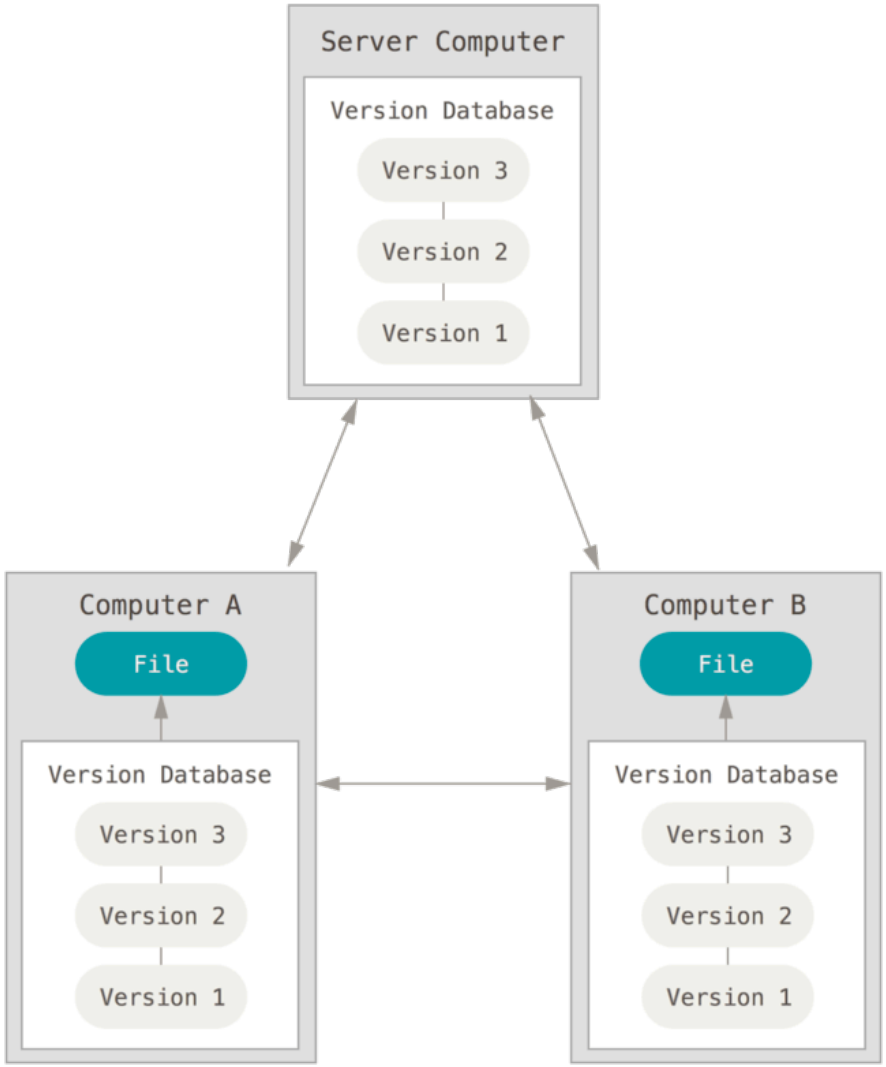
At the end, you get a similar network.

GIT vision of branching



Back on Git features

- Snapshots, not differences
- Git has integrity
- Git generally only adds data
- The three states
- A distributed system
- Nearly every operation is local



