

git: A Version Control System

(Anybody can make history, but only few can (re)write it)

Emmanuel Fleury

<emmanuel.fleury@u-bordeaux.fr>

Université de Bordeaux, France

January 21, 2025



1 Introduction

2 Basic Usages

- Command Syntax
- Configuration
- Handling the Index
- Dealing with History
- Using Branches
- Getting a Repository
- Synchronize with Remote
- Solving Conflicts
- Managing Remotes & Other Useful Commands

3 Development Workflows

- Centralized Workflow
- Feature Branch Workflow
- Forking Workflow
- Massively Distributed Workflow
- Best Practices

1 Introduction

2 Basic Usages

- Command Syntax
- Configuration
- Handling the Index
- Dealing with History
- Using Branches
- Getting a Repository
- Synchronize with Remote
- Solving Conflicts
- Managing Remotes & Other Useful Commands

3 Development Workflows

- Centralized Workflow
- Feature Branch Workflow
- Forking Workflow
- Massively Distributed Workflow
- Best Practices

A **Version Control Software** is a tool that:

- Keeps a **log of changes** applied on source code.
- Helps to **merge your work** with other developers.
- Allows to **navigate within history** of source code versions.

Even when coding alone, a VCS is extremely useful to not be annoyed by tracking different revisions of your software.

A few (famous) VCS:

- **sccs** (1972)
- **cvs** (1990)
- **darcs** (2003)
- **hg** (2005)
- **rcs** (1982)
- **svn** (2000)
- **git** (2005)
- ...

Get used to VCS tools on long term projects !!!

The **sequence of modifications** bringing you from the start of the project til now.

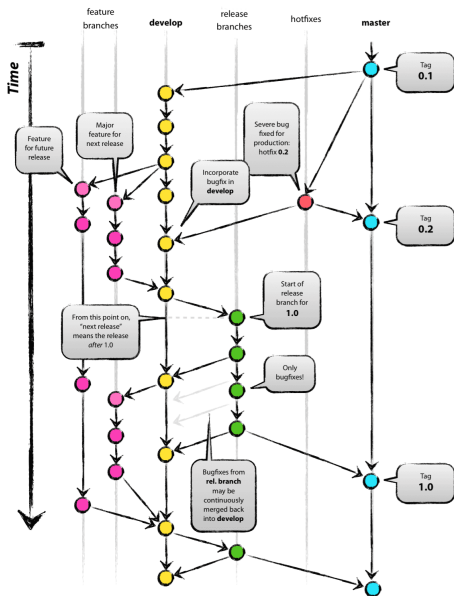
Two distinct **type of approaches**:

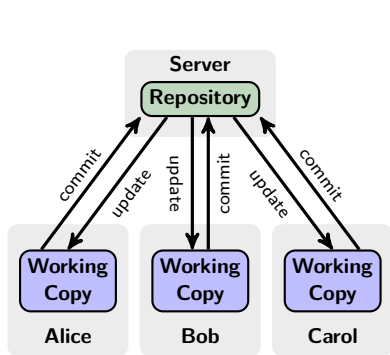
- **History cannot be rewritten** (cvs, svn, hg, ...)

History represents the chronological order of events, we cannot erase it. Thus, it keeps track of all your mistakes.

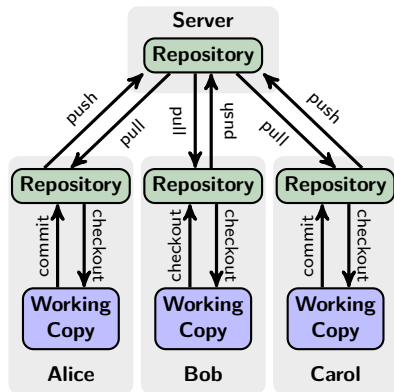
- **History is a set of lies agreed upon** (git, darcs, ...)

History tries to show the best logical path from one point to another, we can modify it to make it more understandable.





Centralized Version Control



Distributed Version Control

Repository

Hold the whole history of the project (as a data-base of code).

Working Copy

A snapshot of the code taken from the repository on which you can work.

1 Introduction

2 Basic Usages

- Command Syntax
- Configuration
- Handling the Index
- Dealing with History
- Using Branches
- Getting a Repository
- Synchronize with Remote
- Solving Conflicts
- Managing Remotes & Other Useful Commands

3 Development Workflows

- Centralized Workflow
- Feature Branch Workflow
- Forking Workflow
- Massively Distributed Workflow
- Best Practices

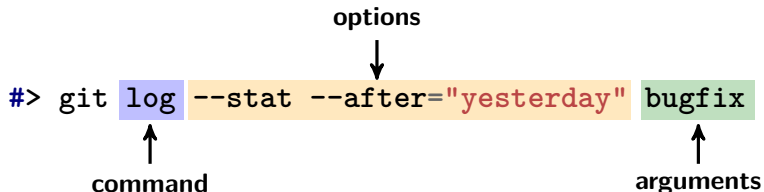
1 Introduction

2 Basic Usages

- Command Syntax
- Configuration
- Handling the Index
- Dealing with History
- Using Branches
- Getting a Repository
- Synchronize with Remote
- Solving Conflicts
- Managing Remotes & Other Useful Commands

3 Development Workflows

- Centralized Workflow
- Feature Branch Workflow
- Forking Workflow
- Massively Distributed Workflow
- Best Practices



Git main commands

```
add blame branch checkout clone commit config
diff init log merge pull push rebase reset
revert rm stash status switch tag ...
```

How to Get Help

```
#> git help <command>
#> git <command> --help
#> man git-<command>
```

Help Examples

```
#> git help log
#> git log --help
#> man git-log
```

1 Introduction

2 Basic Usages

- Command Syntax
- Configuration
- Handling the Index
- Dealing with History
- Using Branches
- Getting a Repository
- Synchronize with Remote
- Solving Conflicts
- Managing Remotes & Other Useful Commands

3 Development Workflows

- Centralized Workflow
- Feature Branch Workflow
- Forking Workflow
- Massively Distributed Workflow
- Best Practices

First of all, you need to **set your identity** in order to **tag properly your commits** and get **your usual tools** (e.g. editor).

You have two options ('global' or 'local').

Global Configuration ('~/.gitconfig' or '~/.config/git/config')

```
#> git config --global user.name "John Doe"  
#> git config --global user.email john.doe@student.edu  
#> git config --global core.editor emacs
```

Local Configuration ('project/.git/config')

```
#> git config --local user.name "John Doe"  
#> git config --local user.email john.doe@email.net  
#> git config --local core.editor emacs
```

Where: Local > Global

Setting Config

```
#> git config section.key1 'value1'
#> git config section.key2 'value2'
#> git config section.subsection.key3 'value3'
#> git config --list
section.key1=value1
section.key2=value2
section.subsection.key3=value3
```

Config File Format

```
# This is a comment
[section]
  key1 = value1 # Another comment
  key2 = value2
[section "subsection"]
  key3 = value3
```

Sections and keys are case-sensitive!

[user]

```
name = John Doe
email = john.doe@student.edu
```

[init]

```
# Set name of default branch
defaultBranch = main
```

[pull]

```
# Rebase unpushed work in case of 'pull'
rebase = true
```

[push]

```
default = simple
```

[core]

```
# Default editor to write commit message
editor = emacs -nw
```

[color]

```
ui = auto
```

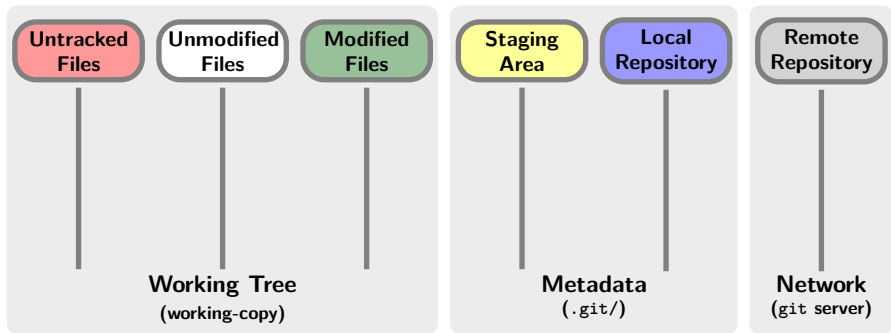
1 Introduction

2 Basic Usages

- Command Syntax
- Configuration
- Handling the Index
- Dealing with History
- Using Branches
- Getting a Repository
- Synchronize with Remote
- Solving Conflicts
- Managing Remotes & Other Useful Commands

3 Development Workflows

- Centralized Workflow
- Feature Branch Workflow
- Forking Workflow
- Massively Distributed Workflow
- Best Practices



Working Tree

- **Untracked Files:** Files that are not tracked by git in the working copy.
- **Unmodified Files:** Tracked files that are not modified.
- **Modified Files:** Tracked files that are modified and ready to get staged.

Metadata

- **Staging Area** or **Index** (`.git/index`): Stores all the temporary modifications planned for commit.
- **Local Repository** (`.git/objects/`): Hold the whole history of the project with all the modifications.

Network

- **Remote Repository:** A place used to synchronize your repository with other developers.

Gives the **current status** of all your files which are in the **untracked files**, the **working tree** or the **staging area**.

Untracked Files

Untracked files:

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

README

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

Modified Files

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: README

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

Unmodified Files

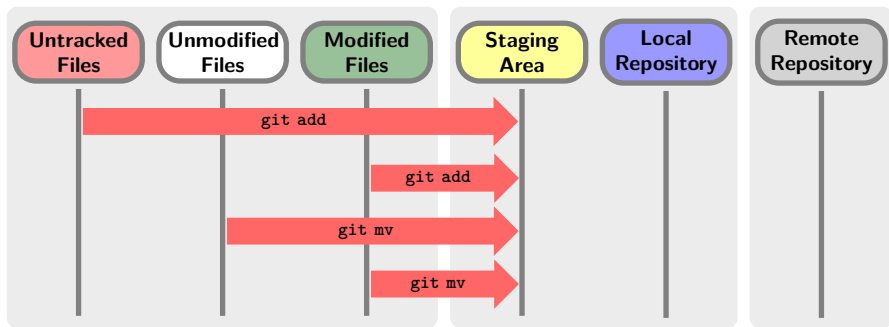
nothing to commit, working tree clean

Staged Files

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file: README



Command 'git add FILE'

Used to add an **untracked file** or a **modification on a file** to the **staging area**. Note that moving an untrack file to staging is not the same than moving it from modified files.

Command 'git mv FLIE FILE'

Used to rename or move a **tracked file** in the **staging area**.

Example: 'add' and 'mv'

```
#> echo "Hello World!" > REAMDE
```

```
#> git add REAMDE
```

```
#> git status
```

```
No commits yet
```

```
Changes to be committed:
```

```
(use "git rm --cached <file>..." to unstage)
```

```
new file:   REAMDE
```

```
#> git mv REAMDE README
```

```
#> git status
```

```
No commits yet
```

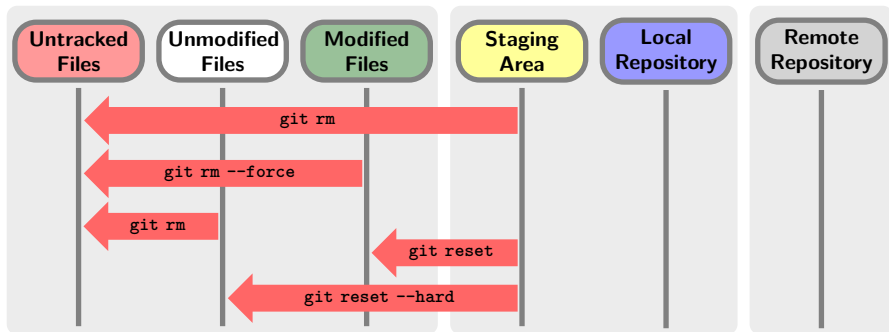
```
Changes to be committed:
```

```
(use "git rm --cached <file>..." to unstage)
```

```
new file:   README
```

Example: 'add --patch'

```
#> echo "Hello World!" > README
#> git add README
#> echo "Second line" >> README
#> git add --patch README
diff --git a/README b/README
index 980a0d5..307dca1 100644
--- a/README
+++ b/README
@@ -1,2 @@
 Hello World!
+Second line
Stage this hunk [y,n,q,a,d,e,?]? ?
y - stage this hunk
n - do not stage this hunk
q - quit; do not stage this hunk or any of the remaining ones
a - stage this hunk and all later hunks in the file
d - do not stage this hunk or any of the later hunks in the file
e - manually edit the current hunk
? - print help
@@ -1,2 @@
 Hello World!
+Second line
Stage this hunk [y,n,q,a,d,e,?]? y
```



Command 'git rm FILE'

Used to **remove a file from the tracked files** (unmodified, modified or staged).

- `'git rm -r DIR/'`: Recursive removal of the content of a directory.
- `'git rm --cached FILE'`: Do not delete the file which is removed from the **tracked files**;

Command 'git reset FILE'

Used to **remove changes from the staging area** to the working tree.

- `'git reset --hard'`: Throw away all the pending changes (you can't get it back!).

Example: 'rm'

```
#> echo "Hello World!" > README
#> git add README
#> git status
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

       new file:   README

#> git rm README
error: the following file has changes staged in the index:
  README
(use --cached to keep the file, or -f to force removal)
#> git rm --cached README
rm 'README'
#> git status
Untracked files:
  (use "git add <file>..." to include in what will be committed)

       README

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

Example: 'reset'

```
#> echo "Hello World!" > README
#> git add README
#> git status
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

       new file:   README

#> git reset
#> git status
No commits yet

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

       README

nothing added to commit but untracked files present (use "git add" to track)
#> git add README
#> git reset --hard
#> git status
On branch main

No commits yet

nothing to commit (create/copy files and use "git add" to track)
```

Why Ignoring Files?

- You do not want to see these files in 'git status' to have a better understanding of the modifications you did on the code.
- Building the project and editing files produce a lot of intermediate files that are not relevant to be cared upon. These files must be banned from the history because:
 - It blurs the 'status' command a lot;
 - Versioning binary files is inefficient;
 - It breaks Makefile timestamps on files;
 - It breaks merge algorithm;
 - It eats disk space for nothing.

How to Ignore Files?

- Ignore files: '~/.config/git/ignore' (global) or 'project/.gitignore' (local)
- Basic Ignore Format:

```
# This is a comment
# Ignore all files ended by '~'
*~
# Ignore all file named 'core'
core
# Ignore the directory 'images/' and its content
images/
```

Bad!

```
#> git status
On branch main
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:   src/project.c

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

        project
        src/module.o
        src/project
        src/project.o

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

Good!

```
#> git status
On branch main
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:   src/project.c

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

.gitignore

```
*~
*.o
project
src/project
```


1 Introduction

2 Basic Usages

- Command Syntax
- Configuration
- Handling the Index
- **Dealing with History**
- Using Branches
- Getting a Repository
- Synchronize with Remote
- Solving Conflicts
- Managing Remotes & Other Useful Commands

3 Development Workflows

- Centralized Workflow
- Feature Branch Workflow
- Forking Workflow
- Massively Distributed Workflow
- Best Practices

```
#> git log
commit 848679e3dc10b6ef13e1d2ea2a5055 (HEAD->main)
Author: John Doe <john.doe@student.edu>
Date: Mon Nov 19 12:02:05 2018 +0100
```

Cleaning and fixing bugs

- Removed 'project' exec from tracked files
- Updated the 'clean' target in Makefile
- Fixing a missing include in 'project.c'

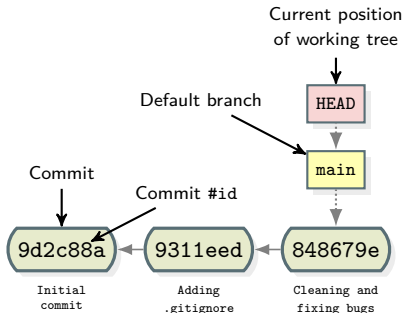
```
commit 9311eedbc1c00071055abfbd4228bd
Author: John Doe <john.doe@student.edu>
Date: Mon Nov 19 12:00:13 2018 +0100
```

Adding a .gitignore file

```
commit 9d2c88af92b23131b74bbe6fdcd3f8
Author: John Doe <john.doe@student.edu>
Date: Mon Nov 19 11:59:19 2018 +0100
```

Initial commit

```
#> git log --oneline
848679e (HEAD -> main) Cleaning and fixing bugs
9311eed Adding a .gitignore file
9d2c88a Initial commit
```




HEAD: Convention to name current position of working tree in history and where:

- **HEAD~1** (HEAD~): First ancestor of HEAD.
- **HEAD~2** (HEAD~~): Second ancestor of HEAD.
- **HEAD~n**: *n*-th ancestor of HEAD.

9d2c88a: Commit #id is obtained by hashing the tree of changes as a Merkle tree with SHA-1 (160-bit).

```
#> git log --oneline --stat
9311eed (HEAD -> main) Adding a .gitignore file
.gitignore | 3 +++
1 file changed, 3 insertions(+)
9d2c88a Initial commit
Makefile | 13 ++++++++
project | Bin 0 -> 1708464 bytes
project.c | 21 ++++++++
project.h | 8 ++++++
4 files changed, 42 insertions(+)
```

Displays the statistics
on changed lines in files




```
#> git log --patch
commit 9311eedbc1c031fe09dbfd0071055abfbd4228bd (HEAD -> main)
Author: John Doe <john.doe@student.edu>
Date: Mon Nov 19 12:00:13 2018 +0100

    Adding a .gitignore file

diff --git a/.gitignore b/.gitignore
new file mode 100644
index 0000000..b069126
--- /dev/null
+++ b/.gitignore
@@ -0,0 +1,3 @@
+*~
+*.o
+project
```

Displays all the diffs
of the commit



Summarize changes in around 50 characters or less

More detailed explanatory text, if necessary. Wrap it to about 72 characters or so. In some contexts, the first line is treated as the subject of the commit and the rest of the text as the body. The blank line separating the summary from the body is critical (unless you omit the body entirely); various commands like ``log``, ``shortlog`` and ``rebase`` can get confused if you run the two together.

Explain the problem that this commit is solving. Focus on why you are making this change as opposed to how (the code explains that). Are there side effects or other unintuitive consequences of this change? Here's the place to explain them.

Further paragraphs come after blank lines.

- Bullet points are okay, too
- Typically a hyphen or asterisk is used for the bullet, preceded by a single space, with blank lines in between.

If you use an issue tracker, put references at the bottom, like this:

```
Resolves: #123  
See also: #456, #789
```

For more, see: **How to Write a Git Commit Message**, by *Chris Beams*, 2014.
<https://chris.beams.io/posts/git-commit/>

Summarize changes in around 50 characters or less

← Explicit Title

More detailed explanatory text, if necessary. Wrap it to about 72 characters or so. In some contexts, the first line is treated as the subject of the commit and the rest of the text as the body. The blank line separating the summary from the body is critical (unless you omit the body entirely); various commands like ``log``, ``shortlog`` and ``rebase`` can get confused if you run the two together.

Explain the problem that this commit is solving. Focus on why you are making this change as opposed to how (the code explains that). Are there side effects or other unintuitive consequences of this change? Here's the place to explain them.

Further paragraphs come after blank lines.

- Bullet points are okay, too
- Typically a hyphen or asterisk is used for the bullet, preceded by a single space, with blank lines in between.

If you use an issue tracker, put references at the bottom, like this:

```
Resolves: #123  
See also: #456, #789
```

For more, see: **How to Write a Git Commit Message**, by *Chris Beams*, 2014.
<https://chris.beams.io/posts/git-commit/>

Summarize changes in around 50 characters or less

← Explicit Title
← Empty line

More detailed explanatory text, if necessary. Wrap it to about 72 characters or so. In some contexts, the first line is treated as the subject of the commit and the rest of the text as the body. The blank line separating the summary from the body is critical (unless you omit the body entirely); various commands like ``log``, ``shortlog`` and ``rebase`` can get confused if you run the two together.

Explain the problem that this commit is solving. Focus on why you are making this change as opposed to how (the code explains that). Are there side effects or other unintuitive consequences of this change? Here's the place to explain them.

Further paragraphs come after blank lines.

- Bullet points are okay, too
- Typically a hyphen or asterisk is used for the bullet, preceded by a single space, with blank lines in between.

If you use an issue tracker, put references at the bottom, like this:

```
Resolves: #123  
See also: #456, #789
```

For more, see: **How to Write a Git Commit Message**, by *Chris Beams*, 2014.
<https://chris.beams.io/posts/git-commit/>

Summarize changes in around 50 characters or less

← Explicit Title
← Empty line

More detailed explanatory text, if necessary. Wrap it to about 72 characters or so. In some contexts, the first line is treated as the subject of the commit and the rest of the text as the body. The blank line separating the summary from the body is critical (unless you omit the body entirely); various commands like ``log``, ``shortlog`` and ``rebase`` can get confused if you run the two together.

Explain the problem that this commit is solving. Focus on why you are making this change as opposed to how (the code explains that). Are there side effects or other unintuitive consequences of this change? Here's the place to explain them.

← Optional body

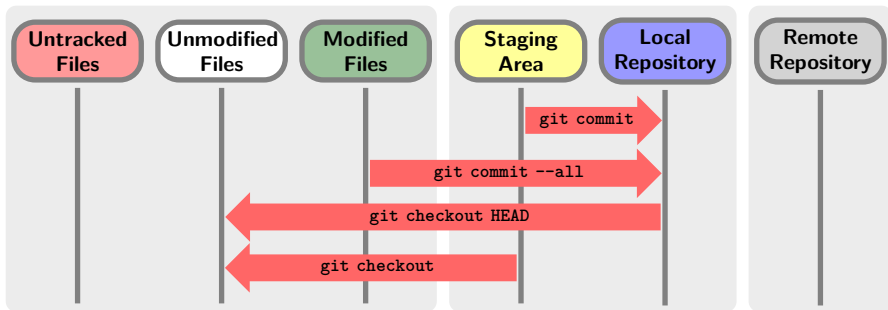
Further paragraphs come after blank lines.

- Bullet points are okay, too
- Typically a hyphen or asterisk is used for the bullet, preceded by a single space, with blank lines in between.

If you use an issue tracker, put references at the bottom, like this:

Resolves: #123
See also: #456, #789

For more, see: **How to Write a Git Commit Message**, by *Chris Beams*, 2014.
<https://chris.beams.io/posts/git-commit/>



Command 'git commit'

Store the content of the staging area in local repository.

- `'git commit --message="My message"':` Send the commit with the given message.
- `'git commit --amend':` Merge the current commit with the last one.
(beware, use this only if you haven't pushed to remote!)

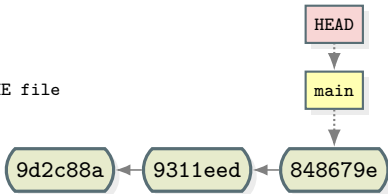
Command 'git checkout'

Change current working tree to another state in history or reset staging area.

- `'git checkout HEAD~n':` Change the working tree to `n` commits in the past.

Example: 'commit' and 'checkout'

```
#> echo "Hello World!" > README
#> git add README
#> git commit -m "Adding a README file"
[main 52ee35a] Adding a README file
 1 file changed, 1 insertion(+)
 create mode 100644 README
#> git log --oneline
52ee35a (HEAD -> main) Adding a README file
848679e Cleaning and fixing bugs
9311eed Adding a .gitignore file
9d2c88a Initial commit
#> git checkout HEAD~2
Note: checking out 'HEAD~2'.
```



You are in 'detached HEAD' state. You can look around, make experimental changes and commit them, and you can discard any commits you make in this state without impacting any branches by performing another checkout.

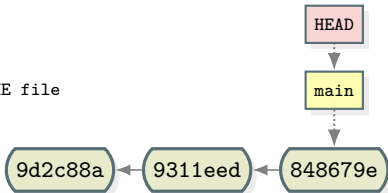
If you want to create a new branch to retain commits you create, you may do so (now or later) by using `-b` with the checkout command again. Example:

```
git checkout -b <new-branch-name>
```

```
HEAD is now at 9311eed Adding a .gitignore file
#> git log --oneline
9311eed (HEAD) Adding a .gitignore file
9d2c88a Initial commit
```

Example: 'commit' and 'checkout'

```
#> echo "Hello World!" > README
#> git add README
#> git commit -m "Adding a README file"
[main 52ee35a] Adding a README file
 1 file changed, 1 insertion(+)
 create mode 100644 README
#> git log --oneline
52ee35a (HEAD -> main) Adding a README file
848679e Cleaning and fixing bugs
9311eed Adding a .gitignore file
9d2c88a Initial commit
#> git checkout HEAD~2
Note: checking out 'HEAD~2'.
```



You are in 'detached HEAD' state. You can look around, make experimental changes and commit them, and you can discard any commits you make in this state without impacting any branches by performing another checkout.

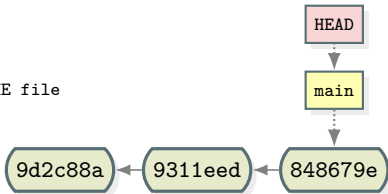
If you want to create a new branch to retain commits you create, you may do so (now or later) by using `-b` with the checkout command again. Example:

```
git checkout -b <new-branch-name>
```

```
HEAD is now at 9311eed Adding a .gitignore file
#> git log --oneline
9311eed (HEAD) Adding a .gitignore file
9d2c88a Initial commit
```

Example: 'commit' and 'checkout'

```
#> echo "Hello World!" > README
#> git add README
#> git commit -m "Adding a README file"
[main 52ee35a] Adding a README file
 1 file changed, 1 insertion(+)
 create mode 100644 README
#> git log --oneline
52ee35a (HEAD -> main) Adding a README file
848679e Cleaning and fixing bugs
9311eed Adding a .gitignore file
9d2c88a Initial commit
#> git checkout HEAD~2
Note: checking out 'HEAD~2'.
```



You are in 'detached HEAD' state. You can look around, make experimental changes and commit them, and you can discard any commits you make in this state without impacting any branches by performing another checkout.

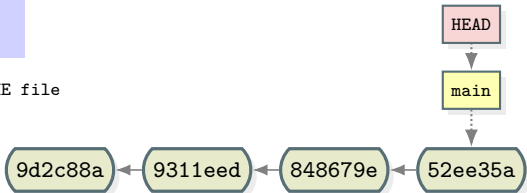
If you want to create a new branch to retain commits you create, you may do so (now or later) by using `-b` with the checkout command again. Example:

```
git checkout -b <new-branch-name>
```

```
HEAD is now at 9311eed Adding a .gitignore file
#> git log --oneline
9311eed (HEAD) Adding a .gitignore file
9d2c88a Initial commit
```

Example: 'commit' and 'checkout'

```
#> echo "Hello World!" > README
#> git add README
#> git commit -m "Adding a README file"
[main 52ee35a] Adding a README file
1 file changed, 1 insertion(+)
create mode 100644 README
#> git log --oneline
52ee35a (HEAD -> main) Adding a README file
848679e Cleaning and fixing bugs
9311eed Adding a .gitignore file
9d2c88a Initial commit
#> git checkout HEAD~2
Note: checking out 'HEAD~2'.
```



You are in 'detached HEAD' state. You can look around, make experimental changes and commit them, and you can discard any commits you make in this state without impacting any branches by performing another checkout.

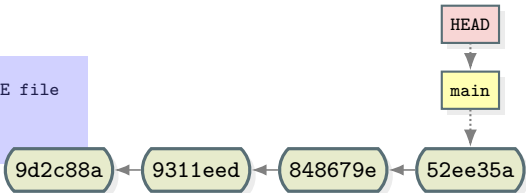
If you want to create a new branch to retain commits you create, you may do so (now or later) by using `-b` with the checkout command again. Example:

```
git checkout -b <new-branch-name>
```

```
HEAD is now at 9311eed Adding a .gitignore file
#> git log --oneline
9311eed (HEAD) Adding a .gitignore file
9d2c88a Initial commit
```

Example: 'commit' and 'checkout'

```
#> echo "Hello World!" > README
#> git add README
#> git commit -m "Adding a README file"
[main 52ee35a] Adding a README file
 1 file changed, 1 insertion(+)
 create mode 100644 README
#> git log --oneline
52ee35a (HEAD -> main) Adding a README file
848679e Cleaning and fixing bugs
9311eed Adding a .gitignore file
9d2c88a Initial commit
#> git checkout HEAD~2
Note: checking out 'HEAD~2'.
```



You are in 'detached HEAD' state. You can look around, make experimental changes and commit them, and you can discard any commits you make in this state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may do so (now or later) by using `-b` with the checkout command again. Example:

```
git checkout -b <new-branch-name>
```

```
HEAD is now at 9311eed Adding a .gitignore file
#> git log --oneline
9311eed (HEAD) Adding a .gitignore file
9d2c88a Initial commit
```

Example: 'commit' and 'checkout'

```
#> echo "Hello World!" > README
#> git add README
#> git commit -m "Adding a README file"
[main 52ee35a] Adding a README file
 1 file changed, 1 insertion(+)
 create mode 100644 README
#> git log --oneline
52ee35a (HEAD -> main) Adding a README file
848679e Cleaning and fixing bugs
9311eed Adding a .gitignore file
9d2c88a Initial commit
```

```
#> git checkout HEAD~2
Note: checking out 'HEAD~2'.
```

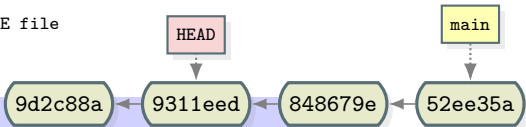
You are in 'detached HEAD' state. You can look around, make experimental changes and commit them, and you can discard any commits you make in this state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may do so (now or later) by using `-b` with the checkout command again. Example:

```
git checkout -b <new-branch-name>
```

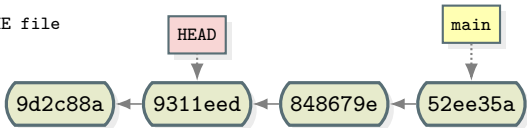
```
HEAD is now at 9311eed Adding a .gitignore file
```

```
#> git log --oneline
9311eed (HEAD) Adding a .gitignore file
9d2c88a Initial commit
```



Example: 'commit' and 'checkout'

```
#> echo "Hello World!" > README
#> git add README
#> git commit -m "Adding a README file"
[main 52ee35a] Adding a README file
 1 file changed, 1 insertion(+)
 create mode 100644 README
#> git log --oneline
52ee35a (HEAD -> main) Adding a README file
848679e Cleaning and fixing bugs
9311eed Adding a .gitignore file
9d2c88a Initial commit
#> git checkout HEAD~2
Note: checking out 'HEAD~2'.
```



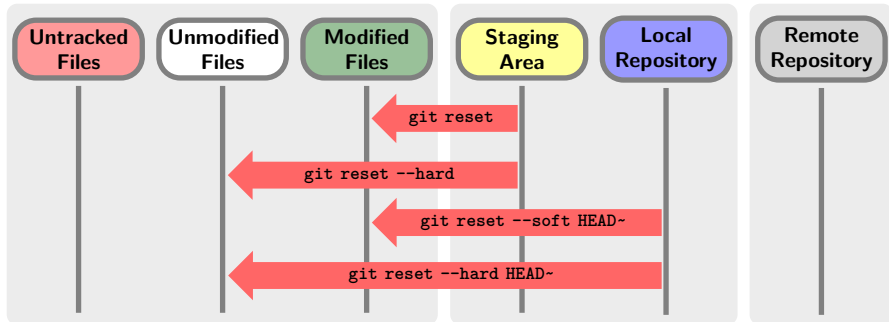
You are in 'detached HEAD' state. You can look around, make experimental changes and commit them, and you can discard any commits you make in this state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may do so (now or later) by using `-b` with the checkout command again. Example:

```
git checkout -b <new-branch-name>
```

HEAD is now at 9311eed Adding a .gitignore file

```
#> git log --oneline
9311eed (HEAD) Adding a .gitignore file
9d2c88a Initial commit
```

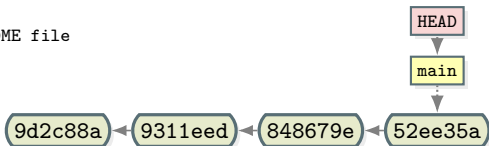


'`git reset`': Move HEAD to a specified state in the history.

- '`git reset --mixed`' (default): Resets staging area but not working tree. So, it keeps all the changes and they appear in the working tree as they were before '`git add`'.
- '`git reset --soft`': Leave staging area and working tree untouched but move HEAD to the given position. This option ensures that you keep all the changes you did since then, they appear in the staging area (but uncommitted).
- '`git reset --hard`': Reset staging area and working tree to the new state. It throws away all the pending changes (you can't get it back!) and return to a clean working tree.

Example: 'checkout' vs. 'reset' (1/2)

```
#> git log --oneline
52ee35a ((HEAD -> main) Adding a README file
848679e Cleaning and fixing bugs
9311eed Adding a .gitignore file
9d2c88a Initial commit
#> git checkout HEAD~
Note: checking out 'HEAD~'.
```



You are in 'detached HEAD' state. You can look around, make experimental changes and commit them, and you can discard any commits you make in this state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may do so (now or later) by using `-b` with the checkout command again. Example:

```
git checkout -b <new-branch-name>
```

HEAD is now at 848679e Cleaning and fixing bugs

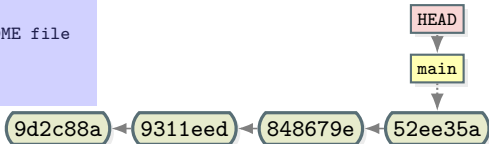
```
#> git switch main
```

Previous HEAD position was 848679e Cleaning and fixing bugs

Switched to branch 'main'

Example: 'checkout' vs. 'reset' (1/2)

```
#> git log --oneline
52ee35a ((HEAD -> main) Adding a README file
848679e Cleaning and fixing bugs
9311eed Adding a .gitignore file
9d2c88a Initial commit
#> git checkout HEAD~
Note: checking out 'HEAD~'.
```



You are in 'detached HEAD' state. You can look around, make experimental changes and commit them, and you can discard any commits you make in this state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may do so (now or later) by using `-b` with the checkout command again. Example:

```
git checkout -b <new-branch-name>
```

HEAD is now at 848679e Cleaning and fixing bugs

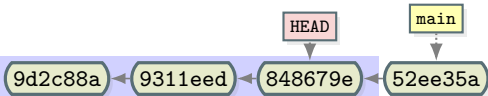
```
#> git switch main
```

Previous HEAD position was 848679e Cleaning and fixing bugs

Switched to branch 'main'

Example: 'checkout' vs. 'reset' (1/2)

```
#> git log --oneline
52ee35a ((HEAD -> main) Adding a README file
848679e Cleaning and fixing bugs
9311eed Adding a .gitignore file
9d2c88a Initial commit
#> git checkout HEAD~
Note: checking out 'HEAD~'.
```



You are in 'detached HEAD' state. You can look around, make experimental changes and commit them, and you can discard any commits you make in this state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may do so (now or later) by using `-b` with the checkout command again. Example:

```
git checkout -b <new-branch-name>
```

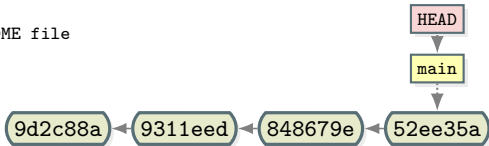
HEAD is now at 848679e Cleaning and fixing bugs

```
#> git switch main
```

```
Previous HEAD position was 848679e Cleaning and fixing bugs
Switched to branch 'main'
```

Example: 'checkout' vs. 'reset' (1/2)

```
#> git log --oneline
52ee35a ((HEAD -> main) Adding a README file
848679e Cleaning and fixing bugs
9311eed Adding a .gitignore file
9d2c88a Initial commit
#> git checkout HEAD~
Note: checking out 'HEAD~'.
```



You are in 'detached HEAD' state. You can look around, make experimental changes and commit them, and you can discard any commits you make in this state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may do so (now or later) by using `-b` with the checkout command again. Example:

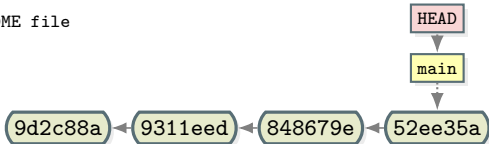
```
git checkout -b <new-branch-name>
```

HEAD is now at 848679e Cleaning and fixing bugs

```
#> git switch main
Previous HEAD position was 848679e Cleaning and fixing bugs
Switched to branch 'main'
```

Example: 'checkout' vs. 'reset' (2/2)

```
#> git log --oneline
52ee35a ((HEAD -> main) Adding a README file
848679e Cleaning and fixing bugs
9311eed Adding a .gitignore file
9d2c88a Initial commit
#> git reset HEAD~
#> git status
On branch main
Untracked files:
  (use "git add <file>..." to include in what will be committed)
```



README

```
nothing added to commit but untracked files present (use "git add" to track)
#> git checkout main
Already on 'main'
#> git add README
#> git commit -m "Adding a README file"
[main a89420a] Adding a README file
 1 file changed, 1 insertion(+)
 create mode 100644 README
#> git reset --hard HEAD~
HEAD is now at 848679e Cleaning and fixing bugs
#> git status
On branch main
nothing to commit, working tree clean
```

Example: 'checkout' vs. 'reset' (2/2)

```
#> git log --oneline
52ee35a ((HEAD -> main) Adding a README file
848679e Cleaning and fixing bugs
9311eed Adding a .gitignore file
9d2c88a Initial commit
```

```
#> git reset HEAD~
```

```
#> git status
```

```
On branch main
```

```
Untracked files:
```

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

README

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

```
#> git checkout main
```

```
Already on 'main'
```

```
#> git add README
```

```
#> git commit -m "Adding a README file"
```

```
[main a89420a] Adding a README file
```

```
1 file changed, 1 insertion(+)
```

```
create mode 100644 README
```

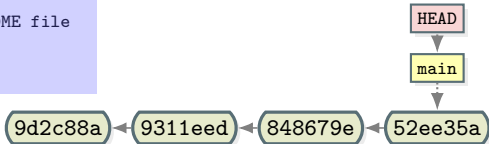
```
#> git reset --hard HEAD~
```

```
HEAD is now at 848679e Cleaning and fixing bugs
```

```
#> git status
```

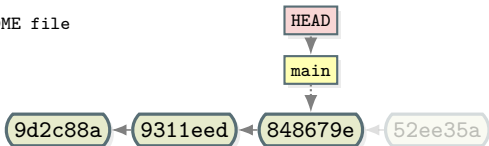
```
On branch main
```

```
nothing to commit, working tree clean
```



Example: 'checkout' vs. 'reset' (2/2)

```
#> git log --oneline
52ee35a ((HEAD -> main) Adding a README file
848679e Cleaning and fixing bugs
9311eed Adding a .gitignore file
9d2c88a Initial commit
#> git reset HEAD~
#> git status
On branch main
Untracked files:
  (use "git add <file>..." to include in what will be committed)
```



README

```
nothing added to commit but untracked files present (use "git add" to track)
#> git checkout main
Already on 'main'
#> git add README
#> git commit -m "Adding a README file"
[main a89420a] Adding a README file
 1 file changed, 1 insertion(+)
 create mode 100644 README
#> git reset --hard HEAD~
HEAD is now at 848679e Cleaning and fixing bugs
#> git status
On branch main
nothing to commit, working tree clean
```

Example: 'checkout' vs. 'reset' (2/2)

```
#> git log --oneline
52ee35a ((HEAD -> main) Adding a README file
848679e Cleaning and fixing bugs
9311eed Adding a .gitignore file
9d2c88a Initial commit
```

```
#> git reset HEAD~
```

```
#> git status
```

```
On branch main
```

```
Untracked files:
```

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

```
README
```

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

```
#> git checkout main
```

```
Already on 'main'
```

```
#> git add README
```

```
#> git commit -m "Adding a README file"
```

```
[main a89420a] Adding a README file
```

```
1 file changed, 1 insertion(+)
```

```
create mode 100644 README
```

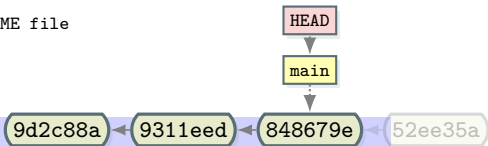
```
#> git reset --hard HEAD~
```

```
HEAD is now at 848679e Cleaning and fixing bugs
```

```
#> git status
```

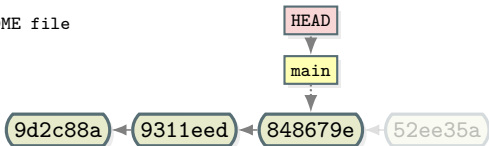
```
On branch main
```

```
nothing to commit, working tree clean
```



Example: 'checkout' vs. 'reset' (2/2)

```
#> git log --oneline
52ee35a ((HEAD -> main) Adding a README file
848679e Cleaning and fixing bugs
9311eed Adding a .gitignore file
9d2c88a Initial commit
#> git reset HEAD~
#> git status
On branch main
Untracked files:
  (use "git add <file>..." to include in what will be committed)
```



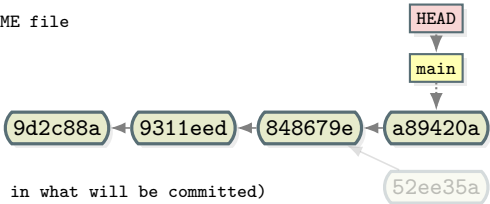
README

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

```
#> git checkout main
Already on 'main'
#> git add README
#> git commit -m "Adding a README file"
[main a89420a] Adding a README file
 1 file changed, 1 insertion(+)
 create mode 100644 README
#> git reset --hard HEAD~
HEAD is now at 848679e Cleaning and fixing bugs
#> git status
On branch main
nothing to commit, working tree clean
```

Example: 'checkout' vs. 'reset' (2/2)

```
#> git log --oneline
52ee35a ((HEAD -> main) Adding a README file
848679e Cleaning and fixing bugs
9311eed Adding a .gitignore file
9d2c88a Initial commit
#> git reset HEAD~
#> git status
On branch main
Untracked files:
  (use "git add <file>..." to include in what will be committed)
```

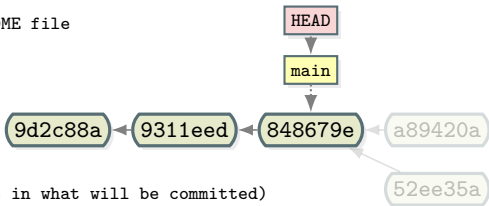


README

```
nothing added to commit but untracked files present (use "git add" to track)
#> git checkout main
Already on 'main'
#> git add README
#> git commit -m "Adding a README file"
[main a89420a] Adding a README file
1 file changed, 1 insertion(+)
create mode 100644 README
#> git reset --hard HEAD~
HEAD is now at 848679e Cleaning and fixing bugs
#> git status
On branch main
nothing to commit, working tree clean
```

Example: 'checkout' vs. 'reset' (2/2)

```
#> git log --oneline
52ee35a ((HEAD -> main) Adding a README file
848679e Cleaning and fixing bugs
9311eed Adding a .gitignore file
9d2c88a Initial commit
#> git reset HEAD~
#> git status
On branch main
Untracked files:
  (use "git add <file>..." to include in what will be committed)
```

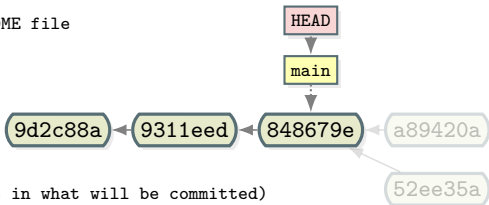


README

```
nothing added to commit but untracked files present (use "git add" to track)
#> git checkout main
Already on 'main'
#> git add README
#> git commit -m "Adding a README file"
[main a89420a] Adding a README file
 1 file changed, 1 insertion(+)
 create mode 100644 README
#> git reset --hard HEAD~
HEAD is now at 848679e Cleaning and fixing bugs
#> git status
On branch main
nothing to commit, working tree clean
```

Example: 'checkout' vs. 'reset' (2/2)

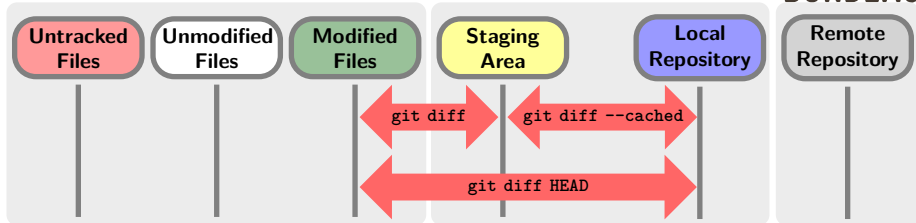
```
#> git log --oneline
52ee35a ((HEAD -> main) Adding a README file
848679e Cleaning and fixing bugs
9311eed Adding a .gitignore file
9d2c88a Initial commit
#> git reset HEAD~
#> git status
On branch main
Untracked files:
  (use "git add <file>..." to include in what will be committed)
```



README

```
nothing added to commit but untracked files present (use "git add" to track)
#> git checkout main
Already on 'main'
#> git add README
#> git commit -m "Adding a README file"
[main a89420a] Adding a README file
 1 file changed, 1 insertion(+)
 create mode 100644 README
#> git reset --hard HEAD~
HEAD is now at 848679e Cleaning and fixing bugs
#> git status
On branch main
nothing to commit, working tree clean
```

Command: 'diff'



'`git diff`': Show the differences between two states of the history.

- '`git diff`': Displays changes between the tracked modified files and the staging area.
- '`git diff --cached`': Displays changes between the staging area and current HEAD.
- '`git diff HEAD`': Displays changes between the tracked modified files and current HEAD.
- '`git diff HEAD HEAD~`': Displays changes between two states (HEAD and its ancestor).

```
#> git diff
diff --git a/README b/README ← Names of the diffed files
index 980a0d5..2c1251f 100644
--- a/README ← Hashes of the diffed commits
+++ b/README
@@ -1 +1,2 @@
Hello World!
+Added to README
```

Unnecessary trailing whitespaces are marked in red!

1 Introduction

2 Basic Usages

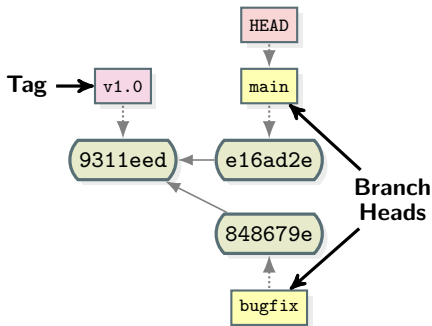
- Command Syntax
- Configuration
- Handling the Index
- Dealing with History
- **Using Branches**
- Getting a Repository
- Synchronize with Remote
- Solving Conflicts
- Managing Remotes & Other Useful Commands

3 Development Workflows

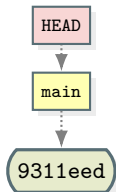
- Centralized Workflow
- Feature Branch Workflow
- Forking Workflow
- Massively Distributed Workflow
- Best Practices

```
#> git log --oneline --graph --all
* e16ad2e (HEAD -> main) Adding a README file
| * 848679e (bugfix) Cleaning and fixing bugs
|/
* 9311eed (tag: v1.0) Initial commit
```

- **Tag:** A label on a commit that will stay forever (except if you delete it).
- **Branch:** Denoted by a label marking the head of the branch. The label will be pushed forward if a new commit occurs on the branch.



```
#> git log --oneline
9311eed (HEAD -> main) Initial commit
#> git tag "v1.0"
#> git log --oneline
9311eed (HEAD -> main, tag: v1.0) Initial commit
#> git checkout -b bugfix
Switched to a new branch 'bugfix'
#> emacs project.c
... Cleaning and fixing bugs ...
#> git commit -a -m "Cleaning and fixing bugs"
[bugfix 848679e] Cleaning and fixing bugs
 1 file changed, 7 insertion(+)
#> git switch main
Switched to branch 'main'
#> emacs README &
... Adding a README file ...
#> git add README
#> git commit -m "Adding a README file"
[main e16ad2e] Adding a README file
 create mode 100644 README
#> git log --oneline --graph --all
* e16ad2e (HEAD -> main) Adding a README file
| * 848679e (bugfix) Cleaning and fixing bugs
|/
* 9311eed (tag: v1.0) Initial commit
```




```
#> git log --oneline
9311eed (HEAD -> main) Initial commit

#> git tag "v1.0"
#> git log --oneline
9311eed (HEAD -> main, tag: v1.0) Initial commit

#> git checkout -b bugfix
Switched to a new branch 'bugfix'

#> emacs project.c
... Cleaning and fixing bugs ...

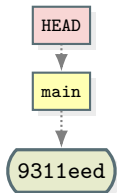
#> git commit -a -m "Cleaning and fixing bugs"
[bugfix 848679e] Cleaning and fixing bugs
 1 file changed, 7 insertion(+)

#> git switch main
Switched to branch 'main'

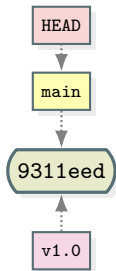
#> emacs README &
... Adding a README file ...

#> git add README
#> git commit -m "Adding a README file"
[main e16ad2e] Adding a README file
 create mode 100644 README

#> git log --oneline --graph --all
* e16ad2e (HEAD -> main) Adding a README file
| * 848679e (bugfix) Cleaning and fixing bugs
|/
* 9311eed (tag: v1.0) Initial commit
```

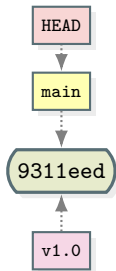


```
#> git log --oneline
9311eed (HEAD -> main) Initial commit
#> git tag "v1.0"
#> git log --oneline
9311eed (HEAD -> main, tag: v1.0) Initial commit
#> git checkout -b bugfix
Switched to a new branch 'bugfix'
#> emacs project.c
... Cleaning and fixing bugs ...
#> git commit -a -m "Cleaning and fixing bugs"
[bugfix 848679e] Cleaning and fixing bugs
 1 file changed, 7 insertion(+)
#> git switch main
Switched to branch 'main'
#> emacs README &
... Adding a README file ...
#> git add README
#> git commit -m "Adding a README file"
[main e16ad2e] Adding a README file
 create mode 100644 README
#> git log --oneline --graph --all
* e16ad2e (HEAD -> main) Adding a README file
| * 848679e (bugfix) Cleaning and fixing bugs
|/
* 9311eed (tag: v1.0) Initial commit
```

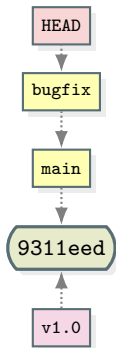


Example: Managing Branches and Tags

```
#> git log --oneline
9311eed (HEAD -> main) Initial commit
#> git tag "v1.0"
#> git log --oneline
9311eed (HEAD -> main, tag: v1.0) Initial commit
#> git checkout -b bugfix
Switched to a new branch 'bugfix'
#> emacs project.c
... Cleaning and fixing bugs ...
#> git commit -a -m "Cleaning and fixing bugs"
[bugfix 848679e] Cleaning and fixing bugs
 1 file changed, 7 insertion(+)
#> git switch main
Switched to branch 'main'
#> emacs README &
... Adding a README file ...
#> git add README
#> git commit -m "Adding a README file"
[main e16ad2e] Adding a README file
 create mode 100644 README
#> git log --oneline --graph --all
* e16ad2e (HEAD -> main) Adding a README file
| * 848679e (bugfix) Cleaning and fixing bugs
|/
* 9311eed (tag: v1.0) Initial commit
```

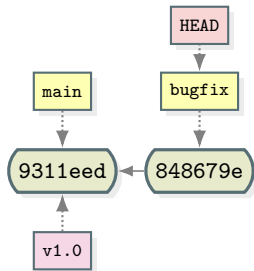