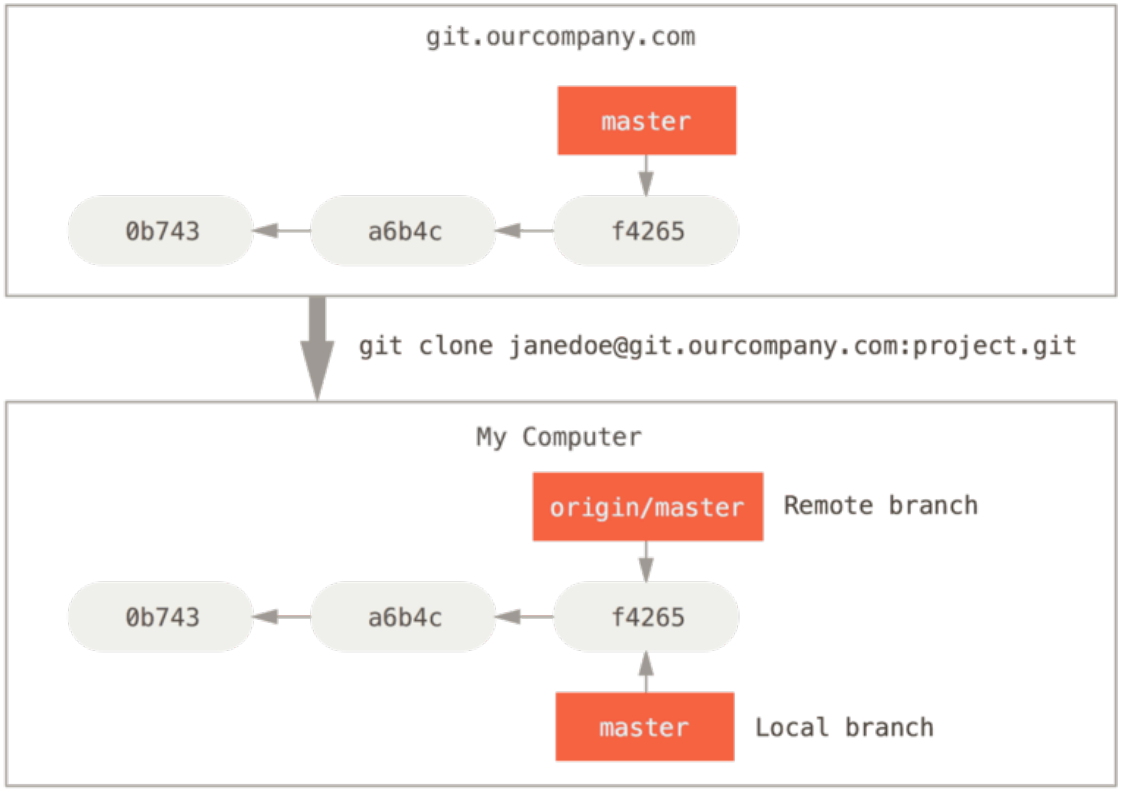
Git branching : remote branches

Git is a distributed system and allows to work with remote repositories.

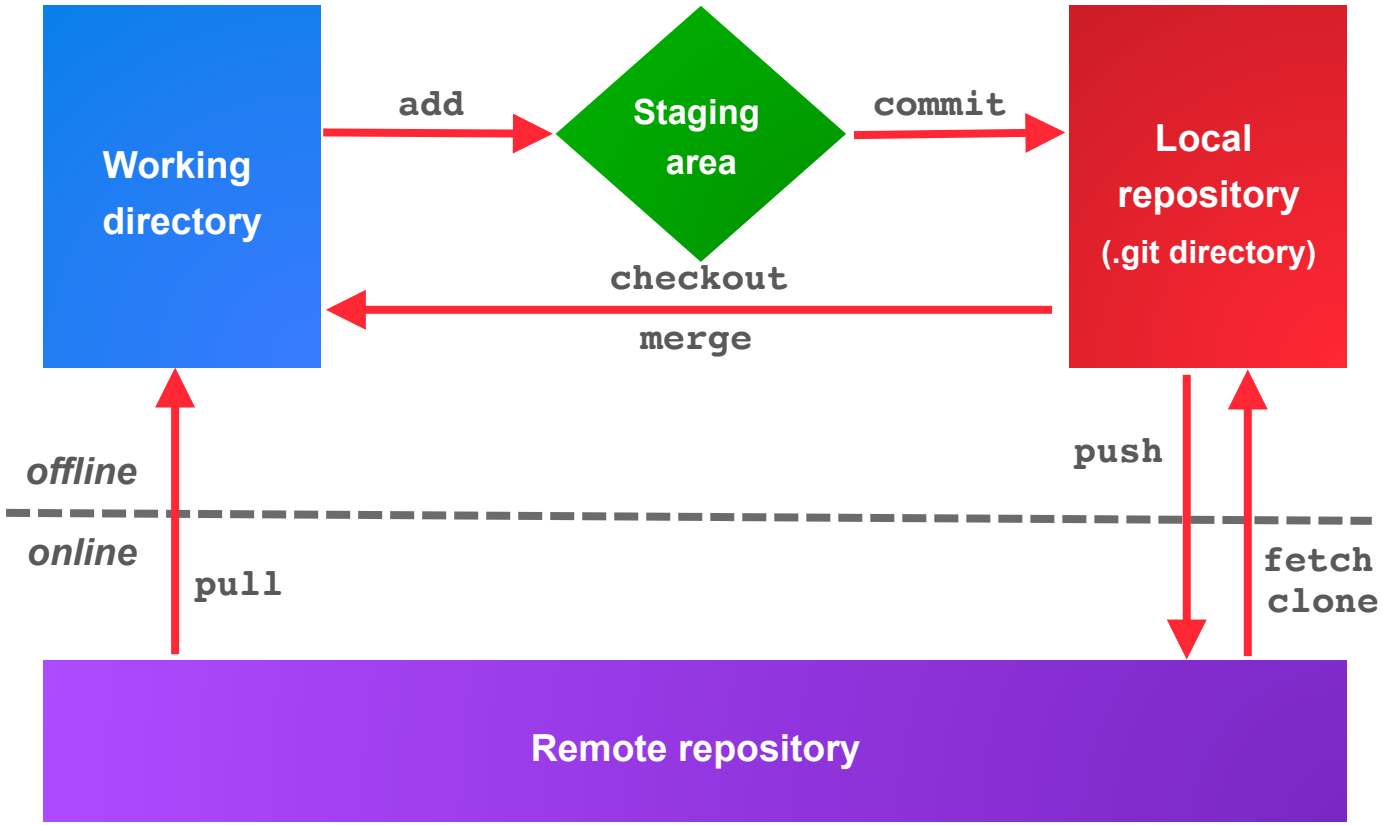
Remote repositories are versions of your project that are hosted on the Internet or network somewhere.

You can have several of them, each of which generally is either read-only or read/write for you.

Collaborating with others involves managing these remote repositories and pushing and pulling data to and from them when you need to share work.



Working with remote repositories

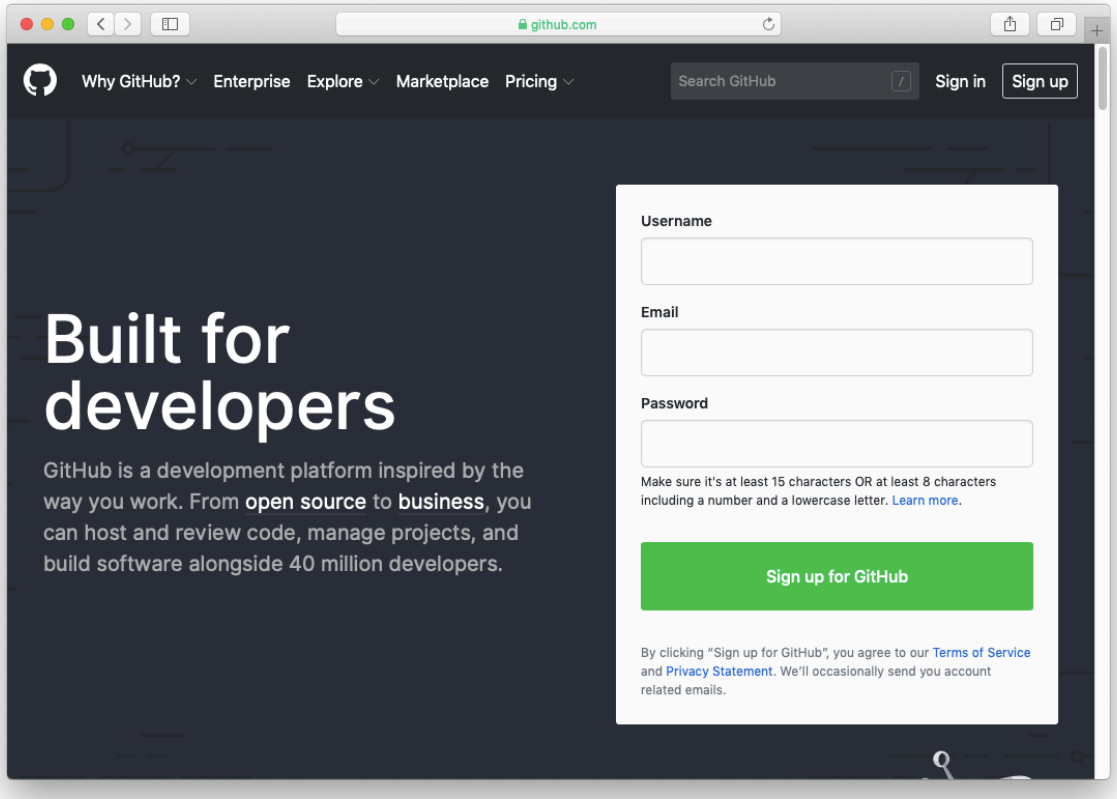


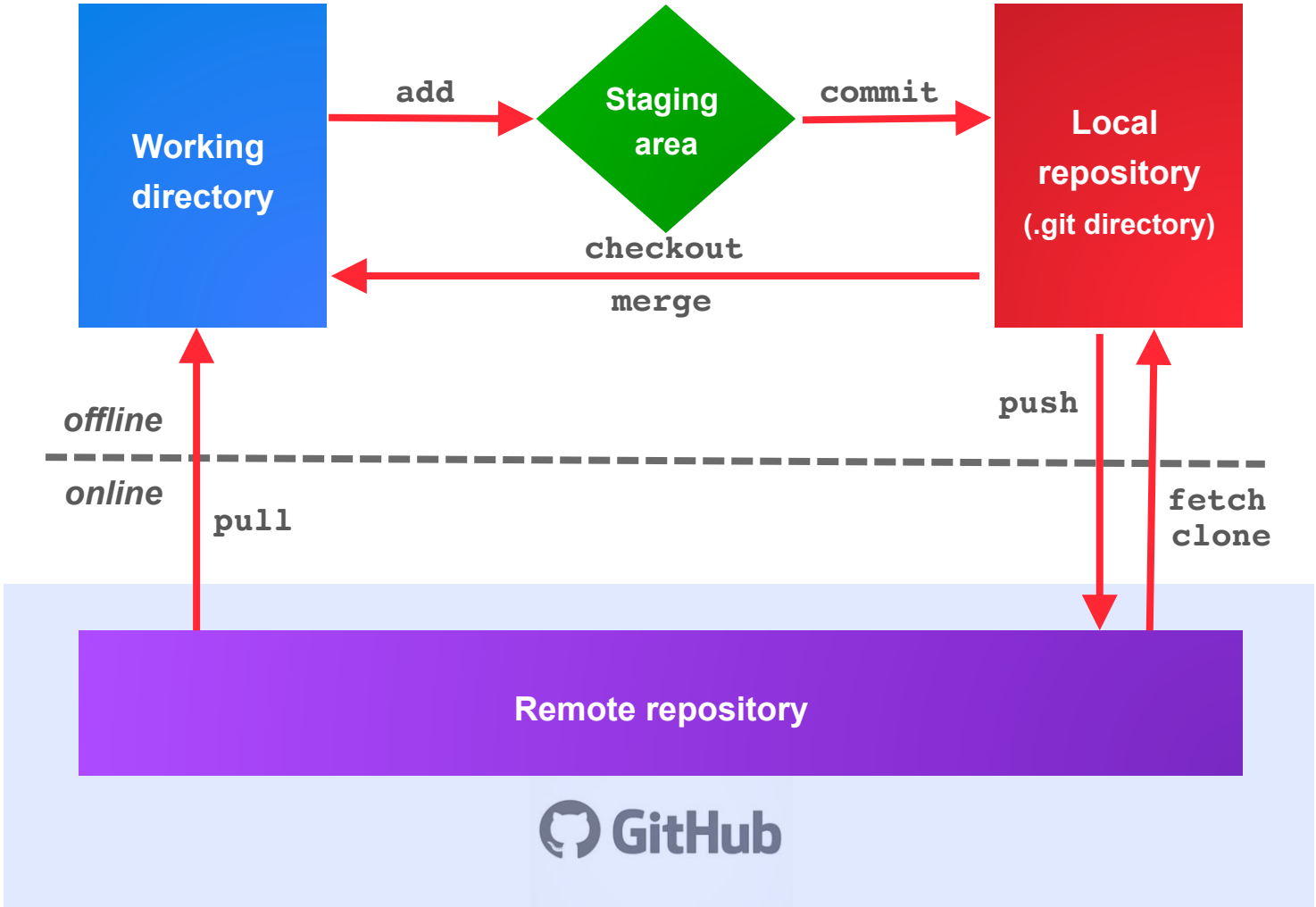
Using Github as a remote repository (and much more)

Using a remote repository could be done from many ways. Many solutions exist: GitHub, GitLab, Bitbucket among others.

In order to use GitHub, it suffices to sign-up on their website and create repositories. These repositories will act as cloud-based copies of the local repositories, or, rather, as a centralized single source of truth of the stable state of the codebase to which all modifications are pushed and then spread to others.

First connect to github.com and sign-in or sign-up.




Working directory
add
Staging area
commit
Local repository
(.git directory)
checkout
merge
offline
online
pull
push
fetch
clone
Remote repository
GitHub

Create a new GitHub project (owner of project)

1. Create a GitHub account and a new GitHub repository (make sure it is empty); then copy its URL and send that URL to your collaborator(s)

2. Add (register) the account of the collaborator(s) as collaborator(s) of the projects you have just created.

```
$ cd /some/parent/directory/
$ mkdir myrepo
$ cd myrepo
```

3. Initialize your Git local repository

```
$ git init
```

4. Create at least one file

```
$ nano OwnerFile1.html
```

5. Add at least one commit to be pushed

```
$ git add .
$ git commit -m "OwnerFile1.html added"
```

6. Push the changes of your local repository to the Web one (called origin in the previous command) with the following command:

```
$ git push -u origin master
```

Connect to a Github project (collaborator of the project)

1. Create a GitHub account and share that information with the project owber.

2. Get the repository URL of the HitHub project created by the Project owner (the one sent by the Project owner).

3. In your GitHub account, **create a new fork with the URL of the GitHub project (so, create your own copy of someone else's repository in your GitHub account)**