```
#> git log --oneline
9311eed (HEAD -> main) Initial commit
#> git tag "v1.0"
#> git log --oneline
9311eed (HEAD -> main, tag: v1.0) Initial commit
#> git checkout -b bugfix
Switched to a new branch 'bugfix'
#> emacs project.c
... Cleaning and fixing bugs ...
#> git commit -a -m "Cleaning and fixing bugs"
[bugfix 848679e] Cleaning and fixing bugs
 1 file changed, 7 insertion(+)
#> git switch main
Switched to branch 'main'
#> emacs README &
... Adding a README file ...
#> git add README
#> git commit -m "Adding a README file"
[main e16ad2e] Adding a README file
 create mode 100644 README
#> git log --oneline --graph --all
* e16ad2e (HEAD -> main) Adding a README file
| * 848679e (bugfix) Cleaning and fixing bugs
|/
* 9311eed (tag: v1.0) Initial commit
```



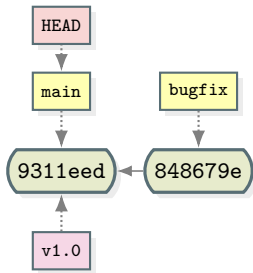
Example: Managing Branches and Tags

```
#> git log --oneline
9311eed (HEAD -> main) Initial commit
#> git tag "v1.0"
#> git log --oneline
9311eed (HEAD -> main, tag: v1.0) Initial commit
#> git checkout -b bugfix
Switched to a new branch 'bugfix'
#> emacs project.c
... Cleaning and fixing bugs ...
#> git commit -a -m "Cleaning and fixing bugs"
[bugfix 848679e] Cleaning and fixing bugs
1 file changed, 7 insertion(+)
#> git switch main
Switched to branch 'main'
#> emacs README &
... Adding a README file ...
#> git add README
#> git commit -m "Adding a README file"
[main e16ad2e] Adding a README file
create mode 100644 README
#> git log --oneline --graph --all
* e16ad2e (HEAD -> main) Adding a README file
| * 848679e (bugfix) Cleaning and fixing bugs
|/
* 9311eed (tag: v1.0) Initial commit
```



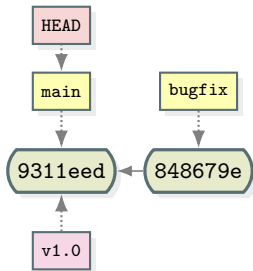
Example: Managing Branches and Tags

```
#> git log --oneline
9311eed (HEAD -> main) Initial commit
#> git tag "v1.0"
#> git log --oneline
9311eed (HEAD -> main, tag: v1.0) Initial commit
#> git checkout -b bugfix
Switched to a new branch 'bugfix'
#> emacs project.c
... Cleaning and fixing bugs ...
#> git commit -a -m "Cleaning and fixing bugs"
[bugfix 848679e] Cleaning and fixing bugs
1 file changed, 7 insertion(+)
#> git switch main
Switched to branch 'main'
#> emacs README &
... Adding a README file ...
#> git add README
#> git commit -m "Adding a README file"
[main e16ad2e] Adding a README file
create mode 100644 README
#> git log --oneline --graph --all
* e16ad2e (HEAD -> main) Adding a README file
| * 848679e (bugfix) Cleaning and fixing bugs
|/
* 9311eed (tag: v1.0) Initial commit
```



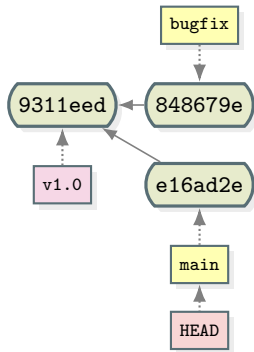
Example: Managing Branches and Tags

```
#> git log --oneline
9311eed (HEAD -> main) Initial commit
#> git tag "v1.0"
#> git log --oneline
9311eed (HEAD -> main, tag: v1.0) Initial commit
#> git checkout -b bugfix
Switched to a new branch 'bugfix'
#> emacs project.c
... Cleaning and fixing bugs ...
#> git commit -a -m "Cleaning and fixing bugs"
[bugfix 848679e] Cleaning and fixing bugs
 1 file changed, 7 insertion(+)
#> git switch main
Switched to branch 'main'
#> emacs README &
... Adding a README file ...
#> git add README
#> git commit -m "Adding a README file"
[main e16ad2e] Adding a README file
 create mode 100644 README
#> git log --oneline --graph --all
* e16ad2e (HEAD -> main) Adding a README file
| * 848679e (bugfix) Cleaning and fixing bugs
|/
* 9311eed (tag: v1.0) Initial commit
```



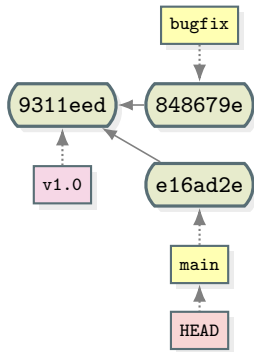
Example: Managing Branches and Tags

```
#> git log --oneline
9311eed (HEAD -> main) Initial commit
#> git tag "v1.0"
#> git log --oneline
9311eed (HEAD -> main, tag: v1.0) Initial commit
#> git checkout -b bugfix
Switched to a new branch 'bugfix'
#> emacs project.c
... Cleaning and fixing bugs ...
#> git commit -a -m "Cleaning and fixing bugs"
[bugfix 848679e] Cleaning and fixing bugs
 1 file changed, 7 insertion(+)
#> git switch main
Switched to branch 'main'
#> emacs README &
... Adding a README file ...
#> git add README
#> git commit -m "Adding a README file"
[main e16ad2e] Adding a README file
 create mode 100644 README
#> git log --oneline --graph --all
* e16ad2e (HEAD -> main) Adding a README file
| * 848679e (bugfix) Cleaning and fixing bugs
|/
* 9311eed (tag: v1.0) Initial commit
```



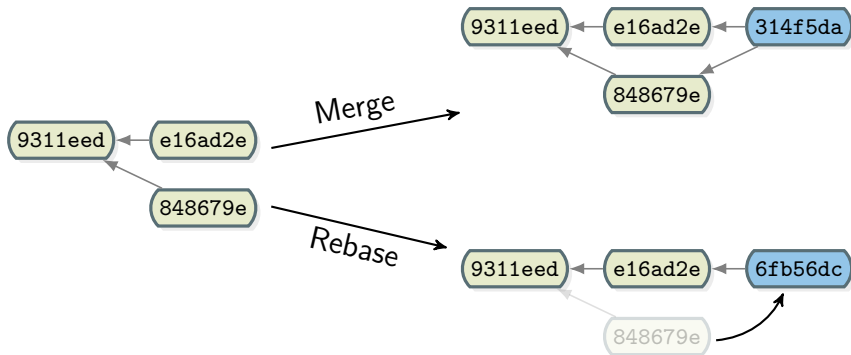
Example: Managing Branches and Tags

```
#> git log --oneline
9311eed (HEAD -> main) Initial commit
#> git tag "v1.0"
#> git log --oneline
9311eed (HEAD -> main, tag: v1.0) Initial commit
#> git checkout -b bugfix
Switched to a new branch 'bugfix'
#> emacs project.c
... Cleaning and fixing bugs ...
#> git commit -a -m "Cleaning and fixing bugs"
[bugfix 848679e] Cleaning and fixing bugs
 1 file changed, 7 insertion(+)
#> git switch main
Switched to branch 'main'
#> emacs README &
... Adding a README file ...
#> git add README
#> git commit -m "Adding a README file"
[main e16ad2e] Adding a README file
 create mode 100644 README
#> git log --oneline --graph --all
* e16ad2e (HEAD -> main) Adding a README file
| * 848679e (bugfix) Cleaning and fixing bugs
|/
* 9311eed (tag: v1.0) Initial commit
```



Merge

Merge is a **non-destructive** operation on the history. It is used when the history need to be kept. Typically when incorporating new features in common branches (e.g. main).

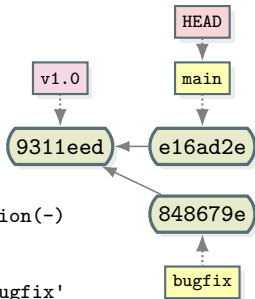


Rebase

An alternative to merge that **rewrite** its own history. It is used to keep your current work-in-progress branches up to date with main when updating.

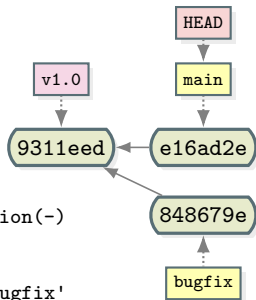
Example: Merging Branches

```
#> git branch
  bugfix
* main
#> git merge bugfix
Removing project
Merge made by the 'recursive' strategy.
 Makefile | 2 +-
 project  | Bin 17084 -> 0 bytes
 project.c | 2 ++
3 files changed, 3 insertions(+), 1 deletion(-)
delete mode 100644 project
#> git log --oneline --graph --all
* 314f5da (HEAD -> main) Merge branch 'bugfix'
| \
| * 848679e (bugfix) Cleaning and fixing bugs
| /
* 9311eed (tag: v1.0) Initial commit
```



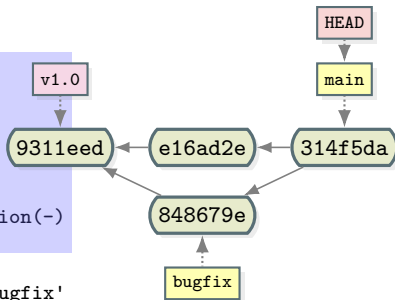
Example: Merging Branches

```
#> git branch  
    bugfix  
* main  
#> git merge bugfix  
Removing project  
Merge made by the 'recursive' strategy.  
    Makefile | 2 +-  
    project  | Bin 17084 -> 0 bytes  
    project.c | 2 ++  
3 files changed, 3 insertions(+), 1 deletion(-)  
delete mode 100644 project  
#> git log --oneline --graph --all  
* 314f5da (HEAD -> main) Merge branch 'bugfix'  
|\  
| * 848679e (bugfix) Cleaning and fixing bugs  
|/  
* 9311eed (tag: v1.0) Initial commit
```



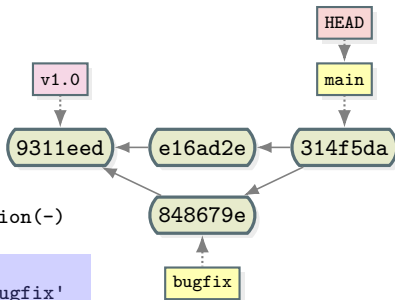
Example: Merging Branches

```
#> git branch
  bugfix
* main
#> git merge bugfix
Removing project
Merge made by the 'recursive' strategy.
 Makefile | 2 +-
 project  | Bin 17084 -> 0 bytes
 project.c | 2 ++
 3 files changed, 3 insertions(+), 1 deletion(-)
 delete mode 100644 project
#> git log --oneline --graph --all
* 314f5da (HEAD -> main) Merge branch 'bugfix'
| \
| * 848679e (bugfix) Cleaning and fixing bugs
| /
* 9311eed (tag: v1.0) Initial commit
```



Example: Merging Branches

```
#> git branch
  bugfix
* main
#> git merge bugfix
Removing project
Merge made by the 'recursive' strategy.
 Makefile | 2 +-
 project  | Bin 17084 -> 0 bytes
 project.c | 2 ++
3 files changed, 3 insertions(+), 1 deletion(-)
delete mode 100644 project
#> git log --oneline --graph --all
* 314f5da (HEAD -> main) Merge branch 'bugfix'
| \
| * 848679e (bugfix) Cleaning and fixing bugs
| /
* 9311eed (tag: v1.0) Initial commit
```



Example: Rebasing Branches

```
#> git branch  
  bugfix  
* main  
#> git switch bugfix  
Switched to branch 'bugfix'
```

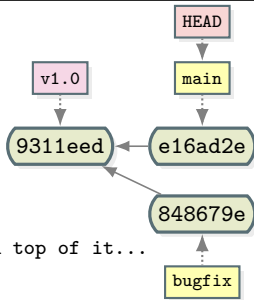
```
#> git branch  
* bugfix  
  main  
#> git rebase main
```

First, rewinding head to replay your work on top of it...

Applying: Cleaning and fixing bugs

```
#> git log --oneline --graph --all
```

- * 6fb56dc (HEAD -> bugfix) Cleaning and fixing bugs
- * e16ad2e (main) Adding a README file
- * 9311eed (tag: v1.0) Initial commit



Example: Rebasing Branches

```
#> git branch  
bugfix
```

```
* main
```

```
#> git switch bugfix
```

```
Switched to branch 'bugfix'
```

```
#> git branch
```

```
* bugfix
```

```
main
```

```
#> git rebase main
```

```
First, rewinding head to replay your work on top of it...
```

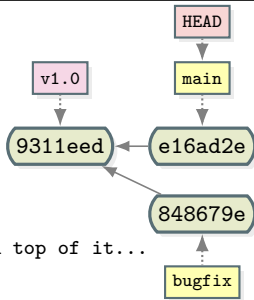
```
Applying: Cleaning and fixing bugs
```

```
#> git log --oneline --graph --all
```

```
* 6fb56dc (HEAD -> bugfix) Cleaning and fixing bugs
```

```
* e16ad2e (main) Adding a README file
```

```
* 9311eed (tag: v1.0) Initial commit
```



Example: Rebasing Branches

```
#> git branch
```

```
  bugfix
```

```
* main
```

```
#> git switch bugfix
```

```
Switched to branch 'bugfix'
```

```
#> git branch
```

```
* bugfix
```

```
  main
```

```
#> git rebase main
```

```
First, rewinding head to replay your work on top of it...
```

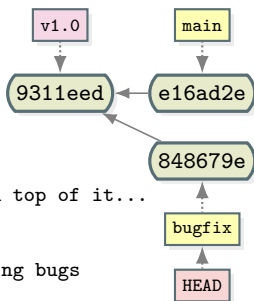
```
Applying: Cleaning and fixing bugs
```

```
#> git log --oneline --graph --all
```

```
* 6fb56dc (HEAD -> bugfix) Cleaning and fixing bugs
```

```
* e16ad2e (main) Adding a README file
```

```
* 9311eed (tag: v1.0) Initial commit
```



Example: Rebasing Branches

```
#> git branch
  bugfix
* main
#> git switch bugfix
Switched to branch 'bugfix'
#> git branch
* bugfix
  main
```

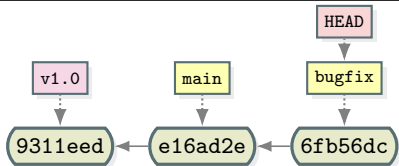
```
#> git rebase main
```

First, rewinding head to replay your work on top of it...

Applying: Cleaning and fixing bugs

```
#> git log --oneline --graph --all
```

```
* 6fb56dc (HEAD -> bugfix) Cleaning and fixing bugs
* e16ad2e (main) Adding a README file
* 9311eed (tag: v1.0) Initial commit
```



Example: Rebasing Branches

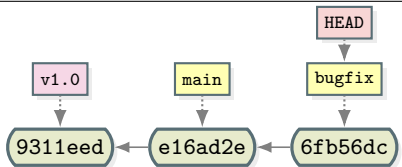
```
#> git branch
  bugfix
* main
#> git switch bugfix
Switched to branch 'bugfix'
#> git branch
* bugfix
  main
#> git rebase main
```

First, rewinding head to replay your work on top of it...

Applying: Cleaning and fixing bugs

```
#> git log --oneline --graph --all
```

```
* 6fb56dc (HEAD -> bugfix) Cleaning and fixing bugs
* e16ad2e (main) Adding a README file
* 9311eed (tag: v1.0) Initial commit
```



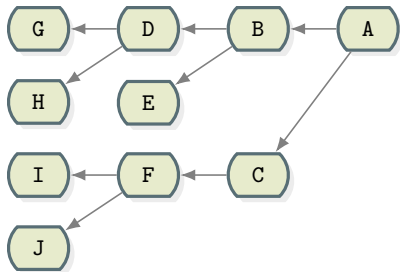
Was it 'HEAD~3~2' or 'HEAD^2~3' ?

As we can branch now, we cannot cope only with the HEAD~ notation. We need to be able to navigate in a bi-dimensional space, so we need to introduce another HEAD^ notation which allows to select a branch.

Here are a few examples:

- HEAD~2: The first ancestor of the ancestor of HEAD.
- HEAD^3: The third ancestor of HEAD (starting from upper side).

A =	=	A^0		
B = A^	=	A^1	=	A~1
C = A^2	=	A^2		
D = A^^	=	A^1^1	=	A~2
E = B^2	=	A^^2		
F = B^3	=	A^^3		
G = A^^^	=	A^1^1^1	=	A~3
H = D^2	=	B^^2	=	A^^^2 = A~2^2
I = F^	=	B^3^	=	A^^3^
J = F^2	=	B^3^2	=	A^^3^2



1 Introduction

2 Basic Usages

- Command Syntax
- Configuration
- Handling the Index
- Dealing with History
- Using Branches
- **Getting a Repository**
- Synchronize with Remote
- Solving Conflicts
- Managing Remotes & Other Useful Commands

3 Development Workflows

- Centralized Workflow
- Feature Branch Workflow
- Forking Workflow
- Massively Distributed Workflow
- Best Practices

A **Git Repository** is a hierarchy of files and directories (working tree) with a hidden `.git/` directory in its top directory (repository data).

A `.git/` directory contains (not exhaustive, see `'gitrepository-layout(5)'`):

```
.git/
|
+- branches/      # Deprecated way of storing branches
+- COMMIT_EDITMSG # Last commit message
+- config         # Configuration file (see later)
+- description    # Used by gitweb to store project description
+- HEAD          # Pointer on refs/heads to the current branch HEAD
+- hooks/        # A set of scripts automatically applied on commit/pull/...
+- index         # Staging area (difference between working tree and HEAD)
+- info/         # Additional information about the repository
| +- exclude     # Locally excluded files (won't be shared as the .gitignore)
+- objects/      # Contains all the files, commits and trees of your history
+- refs/         # References are stored in subdirectories of this directory
| +- heads/foo   # Records tip-of-the-tree commit objects of branch 'foo'
| +- remotes/foo # Records tip-of-the-tree commit objects of remote 'foo'
| +- tags/foo    # Records any object 'foo'
\-- remotes      # URL and default refnames for remote repositories
```

Initialize an empty repository (working tree and .git/)

(Useful to start a new project with its own history)

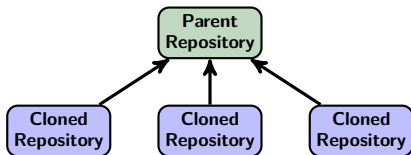
```
#> git init my_project/  
Initialized empty Git repository in /home/user/my_project/.git/  
#> cd my_project/  
#> git status  
On branch main  
No commits yet  
nothing to commit (create/copy files and use "git add" to track)  
#> echo "Hello!" > README  
#> git add .  
#> git commit -m "Initial commit"  
[main (root-commit) 1305a7c] Initial commit  
1 file changed, 1 insertion(+)  
create mode 100644 README
```

Initialize a '*bare*' repository (no working tree, only history)

(Useful to store code history on a server to synchronize with others)

```
#> git init --bare my_project/  
Initialized empty Git repository in /home/user/my_project/.git/  
#> ls my_project/  
branches/  config  description  HEAD  hooks/  info/  objects/  refs/
```

Clone a repository is **equivalent to create a copy and keep a link towards the parent repository** (configured to 'push' to parent and 'pull' from it).



Copy an existing repository (same file-system)

```
#> git clone my_project/ my_project-clone
Cloning into 'my_project-clone'...
done.
```

Copy an existing repository (remote repository through ssh)

```
#> git clone ssh://user@localhost/home/user/my_project my_project-remote
Cloning into 'my_project-remote'...
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (3/3), done.
```

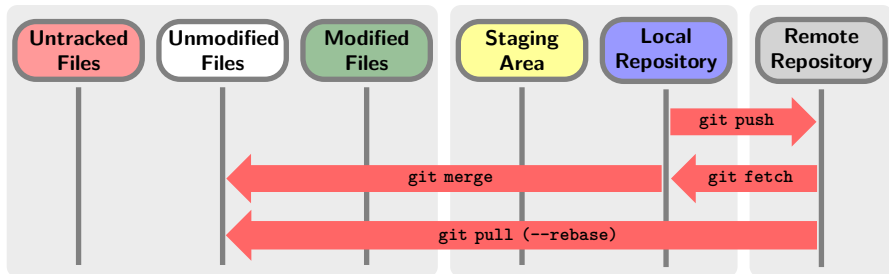

1 Introduction

2 Basic Usages

- Command Syntax
- Configuration
- Handling the Index
- Dealing with History
- Using Branches
- Getting a Repository
- **Synchronize with Remote**
- Solving Conflicts
- Managing Remotes & Other Useful Commands

3 Development Workflows

- Centralized Workflow
- Feature Branch Workflow
- Forking Workflow
- Massively Distributed Workflow
- Best Practices



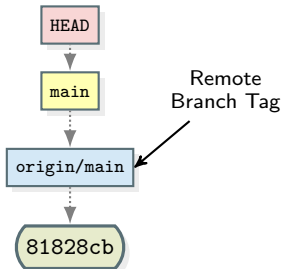
- **git push:** Update local repository with the content of a remote repository. (git refuse to destroy remote objects by default, use 'git push --force' when needed)
- **git fetch:** Download the history from remote repository to local repository.
- **git merge:** Join two (or more) code histories together.
- **git pull:** Perform a fetch and a merge at once. Consider this operation as “*dangerous*” and prefer to use fetch and merge or 'pull --rebase'.
- **git remote:** Manage the remote repositories of this local repository.

```
#> git clone my_repo/ my_project
Cloning into 'my_project'...
done.
#> cd my_project/
#> git log --oneline
81828cb (HEAD -> main, origin/main, origin/HEAD) Initial release
#> git branch -a
* main
  remotes/origin/HEAD -> origin/main
  remotes/origin/main
#> git remote -v
origin          /home/user/my_repo/ (fetch)
origin          /home/user/my_repo/ (push)
#> nano README
#> git commit -a
[main 069db25] Added a line in README file
 1 file changed, 1 insertion(+)
#> git log --oneline
069db25 (HEAD -> main) Added a line in README file
81828cb (origin/main, origin/HEAD) Initial release
#> git push
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Writing objects: 100% (3/3), 285 bytes | 285.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To /home/user/my_repo/
 81828cb..069db25  main -> main
```

```
#> git clone my_repo/ my_project
Cloning into 'my_project'...
done.
#> cd my_project/
#> git log --oneline
81828cb (HEAD -> main, origin/main, origin/HEAD) Initial release
#> git branch -a
* main
remotes/origin/HEAD -> origin/main
remotes/origin/main
#> git remote -v
origin          /home/user/my_repo/ (fetch)
origin          /home/user/my_repo/ (push)
#> nano README
#> git commit -a
[main 069db25] Added a line in README file
1 file changed, 1 insertion(+)
#> git log --oneline
069db25 (HEAD -> main) Added a line in README file
81828cb (origin/main, origin/HEAD) Initial release
#> git push
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Writing objects: 100% (3/3), 285 bytes | 285.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To /home/user/my_repo/
81828cb..069db25 main -> main
```

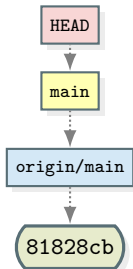
Example: Synchronize with Remotes

```
#> git clone my_repo/ my_project
Cloning into 'my_project'...
done.
#> cd my_project/
#> git log --oneline
81828cb (HEAD -> main, origin/main, origin/HEAD) Initial release
#> git branch -a
* main
remotes/origin/HEAD -> origin/main
remotes/origin/main
#> git remote -v
origin      /home/user/my_repo/ (fetch)
origin      /home/user/my_repo/ (push)
#> nano README
#> git commit -a
[main 069db25] Added a line in README file
1 file changed, 1 insertion(+)
#> git log --oneline
069db25 (HEAD -> main) Added a line in README file
81828cb (origin/main, origin/HEAD) Initial release
#> git push
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Writing objects: 100% (3/3), 285 bytes | 285.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To /home/user/my_repo/
81828cb..069db25  main -> main
```



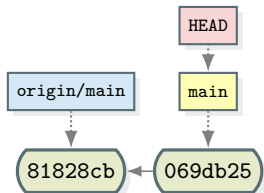
Example: Synchronize with Remotes

```
#> git clone my_repo/ my_project
Cloning into 'my_project'...
done.
#> cd my_project/
#> git log --oneline
81828cb (HEAD -> main, origin/main, origin/HEAD) Initial release
#> git branch -a
* main
remotes/origin/HEAD -> origin/main
remotes/origin/main
#> git remote -v
origin      /home/user/my_repo/ (fetch)
origin      /home/user/my_repo/ (push)
#> nano README
#> git commit -a
[main 069db25] Added a line in README file
1 file changed, 1 insertion(+)
#> git log --oneline
069db25 (HEAD -> main) Added a line in README file
81828cb (origin/main, origin/HEAD) Initial release
#> git push
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Writing objects: 100% (3/3), 285 bytes | 285.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To /home/user/my_repo/
81828cb..069db25  main -> main
```



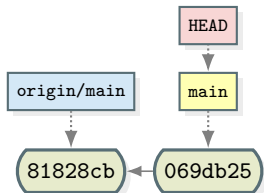
Example: Synchronize with Remotes

```
#> git clone my_repo/ my_project
Cloning into 'my_project'...
done.
#> cd my_project/
#> git log --oneline
81828cb (HEAD -> main, origin/main, origin/HEAD) Initial release
#> git branch -a
* main
  remotes/origin/HEAD -> origin/main
  remotes/origin/main
#> git remote -v
origin      /home/user/my_repo/ (fetch)
origin      /home/user/my_repo/ (push)
#> nano README
#> git commit -a
[main 069db25] Added a line in README file
1 file changed, 1 insertion(+)
#> git log --oneline
069db25 (HEAD -> main) Added a line in README file
81828cb (origin/main, origin/HEAD) Initial release
#> git push
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Writing objects: 100% (3/3), 285 bytes | 285.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To /home/user/my_repo/
81828cb..069db25  main -> main
```



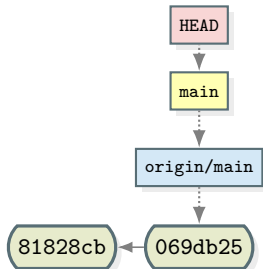
Example: Synchronize with Remotes

```
#> git clone my_repo/ my_project
Cloning into 'my_project'...
done.
#> cd my_project/
#> git log --oneline
81828cb (HEAD -> main, origin/main, origin/HEAD) Initial release
#> git branch -a
* main
  remotes/origin/HEAD -> origin/main
  remotes/origin/main
#> git remote -v
origin          /home/user/my_repo/ (fetch)
origin          /home/user/my_repo/ (push)
#> nano README
#> git commit -a
[main 069db25] Added a line in README file
1 file changed, 1 insertion(+)
#> git log --oneline
069db25 (HEAD -> main) Added a line in README file
81828cb (origin/main, origin/HEAD) Initial release
#> git push
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Writing objects: 100% (3/3), 285 bytes | 285.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To /home/user/my_repo/
 81828cb..069db25  main -> main
```



Example: Synchronize with Remotes

```
#> git clone my_repo/ my_project
Cloning into 'my_project'...
done.
#> cd my_project/
#> git log --oneline
81828cb (HEAD -> main, origin/main, origin/HEAD) Initial release
#> git branch -a
* main
  remotes/origin/HEAD -> origin/main
  remotes/origin/main
#> git remote -v
origin      /home/user/my_repo/ (fetch)
origin      /home/user/my_repo/ (push)
#> nano README
#> git commit -a
[main 069db25] Added a line in README file
 1 file changed, 1 insertion(+)
#> git log --oneline
069db25 (HEAD -> main) Added a line in README file
81828cb (origin/main, origin/HEAD) Initial release
#> git push
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Writing objects: 100% (3/3), 285 bytes | 285.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To /home/user/my_repo/
 81828cb..069db25  main -> main
```



```
#> git remote -v
origin          /home/user/my_repo/ (fetch)
origin          /home/user/my_repo/ (push)
#> git branch -a
* main
  remotes/origin/HEAD -> origin/main
  remotes/origin/main
#> git checkout -b bugfix
Switched to a new branch 'bugfix'

    ... work and commit on this branch ...

#> git push --set-upstream origin bugfix
Total 0 (delta 0), reused 0 (delta 0)
To /home/user/my_repo/
 * [new branch]      bugfix -> bugfix
Branch 'bugfix' set up to track remote branch 'bugfix' from 'origin'.
#> git branch -a
* bugfix
  main
  remotes/origin/HEAD -> origin/main
  remotes/origin/bugfix
  remotes/origin/main
```

Creating the main repository

```
#> git init --bare my_repo  
Initialized empty Git repository in /home/user/my_repo/
```

Cloning the main repository to project1/

```
#> git clone my_repo/ project1  
Cloning into 'project1'...  
warning: You appear to have cloned an empty repository.  
done.
```

Cloning the main repository to project2/

```
#> git clone my_repo/ project2  
Cloning into 'project2'...  
warning: You appear to have cloned an empty repository.  
done.
```

Example: 'push' (project1)

```
#> echo "Hello World!" > README
#> git add README
#> git commit -m "Initial commit"
[main (root-commit) 848679e] Initial commit
 1 file changed, 1 insertion(+)
 create mode 100644 README

#> echo "Second line" >> README
#> git commit -a -m "Second commit"
[main aea37e6] Second commit
 1 file changed, 1 insertion(+)

#> git push
Enumerating objects: 6, done.
Counting objects: 100% (6/6), done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (6/6), 471 bytes | 471.00 KiB/s, done.
Total 6 (delta 0), reused 0 (delta 0)
To /home/user/my_repo/
 * [new branch]      main -> main

#> git log --oneline
aea37e6 (HEAD -> main, origin/main, origin/HEAD) Second commit
848679e Initial commit
```

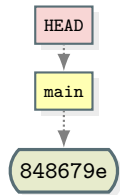
Example: 'push' (project1)

```
#> echo "Hello World!" > README
#> git add README
#> git commit -m "Initial commit"
[main (root-commit) 848679e] Initial commit
 1 file changed, 1 insertion(+)
 create mode 100644 README

#> echo "Second line" >> README
#> git commit -a -m "Second commit"
[main aea37e6] Second commit
 1 file changed, 1 insertion(+)

#> git push
Enumerating objects: 6, done.
Counting objects: 100% (6/6), done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (6/6), 471 bytes | 471.00 KiB/s, done.
Total 6 (delta 0), reused 0 (delta 0)
To /home/user/my_repo/
 * [new branch]      main -> main

#> git log --oneline
aea37e6 (HEAD -> main, origin/main, origin/HEAD) Second commit
848679e Initial commit
```



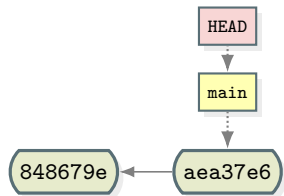
Example: 'push' (project1)

```
#> echo "Hello World!" > README
#> git add README
#> git commit -m "Initial commit"
[main (root-commit) 848679e] Initial commit
 1 file changed, 1 insertion(+)
 create mode 100644 README

#> echo "Second line" >> README
#> git commit -a -m "Second commit"
[main aea37e6] Second commit
 1 file changed, 1 insertion(+)

#> git push
Enumerating objects: 6, done.
Counting objects: 100% (6/6), done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (6/6), 471 bytes | 471.00 KiB/s, done.
Total 6 (delta 0), reused 0 (delta 0)
To /home/user/my_repo/
 * [new branch]      main -> main

#> git log --oneline
aea37e6 (HEAD -> main, origin/main, origin/HEAD) Second commit
848679e Initial commit
```



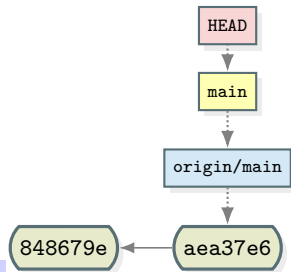
Example: 'push' (project1)

```
#> echo "Hello World!" > README
#> git add README
#> git commit -m "Initial commit"
[main (root-commit) 848679e] Initial commit
 1 file changed, 1 insertion(+)
 create mode 100644 README
```

```
#> echo "Second line" >> README
#> git commit -a -m "Second commit"
[main aea37e6] Second commit
 1 file changed, 1 insertion(+)
```

```
#> git push
Enumerating objects: 6, done.
Counting objects: 100% (6/6), done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (6/6), 471 bytes | 471.00 KiB/s, done.
Total 6 (delta 0), reused 0 (delta 0)
To /home/user/my_repo/
 * [new branch]      main -> main
```

```
#> git log --oneline
aea37e6 (HEAD -> main, origin/main, origin/HEAD) Second commit
848679e Initial commit
```

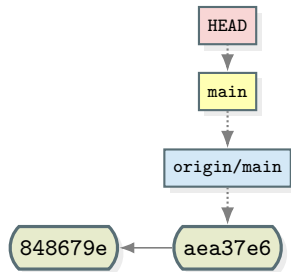


Example: 'push' (project1)

```
#> echo "Hello World!" > README
#> git add README
#> git commit -m "Initial commit"
[main (root-commit) 848679e] Initial commit
 1 file changed, 1 insertion(+)
 create mode 100644 README

#> echo "Second line" >> README
#> git commit -a -m "Second commit"
[main aea37e6] Second commit
 1 file changed, 1 insertion(+)

#> git push
Enumerating objects: 6, done.
Counting objects: 100% (6/6), done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (6/6), 471 bytes | 471.00 KiB/s, done.
Total 6 (delta 0), reused 0 (delta 0)
To /home/user/my_repo/
 * [new branch]      main -> main
```



```
#> git log --oneline
aea37e6 (HEAD -> main, origin/main, origin/HEAD) Second commit
848679e Initial commit
```



```
#> cd project2/

#> echo "John Doe <john.doe@student.edu" > AUTHORS
#> git add AUTHORS
#> git commit -m "Adding first author"
[main 7d385a4] Adding first author
 1 file changed, 1 insertion(+)
 create mode 100644 AUTHORS

#> git push
To /home/user/my_repo/
 ! [rejected]      main -> main (fetch first)
error: failed to push some refs to '/home/user/my_repo/'
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 integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

```
#> cd project2/

#> echo "John Doe <john.doe@student.edu" > AUTHORS
#> git add AUTHORS
#> git commit -m "Adding first author"
[main 7d385a4] Adding first author
 1 file changed, 1 insertion(+)
 create mode 100644 AUTHORS

#> git push
To /home/user/my_repo/
 ! [rejected]      main -> main (fetch first)
error: failed to push some refs to '/home/user/my_repo/'
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 integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

Example: 'push' Attempt (project2)

```
#> cd project2/
```

```
#> echo "John Doe <john.doe@student.edu" > AUTHORS  
#> git add AUTHORS  
#> git commit -m "Adding first author"  
[main 7d385a4] Adding first author  
1 file changed, 1 insertion(+)  
create mode 100644 AUTHORS
```

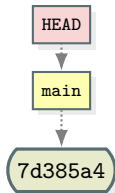
```
#> git push
```

```
To /home/user/my_repo/
```

```
! [rejected]      main -> main (fetch first)
```

```
error: failed to push some refs to '/home/user/my_repo/'
```

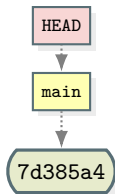
```
hint: Updates were rejected because the remote contains work that you do  
hint: not have locally. This is usually caused by another repository pushing  
hint: to the same ref. You may want to first integrate the remote changes  
hint: (e.g., 'git pull ...') before pushing again.  
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```



Example: 'push' Attempt (project2)

```
#> cd project2/

#> echo "John Doe <john.doe@student.edu" > AUTHORS
#> git add AUTHORS
#> git commit -m "Adding first author"
[main 7d385a4] Adding first author
1 file changed, 1 insertion(+)
create mode 100644 AUTHORS
```

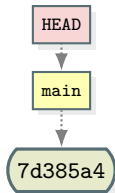


```
#> git push
To /home/user/my_repo/
 ! [rejected]      main -> main (fetch first)
error: failed to push some refs to '/home/user/my_repo/'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

Example: 'push' Attempt (project2)

```
#> cd project2/

#> echo "John Doe <john.doe@student.edu" > AUTHORS
#> git add AUTHORS
#> git commit -m "Adding first author"
[main 7d385a4] Adding first author
1 file changed, 1 insertion(+)
create mode 100644 AUTHORS
```



```
#> git push
To /home/user/my_repo/
 ! [rejected]          main -> main (fetch first)
error: failed to push some refs to '/home/user/my_repo/'
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 integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

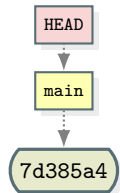
What do we do ???