4. Clone this fork in your local repository with the following command:

```
$ git clone <fork URL of remote repo>
```

5. Create at least one file

```
$ nano file.html
```

6. Add at least one commit to be pushed

```
$ git add .
$ git commit -m "file.html added"
```

7. Add your repository's URL to Git with the following command. This will make Git interpret the origin keyword as a reference to the specified URL (copy it from the "Clone or download" button of the GitHub interface).

```
$ git remote add origin <your URL>
```

8. Push the changes of your local repository to the Web one (called origin in the previous command) with the following command:

```
$ git push -u origin master
```

Dealing with remote repository

Until now, that covers the procedure to be followed in order to make your commits appear in the remote repository; however, when one works as part of a team, it is also essential to have the commits of others available in the local repository.
If you are the owner of the project, you just execute the following command:

```
$ git fetch
```

After you run the fetch command, you would usually merge the changes into the local branch.

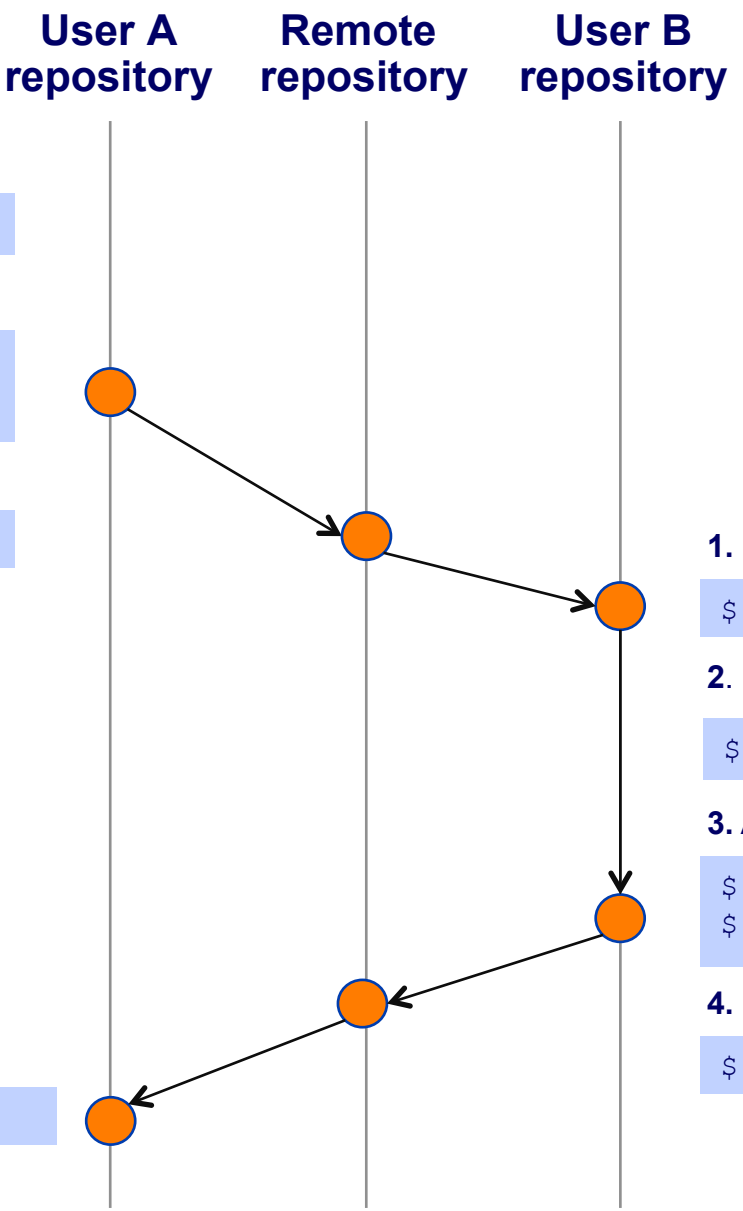
```
$ git merge origin/development
```

Evidently you must take care to avoid merge conflicts...

Now, in terms of the other common-use command, the git pull command acts in a very similar fashion to what has just been said.

However, one key difference separates it from fetch: when you run a pull command, it has the effect of doing a git fetch command first and then immediately running a merge command.
It is worth notice however, that, in general, pull commands can be used in place of manually merging any changes that were added to the remote repository.

```
$ git pull
```