Example: 'pull' Attempt (project2)

```
#> git pull
warning: no common commits
remote: Enumerating objects: 6, done.
remote: Counting objects: 100% (6/6), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 6 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (6/6), done.
From /home/user/my_repo
 * [new branch]      main      -> origin/main
fatal: refusing to merge unrelated histories
```

```
#> git log --oneline
7d385a4 (HEAD -> main) Adding first author
```

```
#> git push
To /home/user/my_repo/
 ! [rejected]        main -> main (fetch first)
error: failed to push some refs to '/home/user/my_repo/'
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 integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

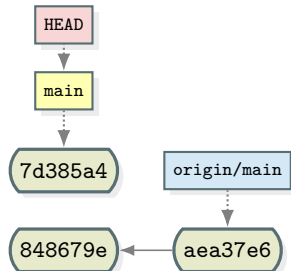


Example: 'pull' Attempt (project2)

```
#> git pull
warning: no common commits
remote: Enumerating objects: 6, done.
remote: Counting objects: 100% (6/6), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 6 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (6/6), done.
From /home/user/my_repo
 * [new branch]      main      -> origin/main
fatal: refusing to merge unrelated histories
```

```
#> git log --oneline
7d385a4 (HEAD -> main) Adding first author
```

```
#> git push
To /home/user/my_repo/
 ! [rejected]        main -> main (fetch first)
error: failed to push some refs to '/home/user/my_repo/'
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 integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

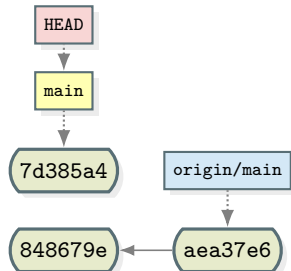


Example: 'pull' Attempt (project2)

```
#> git pull
warning: no common commits
remote: Enumerating objects: 6, done.
remote: Counting objects: 100% (6/6), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 6 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (6/6), done.
From /home/user/my_repo
 * [new branch]      main      -> origin/main
fatal: refusing to merge unrelated histories
```

```
#> git log --oneline
7d385a4 (HEAD -> main) Adding first author
```

```
#> git push
To /home/user/my_repo/
 ! [rejected]        main -> main (fetch first)
error: failed to push some refs to '/home/user/my_repo/'
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 integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

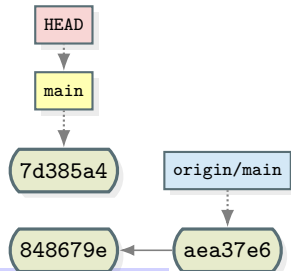


Example: 'pull' Attempt (project2)

```
#> git pull
warning: no common commits
remote: Enumerating objects: 6, done.
remote: Counting objects: 100% (6/6), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 6 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (6/6), done.
From /home/user/my_repo
 * [new branch]      main      -> origin/main
fatal: refusing to merge unrelated histories
```

```
#> git log --oneline
7d385a4 (HEAD -> main) Adding first author
```

```
#> git push
To /home/user/my_repo/
 ! [rejected]        main -> main (fetch first)
error: failed to push some refs to '/home/user/my_repo/'
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 integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

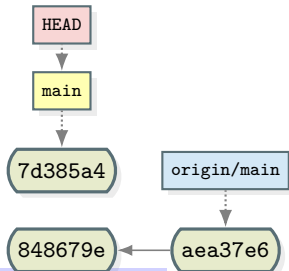


Example: 'pull' Attempt (project2)

```
#> git pull
warning: no common commits
remote: Enumerating objects: 6, done.
remote: Counting objects: 100% (6/6), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 6 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (6/6), done.
From /home/user/my_repo
 * [new branch]      main      -> origin/main
fatal: refusing to merge unrelated histories
```

```
#> git log --oneline
7d385a4 (HEAD -> main) Adding first author
```

```
#> git push
To /home/user/my_repo/
 ! [rejected]        main -> main (fetch first)
error: failed to push some refs to '/home/user/my_repo/'
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 integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```



What do we do ???

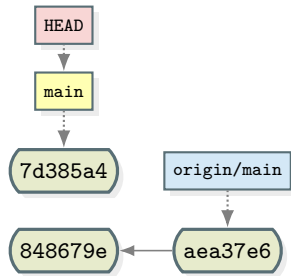
Example: 'pull --rebase' (project2)

```
#> git pull --rebase
First, rewinding head to replay your work on top of it...
Applying: Adding first author

#> git log --oneline
ce2f041 (HEAD -> main) Adding first author
88a843d (origin/main) Second commit
617567 Initial commit

#> git push
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 320 bytes | 320.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To /home/user/my_repo/
 88a843d..7d385a4  main -> main

#> git log --oneline
ce2f041 (HEAD -> main, origin/main, origin/HEAD) Adding first author
dbb0042 Second commit
848679e Initial commit
```



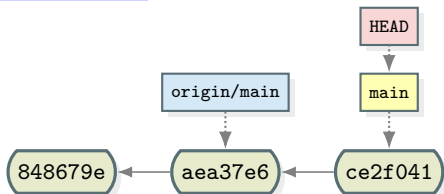
Example: 'pull --rebase' (project2)

```
#> git pull --rebase
First, rewinding head to replay your work on top of it...
Applying: Adding first author

#> git log --oneline
ce2f041 (HEAD -> main) Adding first author
88a843d (origin/main) Second commit
617567 Initial commit

#> git push
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 320 bytes | 320.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To /home/user/my_repo/
 88a843d..7d385a4  main -> main

#> git log --oneline
ce2f041 (HEAD -> main, origin/main, origin/HEAD) Adding first author
dbb0042 Second commit
848679e Initial commit
```



The `'rebase'` option push all the new modifications on the top of the history of what you download from the remote repository. It minimize a lot the possibility of a conflict.

Note, also that the hash of the 'Adding first author' commit changed after the rebase!

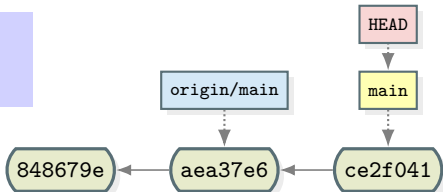
Example: 'pull --rebase' (project2)

```
#> git pull --rebase
First, rewinding head to replay your work on top of it...
Applying: Adding first author

#> git log --oneline
ce2f041 (HEAD -> main) Adding first author
88a843d (origin/main) Second commit
617567 Initial commit

#> git push
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 320 bytes | 320.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To /home/user/my_repo/
 88a843d..7d385a4  main -> main

#> git log --oneline
ce2f041 (HEAD -> main, origin/main, origin/HEAD) Adding first author
dbb0042 Second commit
848679e Initial commit
```



The `'rebase'` option push all the new modifications on the top of the history of what you download from the remote repository. It minimize a lot the possibility of a conflict.

Note, also that the hash of the 'Adding first author' commit changed after the rebase!

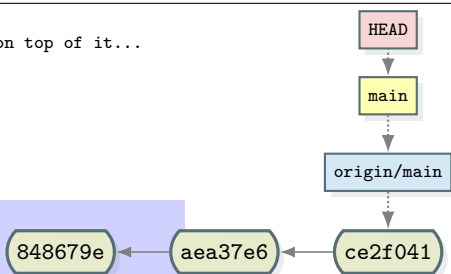
Example: 'pull --rebase' (project2)

```
#> git pull --rebase
First, rewinding head to replay your work on top of it...
Applying: Adding first author
```

```
#> git log --oneline
ce2f041 (HEAD -> main) Adding first author
88a843d (origin/main) Second commit
617567 Initial commit
```

```
#> git push
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 320 bytes | 320.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To /home/user/my_repo/
 88a843d..7d385a4  main -> main
```

```
#> git log --oneline
ce2f041 (HEAD -> main, origin/main, origin/HEAD) Adding first author
dbb0042 Second commit
848679e Initial commit
```



The `'rebase'` option push all the new modifications on the top of the history of what you download from the remote repository. It minimize a lot the possibility of a conflict.

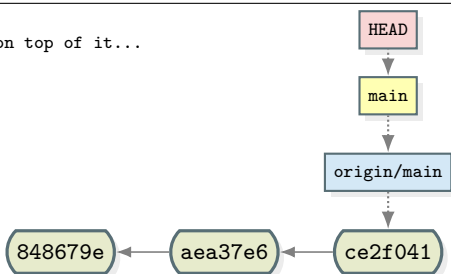
Note, also that the hash of the 'Adding first author' commit changed after the rebase!

Example: 'pull --rebase' (project2)

```
#> git pull --rebase
First, rewinding head to replay your work on top of it...
Applying: Adding first author

#> git log --oneline
ce2f041 (HEAD -> main) Adding first author
88a843d (origin/main) Second commit
617567 Initial commit

#> git push
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 320 bytes | 320.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To /home/user/my_repo/
 88a843d..7d385a4  main -> main
```



```
#> git log --oneline
ce2f041 (HEAD -> main, origin/main, origin/HEAD) Adding first author
dbb0042 Second commit
848679e Initial commit
```

The **'rebase'** option push all the new modifications on the top of the history of what you download from the remote repository. It minimize a lot the possibility of a conflict.

Note, also that the hash of the 'Adding first author' commit changed after the rebase!

1 Introduction

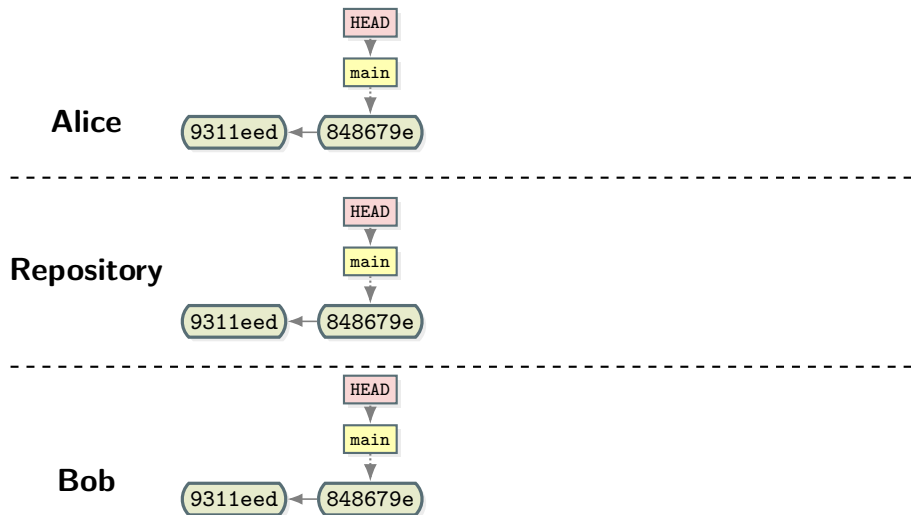
2 Basic Usages

- Command Syntax
- Configuration
- Handling the Index
- Dealing with History
- Using Branches
- Getting a Repository
- Synchronize with Remote
- Solving Conflicts
- Managing Remotes & Other Useful Commands

3 Development Workflows

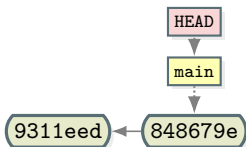
- Centralized Workflow
- Feature Branch Workflow
- Forking Workflow
- Massively Distributed Workflow
- Best Practices

What is a Conflict?

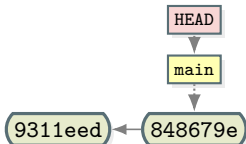


What is a Conflict?

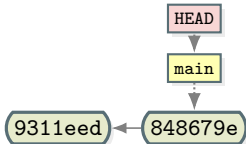
Alice



Repository

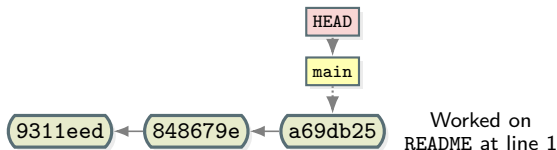


Bob

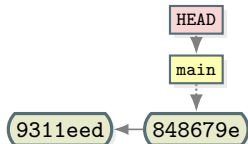


What is a Conflict?

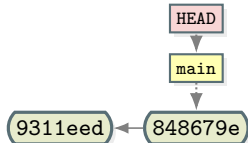
Alice



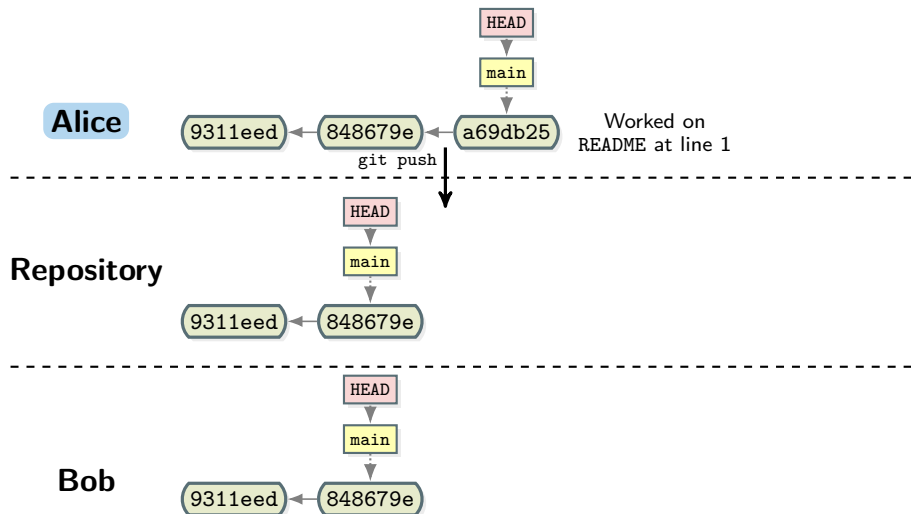
Repository



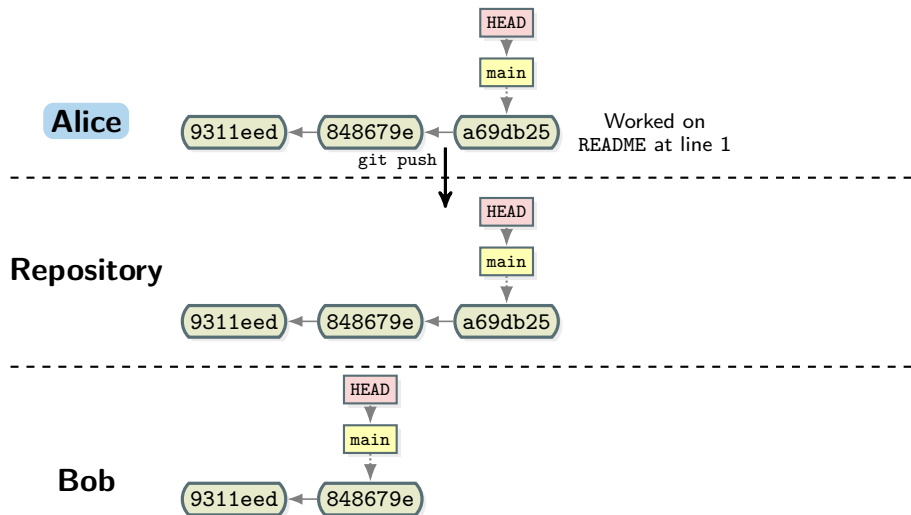
Bob



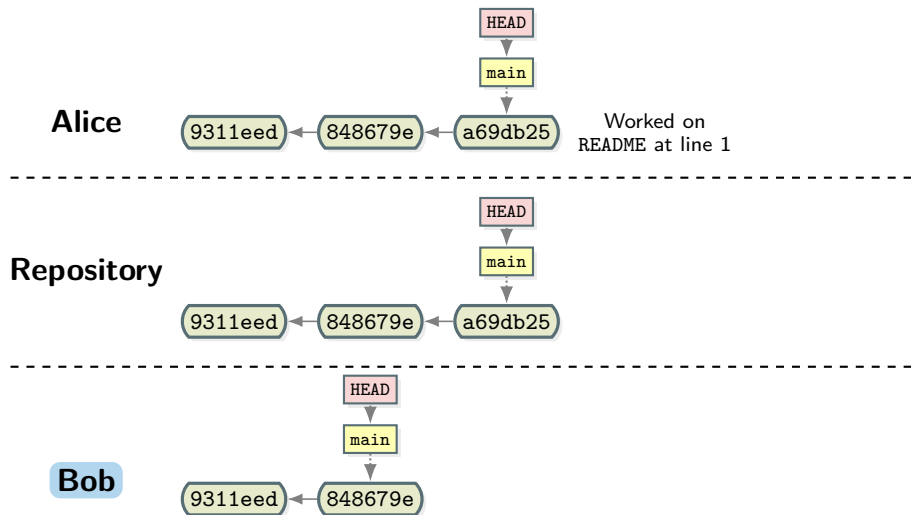
What is a Conflict?



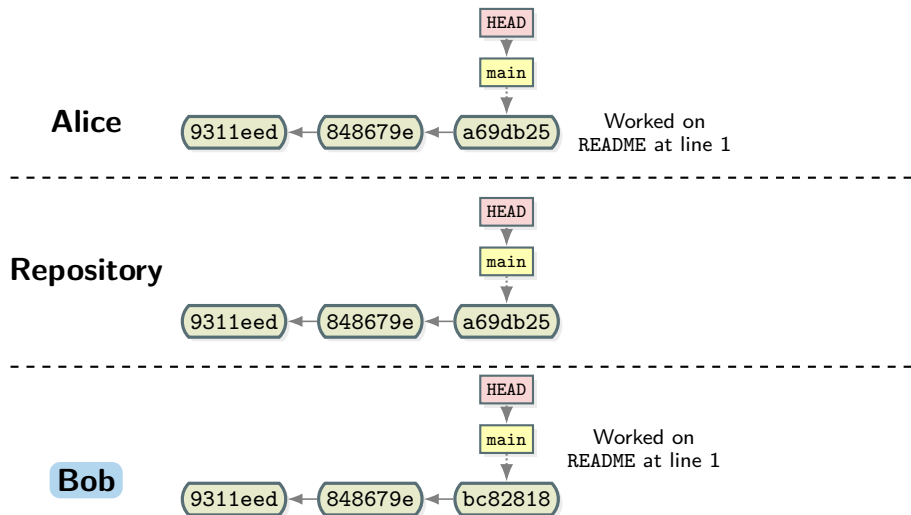
What is a Conflict?



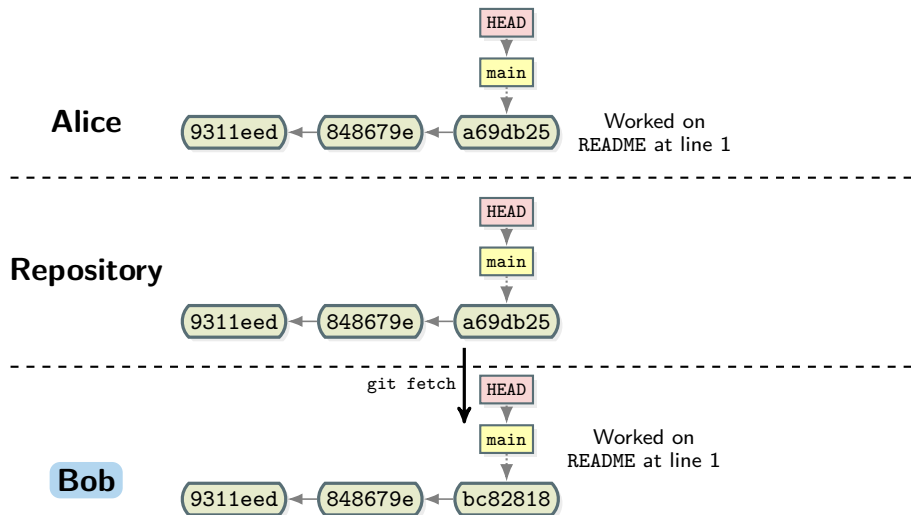
What is a Conflict?



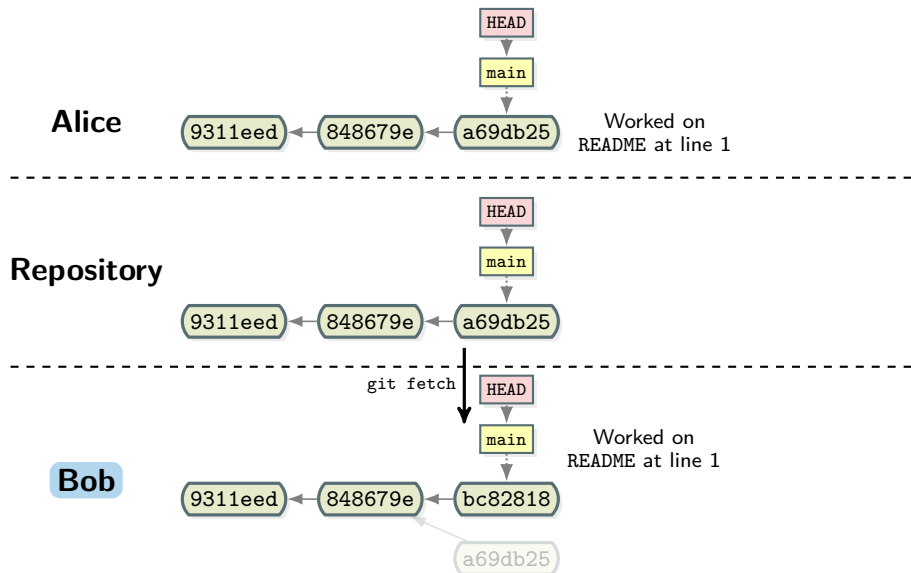
What is a Conflict?



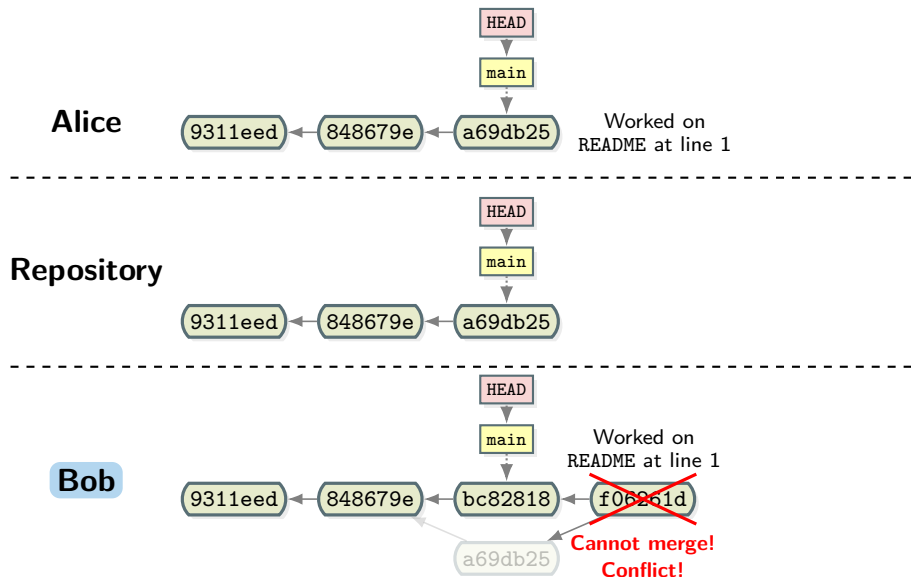
What is a Conflict?



What is a Conflict?



What is a Conflict?



Making a Change on README in project1