1. Edit file1.html

```
$ nano file1.html
```

2. Add it and commit it

```
$ git add .
$ git commit -m "file added"
```

3. Push to remote repository

```
$ git push
```

1. Pull from remote repository

```
$ git pull
```

2. Edit file1.html

```
$ nano file1.html
```

3. Add it and commit it

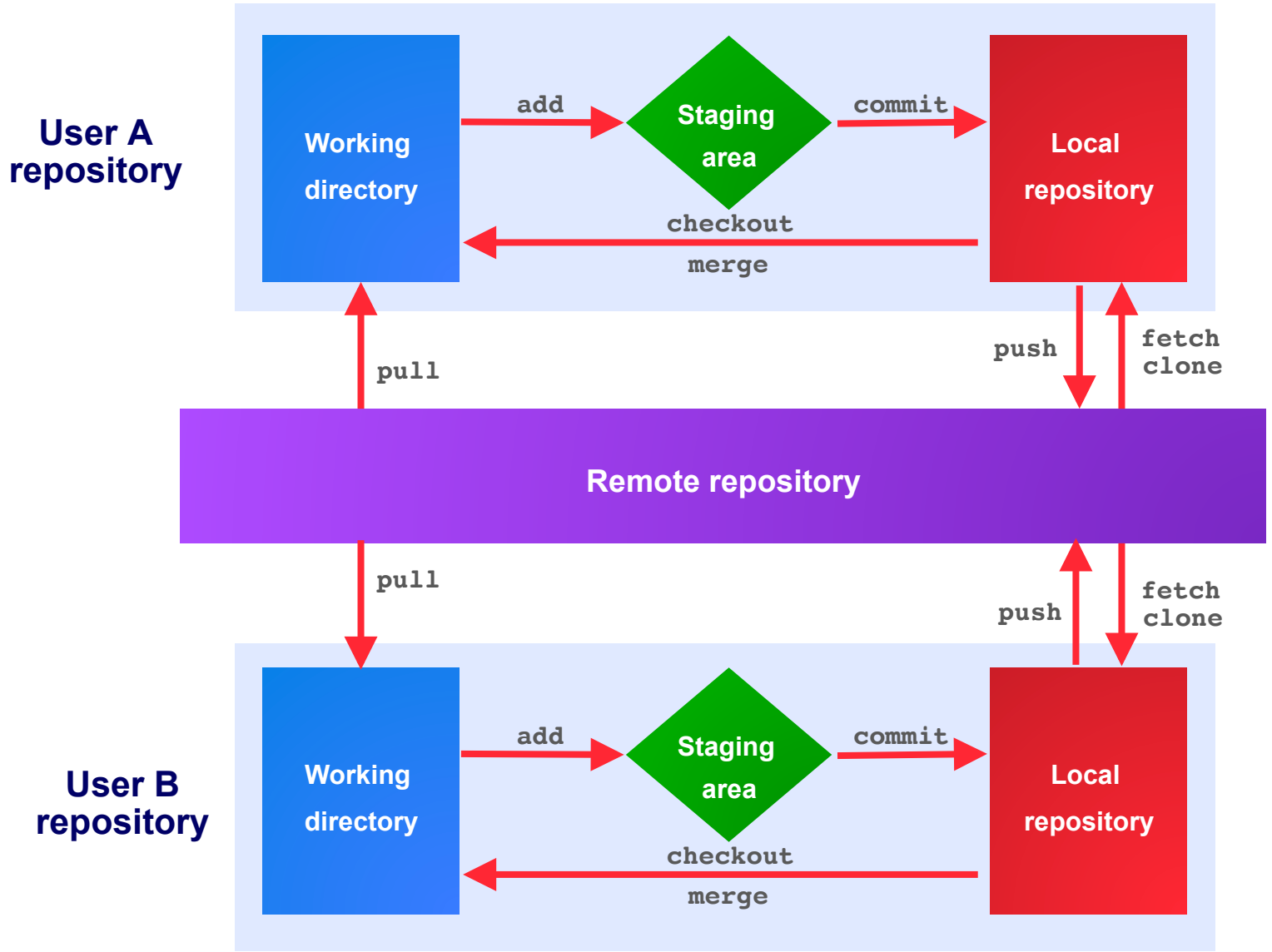
```
$ git add .
$ git commit -m "file modified"
```

4. Push to remote repository

```
$ git push
```

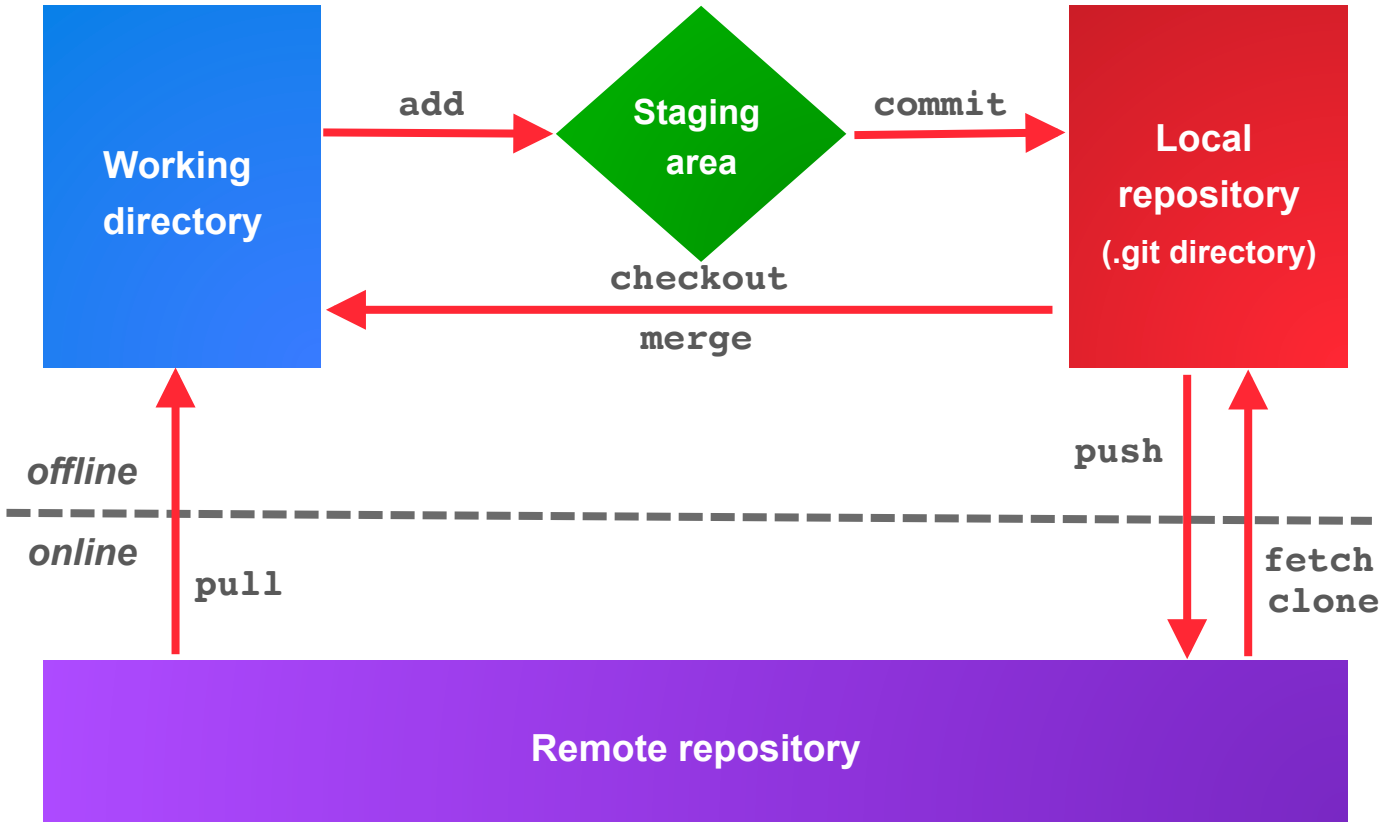
4. Pull from remote repository

```
$ git pull
```


User A repository
Working directory
add
Staging area
commit
Local repository
checkout
merge
pull
push
fetch clone
Remote repository
pull
push
fetch clone
User B repository
Working directory
add
Staging area
commit
Local repository
checkout
merge

Basic commands

- `git add`
- `git commit`
- `git branch`
- `git checkout`
- `git merge`
- `git clone`
- `git push`
- `git fetch`
- `git pull`



Advanced features

- Pull requests: you make local code changes and then submit those changes to a remote project maintainer for review before those changes are implemented, or merged (specific to GitHub)
- Forks: A **fork** is a copy of a repository. Forking a repository allows you to freely experiment with changes without affecting the original project. Most commonly, **forks** are used to either propose changes to someone else's project or to use someone else's project as a starting point for your own idea (specific to GitHub)
- .gitignore
- Working with SSH keys
- Many features to discover on GitHub.com (people management, wiki, issues, comments, graphs and many tools…)

Running an example of collaboration with GitHub

- Create the repository
  - Push access for all the members of the team
- Find the right strategy
  - Everybody work on the same branch (continuous integration)
  - (or) Everybody work in its branch (and merge should be performed)
  - (or) Everybody work in its branch and pull-requests are managed

Common commands and procedures (1)

Clone someone else's repository from GitHub

```
$ git clone https://github.com/
somegithubuser/somerepo.git (1)
$ cd somerepo
```

Command (1) creates a local copy of the repository set in the url. If it is your own repository you will be able to push any local commits to the cloud, otherwise you will not be able to, unless you've been given access by the owner of the repository.

Initialize a git repository in an existing folder

```
$ cd /path/to/my/codebase
$ git init (1)
$ git add . (2)
$ git commit -m "first commit" (3)
```

These commands will allow you to begin tracking your codebase with git, by initializing the .git folder (1), then adding all files and sub-folders to the repository (2), and finally committing the result so that the state of the repository is on par with the files (3).