```
echo "My change1" >> README
#> git commit -a -m "Added a line to the README file"
[main 26caa4e] Added a line to the README file
 1 file changed, 1 insertion(+)
#> git push
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 8 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 334 bytes | 334.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To /home/user/my_repo/
 0d4a3a1..26caa4e  main -> main
```

Making a Change on README in project2

```
#> echo "My change2" >> README
#> git commit -a -m "Added a line to the README file too"
[main 3e0eb0f] Added a line to the README file too
 1 file changed, 1 insertion(+)
```

Still in project2

```
#> git pull
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From /home/user/my_repo
  Od4a3a1..26caa4e  main      -> origin/main
Auto-merging README
CONFLICT (content): Merge conflict in README
Automatic merge failed; fix conflicts and then commit the result.

#> git status
Your branch and 'origin/main' have diverged,
and have 2 and 1 different commits each, respectively.
  (use "git pull" to merge the remote branch into yours)

You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Unmerged paths:
  (use "git add <file>..." to mark resolution)
    both modified:   README
no changes added to commit (use "git add" and/or "git commit -a")
```

Now, we have a conflict on README in project2!

Solving the conflict

```
#> git diff
diff --cc README
index 789f05d,3ae2ab8..0000000
--- a/README
+++ b/README
@@@ -1,3 -1,3 +1,7 @@@
    Hello World!
    Second line
++<<<<<< HEAD
+My change1
+=====
+ My change2
+>>>>>> Added a line to the README file too

#> nano README

#> git diff
diff --cc README
index 789f05d,3ae2ab8..0000000
--- a/README
+++ b/README
@@@ -1,3 -1,3 +1,4 @@@
    Hello World!
    Second line
+My change1
+ My change2
```

Commit and Push

```
#> git add README
#> git commit
[main 536aa05] Merge branch 'main' of /home/user/my_repo

#> git push
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 8 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 342 bytes | 342.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To /home/user/my_repo/
 26caa4e..baf2f20  main -> main
```

Conflict is solved, committed and pushed to remote!

1 Introduction

2 Basic Usages

- Command Syntax
- Configuration
- Handling the Index
- Dealing with History
- Using Branches
- Getting a Repository
- Synchronize with Remote
- Solving Conflicts
- **Managing Remotes & Other Useful Commands**

3 Development Workflows

- Centralized Workflow
- Feature Branch Workflow
- Forking Workflow
- Massively Distributed Workflow
- Best Practices

Listing all Remotes

```
#> git remote
origin
#> git remote --verbose
origin    /home/user/project/ (fetch)
origin    /home/user/project/ (push)
```

Add/Rename/Remove a Remote

```
#> git remote add wip git@git.server.org:wip/project.git
#> git remote rename wip wip-new
#> git remote remove wip-new
```

Get Information about a Remote

```
#> git remote show origin
* remote origin
  Fetch URL: /home/user/project/
  Push  URL: /home/user/project/
  HEAD branch: main
  Remote branches:
    main tracked
  Local branches configured for 'git pull':
    main merges with remote main
  Local refs configured for 'git push':
    main pushes to main (up to date)
```


Verify your Work

```
#> git status
#> git log --oneline --graph --all
```

Put unfinished work aside

```
#> git stash
#> git stash list
#> git stash pop
#> git stash drop
```

Merge last updates first

```
#> git fetch
#> git pull --rebase
... solve possible conflict ...
#> git commit
```

Finally push

```
#> git push
```

1 Introduction

2 Basic Usages

- Command Syntax
- Configuration
- Handling the Index
- Dealing with History
- Using Branches
- Getting a Repository
- Synchronize with Remote
- Solving Conflicts
- Managing Remotes & Other Useful Commands

3 Development Workflows

- Centralized Workflow
- Feature Branch Workflow
- Forking Workflow
- Massively Distributed Workflow
- Best Practices

A **Workflow** is a set of rules that settle the development process in a project. It is mainly used to avoid problems such as:

- **Loss of data;**
- **Bugs in the code;**
- **Conflicts in commits;**
- **Useless code history;**
- ...

There exists several workflows, each one with its own advantages and drawbacks. In the following we will take a look at the following ones:

- 1 **Centralized Workflow;**
- 2 **Feature Branch Workflow;**
- 3 **Forking Workflow.**

1 Introduction

2 Basic Usages

- Command Syntax
- Configuration
- Handling the Index
- Dealing with History
- Using Branches
- Getting a Repository
- Synchronize with Remote
- Solving Conflicts
- Managing Remotes & Other Useful Commands

3 Development Workflows

- Centralized Workflow
- Feature Branch Workflow
- Forking Workflow
- Massively Distributed Workflow
- Best Practices

The **Centralized Workflow** is used in small teams and beginner developers.

It uses a **central repository** to serve as a synchronization point for the team and a **single branch**. Each developer pushes its changes to it in no particular order. Developers have constantly to ensure that their commits are compatible with the work of the others and that no break occurs.



Benefits

- No need to have a complex git server;
- Simple to understand and use;
- Easy to manage.
(developers themselves are in charge of it)

Drawbacks

- Risk of conflicts are extremely high;
- History is not readable;
- No parallel development;
- No code review.

Get Repository

```
#> git clone <url>
```

Pull

```
#> git pull --rebase  
#> make test
```

Push

```
#> git pull --rebase  
#> make test  
#> git push
```

Commit

```
#> git add <file>  
#> git status  
#> git diff  
#> make test  
#> git commit -m "Meaningful message"
```

Conflict

```
#> git status  
#> edit <files>  
#> git add <files>  
#> make test  
#> git commit -m "Merging with main"
```

In this workflow, the **main** branch must **always** be maintained in a **stable** state. (it must compile and pass all the tests after each push)

Note: The '--rebase' option can be enforced globally with:

```
#> git config --global pull.rebase true
```

1 Introduction

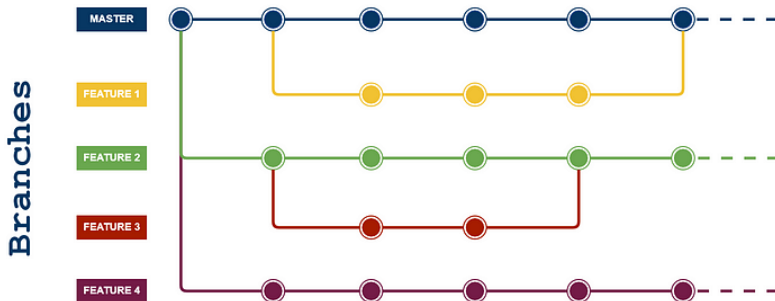
2 Basic Usages

- Command Syntax
- Configuration
- Handling the Index
- Dealing with History
- Using Branches
- Getting a Repository
- Synchronize with Remote
- Solving Conflicts
- Managing Remotes & Other Useful Commands

3 Development Workflows

- Centralized Workflow
- **Feature Branch Workflow**
- Forking Workflow
- Massively Distributed Workflow
- Best Practices

The **Feature Branch Workflow** is used to render the history more readable and allow parallel development. It uses a **single remote repository** and **multiple branches**. Each developer creates a new branch for each feature and merge it back to the main branch when finished. Yet, this is not appropriate for large teams and requires a lot of discipline.



Benefits

- No need to have a complex git server;
- Parallel development is possible;
- History is much more readable;

Drawbacks

- Risk of unmergeable branches is high;
- Project management is more complex;
- No code review.

Get Repository

```
#> git clone <url>
```

Merge a branch

```
#> git switch main  
#> git merge <branch>  
#> git push
```

Conflicts

Conflicts mainly occurs when merging a branch to **main**.

In this workflow, each branch must be dedicated to a **single feature** and **must be merged as soon as possible to main** in order to avoid conflicts. If it takes too long, think about **rebasing your branch on main** from time to time. Also, try to keep as less as possible unmerged branches otherwise history will be unreadable.

Note: When deleting a local branch, the '-d' option can be blocked by local uncommitted modifications, to enforced deletion, use '-D' instead.

Create a branch

```
#> git checkout -b <branch>  
#> edit <files>  
#> git add <files>  
#> git commit -m "Meaningful message"  
#> git push --set-upstream origin <branch>
```

Delete a branch