Review the status of the repository (to check for changes that have not been added to a commit or untracked files)

```
$ cd /path/to/my/codebase
$ git status (1)
```

This command (1) will display the current status of the codebase, and will alert you to anything that differs from the latest snapshot and the actual files.

Common commands and procedures (2)

Configure git with your information

```
$ git config --global user.name "My
Name" (1)
$ git config --global user.email
"my@email.com" (2)
```

These commands (1) and (2) will setup your identifying information so that when you make a commit, this information (your name) will be visible to your collaborators and will allow you to identify yourself with your cloud base repository (your email). This only needs to be done once, as it is set globally.

Add a new file to an existing repository:

```
$ cd /path/to/my/codebase
$ git status (1)
$ touch myfile.java (2)
$ vim myfile.java (3)
$ git add myfile.java (4)
$ git commit -m "added myfile.java”(5)
```

When adding new files to a repository it is a good idea to first check the status of the repository (1). Afterwards, one would create the file and modify it at will (2) and (3) (this could be done in many different ways, git makes no distinction based on what editor or IDE one uses, or how the file came to be in the folder). Finally one would add the file so that git tracks it (4), and then one would commit these changes to update the current snapshot of the repository (5).

Review changes made to files

```
$ cd /path/to/my/codebase
$ vim myfile.txt (1)
$ vim mysecondfile.txt (1)
... (1)
$ git diff (2)
```

When one makes changes to one or more files (1) one can quickly see what additions or deletions have been made to those files by using the diff command (2).

Common commands and procedures (3)

Create a new branch and switch to it

```
$ cd /path/to/my/codebase
$ git branch mybranch (1)
$ git checkout mybranch (2)
```

These commands first create a new branch that begins at the current state of the repository (1) and then switches to that branch (2) so that any further commit will be applied to that branch.
This can also be done with only one command:

```
$ git checkout -b mybranch
```

Check current and available branches

```
$ cd /path/to/my/codebase
$ git branch (1)
```

This command (1) will list all available branches and will show an asterisk (*) beside the current branch.

Remove a branch
(assuming you are on branch mybranch)

```
$ cd /path/to/my/codebase
$ git branch (1)
$ git checkout master (2)
$ git branch -D mybranch (3)
```

These commands remove a branch locally. In order to do so, one should (though not must) first check the current branches (1), switch to another branch (2) and finally remove the branch (3).

Merge a branch to another
(assuming you are on branch mybranch)

```
$ cd /path/to/my/codebase
$ git checkout master (1)
$ git merge mybranch (2)
```

With these commands one would first switch over to the branch one wishes to bring up to date (1) and then one would merge the changes into the branch (2).

Common commands and procedures (4)

Fetch the latest changes from a remote repository

```
$ cd /path/to/my/codebase
$ git fetch (1)
```

This command (1) retrieves the latest information from the repository (new branches, new commits, etc.). Should one desire to merge any changes that were made to the current branch immediately one would run:

```
$ git pull (2)
```

This command (2) is equivalent to running git fetch and then merging the changes into the repository. It is worth mentioning that one may merge the changes from the remote repository as follows:

```
$ git merge origin/somebranch (3)
```

This command (3) will merge the current commit from the remote repository's somebranch branch into the current branch.

Push your committed changes to your online repository

```
$ cd /path/to/my/codebase
$ git push origin master (1)
```

This command (1) allows you to push any commits that do not exist in the remote repository to it.

Add a new remote repository

```
$ cd /path/to/my/codebase
$ git remote add upstream https://
github.com/adev/upstream.git (1)
```

With this command (1) one may add a new repository from which one may fetch changes. This is particularly useful if one is working on one's own copy of a repository.

Common commands and procedures (5)

Remove a file from the repository and the filesystem

```
$ git rm myfile.html (1)
```

This command will erase the file from your repository and will make it so git stops tracking it as well. If you wish to remove an entire folder add the -r flag as follows:

```
$ git rm -r folder
```

Remove a file from git only (and not the filesystem)

```
$ git rm --cached myfile.html (1)
```

This command will remove the file from git and it will no longer be tracked, but it will remain in your filesystem. This might be useful if you previously tracked a file but now wish to ignore it (for instance you might want to add it to .gitignore or move it to another project).

Remove a local branch

```
$ git branch -d mybranch (1)
```

This command (1) removes the local branch named mybranch but only if it is up to date and does not have pending changes. You must also be on another branch when you delete it. If, however, you would like to remove the branch irrespective of its status, use the following command:

```
$ git branch -D mybranch (2)
```

Remove a remote branch

```
$ git push origin :mybranch (1)
```

This command (1) will delete the branch after the colon in the origin remote repository.

Basic commands

```
git add
git commit
git branch
git checkout
git merge
git clone
git push
git fetch
git pull
```