```
#> git switch main  
#> git branch -d <branch>  
#> git push origin --delete <branch>
```

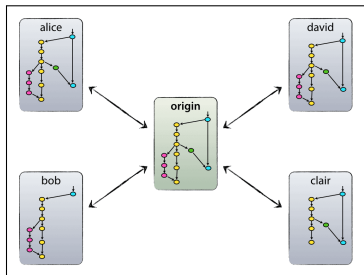
1 Introduction

2 Basic Usages

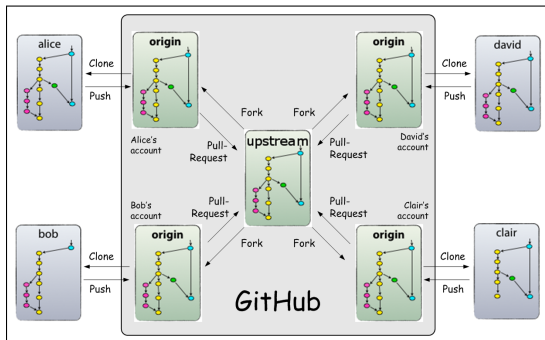
- Command Syntax
- Configuration
- Handling the Index
- Dealing with History
- Using Branches
- Getting a Repository
- Synchronize with Remote
- Solving Conflicts
- Managing Remotes & Other Useful Commands

3 Development Workflows

- Centralized Workflow
- Feature Branch Workflow
- **Forking Workflow**
- Massively Distributed Workflow
- Best Practices



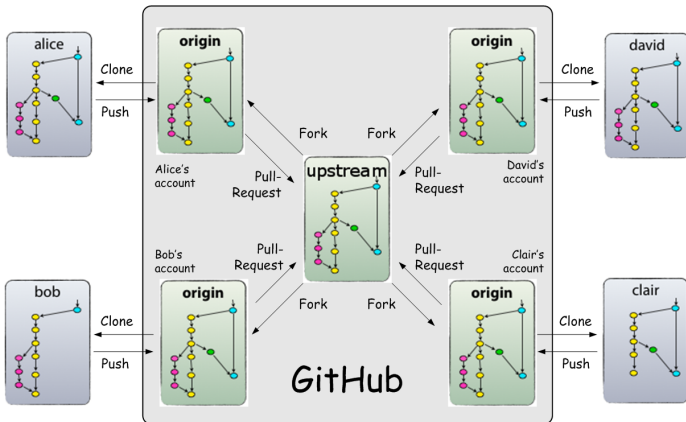
Centralized Workflow



Forking Workflow

The **Forking Workflow** requires an extra layer of functionalities on top of git and different roles among the team, for these it uses a **git forge server**. This workflow is **more complex** but can be used in **large teams** and includes a **peer review process**.

Each **developer** creates a **fork** of the central repository and work on it. Once finished, he creates a **pull request** to the central repository. The **maintainer** can then **review** the code and discuss with the developer. Once ready, the code is **merged** in the upstream.



Benefits

- Parallel development is easy;
- History is much more readable;
- Code review is part of the process.

Drawbacks

- Requires a git forge server;
- Project managers are the bottleneck;
- A lot of code reviews to perform.

A **Software Forge** is a web-based platform that provides a set of tools to manage a software project. It is mainly used to manage the development of a project using a `git` repository. It provides a lot of functionalities such as:

- **Project management tools** (create, delete, fork, ...);
- **Users and Groups management** (set rights, ...);
- **Issue tracking** (bugs and features);
- **Code review tools** (pull requests, merge requests, ...);
- **Continuous Integration** (build, test, runners, containers, ...);
- ...

These software forges provides high-level functionalities such as **forking** and **pull requests** to manage the development of a project. The most famous forges using `git` are **Gitlab**, **GitHub**, **BitBucket**. **Gitlab** is the only one to be Open Source and available at the CREMI.

Fork a Project (developer) (1/2)

Step 1: Fork the upstream

The screenshot shows the GitLab interface for a repository named 'Trial Project' by Emmanuel Fleury. At the top, it displays the repository name, a notification bell, star count (0), and fork count (0). Below this, it shows '1 Commit', '1 Branch', '0 Tags', and '3 KiB Project Storage'. The main content area features an 'Initial commit' by Emmanuel Fleury, with a commit hash 'c0e43ccc'. A breadcrumb trail shows 'main > trial-project / +'. Below the breadcrumb are several action buttons: 'README', 'Add LICENSE', 'Add CHANGELOG', 'Add CONTRIBUTING', 'Enable Auto DevOps', 'Add Kubernetes cluster', 'Set up CI/CD', 'Add Wiki', and 'Configure Integrations'. A table lists the files in the repository:

Name	Last commit	Last update
README.md	Initial commit	just now

Below the table, the content of the 'README.md' file is displayed, starting with the title 'Trial Project' and a section 'Getting started'.

Fork a Project (developer) (1/2)

Step 1: Fork the upstream

Emmanuel Fleury / Trial Project

Trial Project

1 Commit 1 Branch 0 Tags 3 KiB Project Storage

Initial commit
Emmanuel Fleury authored just now
c0e43ccc

main trial-project / + History Find file Edit Code

README Add LICENSE Add CHANGELOG Add CONTRIBUTING

+ Enable Auto DevOps Add Kubernetes cluster + Set up CI/CD Add Wiki

Configure Integrations

Name	Last commit	Last update
README.md	Initial commit	just now

README.md

Trial Project

Getting started

To make it easy for you to get started with GitLab, here's a list of recommended next steps.

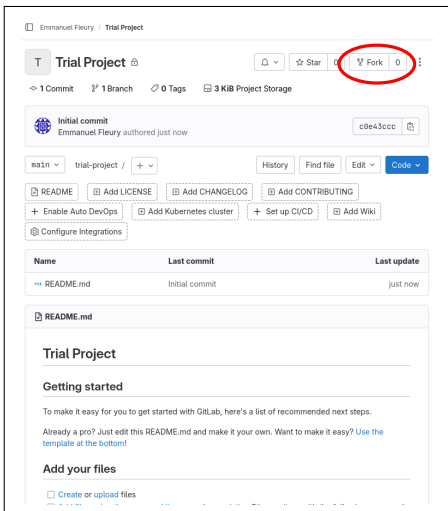
Already a pro? Just edit this README.md and make it your own. Want to make it easy? [Use the template at the bottom!](#)

Add your files

Create or upload files

Fork a Project (developer) (1/2)

Step 1: Fork the upstream



Emmanuel Fleury / Trial Project

Trial Project

1 Commit 1 Branch 0 Tags 3 KiB Project Storage

Initial commit
Emmanuel Fleury authored just now

main trial-project / +

History Find file Edit Code

README Add LICENSE Add CHANGELOG Add CONTRIBUTING

+ Enable Auto DevOps + Add Kubernetes cluster + Set up CI/CD Add Wiki

Configure Integrations

Name	Last commit	Last update
README.md	Initial commit	just now

README.md

Trial Project

Getting started

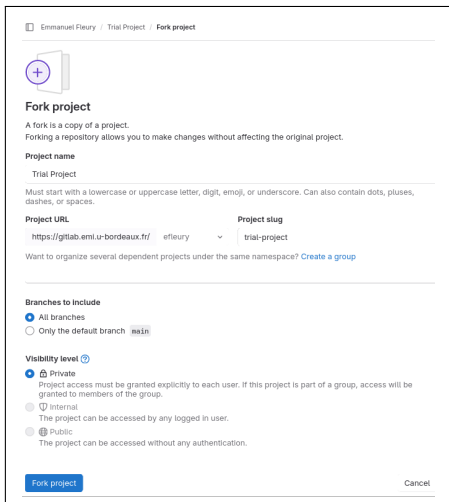
To make it easy for you to get started with GitLab, here's a list of recommended next steps.

Already a pro? Just edit this README.md and make it your own. Want to make it easy? [Use the template at the bottom!](#)

Add your files

Create or upload files

Step 2: Fill the form



Emmanuel Fleury / Trial Project / Fork project

Fork project

A fork is a copy of a project.
Forking a repository allows you to make changes without affecting the original project.

Project name
Trial Project

Must start with a lowercase or uppercase letter, digit, emoji, or underscore. Can also contain dots, pluses, dashes, or spaces.

Project URL
https://gitlab.eml.u-bordeaux.fr/efleury

Project slug
trial-project

Want to organize several dependent projects under the same namespace? [Create a group](#)

Branches to include

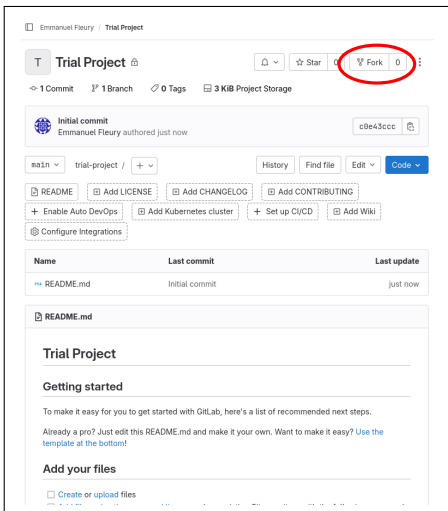
- All branches
- Only the default branch `main`

Visibility level

- Private**
Project access must be granted explicitly to each user. If this project is part of a group, access will be granted to members of the group.
- Internal
The project can be accessed by any logged in user.
- Public
The project can be accessed without any authentication.

Fork a Project (developer) (1/2)

Step 1: Fork the upstream



Emmanuel Fleury / Trial Project

Trial Project

1 Commit 1 Branch 0 Tags 3 KiB Project Storage

Initial commit
Emmanuel Fleury authored just now

main trial-project / +

History Find file Edit Code

README Add LICENSE Add CHANGELOG Add CONTRIBUTING

+ Enable Auto DevOps Add Kubernetes cluster + Set up CI/CD Add Wiki

Configure Integrations

Name	Last commit	Last update
README.md	Initial commit	just now

README.md

Trial Project

Getting started

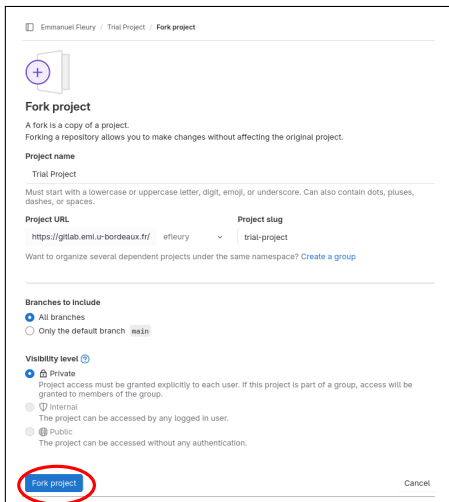
To make it easy for you to get started with GitLab, here's a list of recommended next steps.

Already a pro? Just edit this README.md and make it your own. Want to make it easy? [Use the template at the bottom!](#)

Add your files

Create or upload files

Step 2: Fill the form



Emmanuel Fleury / Trial Project / Fork project

Fork project

A fork is a copy of a project.
Forking a repository allows you to make changes without affecting the original project.

Project name
Trial Project

Must start with a lowercase or uppercase letter, digit, emoji, or underscore. Can also contain dots, pluses, dashes, or spaces.

Project URL
https://gitlab.eml.u-bordeaux.fr/efleury

Project slug
trial-project

Want to organize several dependent projects under the same namespace? [Create a group](#)

Branches to include

- All branches
- Only the default branch `main`

Visibility level

- Private
Project access must be granted explicitly to each user. If this project is part of a group, access will be granted to members of the group.
- Internal
The project can be accessed by any logged in user.
- Public
The project can be accessed without any authentication.

Step 3: Clone forked project locally

```
#> git clone gitlab.emi.u-bordeaux.fr:developer/trialproject.git
Cloning into 'trialproject'...
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
```

Step 4: Add upstream remote

```
#> cd trialproject/
#> git remote add upstream gitlab.emi.u-bordeaux.fr:maintainer/trialproject.git
#> git remote -v
origin      gitlab.emi.u-bordeaux.fr:developer/trialproject.git (fetch)
origin      gitlab.emi.u-bordeaux.fr:developer/trialproject.git (push)
upstream    gitlab.emi.u-bordeaux.fr:maintainer/trialproject.git (fetch)
upstream    gitlab.emi.u-bordeaux.fr:maintainer/trialproject.git (push)
```

Step 5: Keep up with upstream

```
#> git fetch upstream main
#> git switch main
#> git merge upstream/main
#> git push
```

Step 6: Rebase a feature on main

```
#> git switch my_branch
#> git rebase main
#> git push --force
```

Step 1: Create a new branch

```
#> git switch -c new_feature
```

Step 2: Work on it

```
#> edit <files>
#> git add <files>
#> git commit -m "Message"
```

Step 3: Push it

```
#> git push --set-upstream origin new_feature
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
...
remote:To create a merge request for new_feature, visit:
remote:  https://gitlab.emi.u-bordeaux.fr/...

To gitlab.emi.u-bordeaux.fr/trial-project-forked.git
 * [new branch]      new_feature -> new_feature
branch 'new_feature' set up to track 'origin/new_feature'
```

Step 1: Create a new branch

```
#> git switch -c new_feature
```

Step 2: Work on it

```
#> edit <files>
#> git add <files>
#> git commit -m "Message"
```

Step 3: Push it

```
#> git push --set-upstream origin new_feature
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
...
remote: To create a merge request for new feature, visit:
remote: https://gitlab.emi.u-bordeaux.fr/...

To gitlab.emi.u-bordeaux.fr/trial-project-forked.git
 * [new branch]      new_feature -> new_feature
branch 'new_feature' set up to track 'origin/new_feature'
```

Step 1: Create a new branch

```
#> git switch -c new_feature
```

Step 2: Work on it

```
#> edit <files>
#> git add <files>
#> git commit -m "Message"
```

Step 3: Push it

```
#> git push --set-upstream origin new_feature
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
...
remote: To create a merge request for new feature, visit:
remote: https://gitlab.emi.u-bordeaux.fr/...

To gitlab.emi.u-bordeaux.fr/trial-project-forked.git
 * [new branch]      new_feature -> new_feature
branch 'new_feature' set up to track 'origin/new_feature'
```

Step 4: Create merge request

Emmanuel Fleury / Trial Project Forked / Merge requests / New

New merge request

From `efleury/trial-project-forked:new_feature` into `efleury/trial-project:main`
[Change branches](#)

Title (required)

Mark as draft
Drafts cannot be merged until marked ready.

Description

B *I* U |

Switch to rich text editing

Add [description templates](#) to help your contributors to communicate effectively!

Assignee

[Assign to me](#)

Reviewer

Milestone

Get Repository

```
## Fork the project on the forge ##  
#> git clone <forked-project>  
#> git remote add upstream <project>
```

Pull upstream

```
#> git fetch upstream main  
#> git switch main  
#> git merge upstream/main  
#> git push
```

In this workflow, you have to handle two remotes: **origin** and **upstream**. The **origin** remote is your fork of the project and the **upstream** remote is the original project. You have to **pull** from **upstream** (read-only) and **push** to **origin** (read-write). The **upstream** is only written by the maintainer when **merging** your pull requests.

Note: The maintainers are in charge of reviewing all the code. Strictly speaking, they should not push anything themselves without a merge request. That is why they should be at least two to avoid to review their own code. Also, you have to understand that reviewing code is a real skill and is time consuming (and extremely frustrating).

Rebase a branch on main

```
#> git switch <branch>  
#> git rebase main  
#> git push --force
```

Create a pull request

```
#> git push  
## Pull request on the forge ##
```

1 Introduction

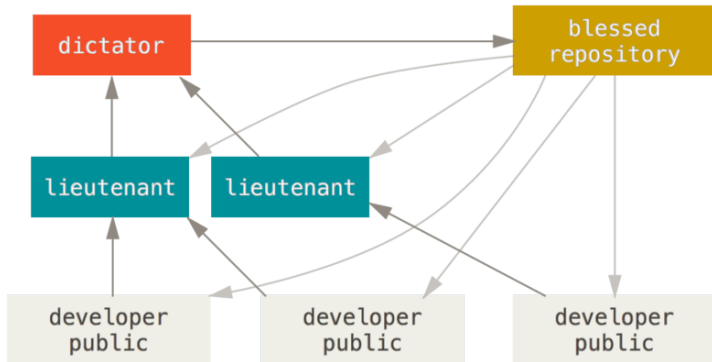
2 Basic Usages

- Command Syntax
- Configuration
- Handling the Index
- Dealing with History
- Using Branches
- Getting a Repository
- Synchronize with Remote
- Solving Conflicts
- Managing Remotes & Other Useful Commands

3 Development Workflows

- Centralized Workflow
- Feature Branch Workflow
- Forking Workflow
- **Massively Distributed Workflow**
- Best Practices

The Linux Kernel Workflow was designed by Linus Torvalds himself to manage the development of the Linux Kernel. It is a **massively distributed workflow** that uses a **benevolent dictator** to manage the project. It is used in large projects with a lot of developers and a lot of sub-projects. Somehow, it generalizes the **forking workflow** to a larger scale with many remotes to handle.



1 Introduction

2 Basic Usages

- Command Syntax
- Configuration
- Handling the Index
- Dealing with History
- Using Branches
- Getting a Repository
- Synchronize with Remote
- Solving Conflicts
- Managing Remotes & Other Useful Commands

3 Development Workflows

- Centralized Workflow
- Feature Branch Workflow
- Forking Workflow
- Massively Distributed Workflow
- Best Practices

- 1 **Don't panic!**
- 2 **Don't overestimate your skills!**
- 3 **Keep control of your repository!**
- 4 **Commit early and commit often!**
 - Always display 'git status' and 'git diff' before a commit.
 - Do not forget to push your work to the server.
- 5 **Your repository is not a trash bin!**
 - Do not use 'git add *'.
 - Do not track files produced by code!
 - Do not commit code that is not part of the project (use git submodules).
- 6 **Do not ignore conflicts! Resolve it!**
- 7 **Merge branches as soon as possible!**

- 1 Know, practice and read about git!
- 2 On doubt, back-up your repository before trying!
- 3 Avoid conflicts at any price!
- 4 Commit early and commit often!
- 5 One commit for one thing, no more!
- 6 Be meaningful on your commit messages!
- 7 Branch as much as possible!
- 8 Merge as soon as possible!
- 9 Try to conform the current workflow!
- 10 Keep up with upstream and the rest of the team!

- 1 Work mainly on `main`;
- 2 You commit your changes as you wish to local or central repository;
- 3 Once you finished your homework (or project), you push everything to `main` (if you were using another branch);
- 4 You create a branch 'homework-N' (or 'project');

```
#> git checkout -b homework-3
```

- 5 You push it upstream on the central repository with:

```
#> git push --set-upstream origin homework-3
```

- 6 You start the next homework/project.
- 7 If you notice a bug in your homework, just switch to the branch and push a fix on it.

- **Git Best Practises**, by Wolfgang Dobler, 2016.
<http://pencil-code.nordita.org/doc/git-best-practises.pdf>
- **Git Tutorial**, by Lars Vogel, 2009-2024.
<http://www.vogella.com/tutorials/Git/article.html>
- **A Visual Git Reference**, by Marko Lodato, 2010-2017.
<https://marklodato.github.io/visual-git-guide/>
- **Learn Git Branching** (Interactive Tutorial), by Peter Cottle, 2012-2024.
<https://learngitbranching.js.org/>
- **Visualizing Git** (Visualization Tool), by Katrina Uychaco, 2019.
<http://git-school.github.io/visualizing-git/>
- **Git Flight Rules**, by Kate Hudson, 2023.
<https://github.com/k88hudson/git-flight-rules>
- **L^AT_EX gitdags package**, by Julien Cretel, 2014.
<https://github.com/Jubobs/gitdags/wiki>

4 Backup Slides

A few commands useful to clean-up and compress your objects data-base in your repository:

```
# Verifies connectivity and validity of objects in repository  
git fsck --full  
  
# Manage reflog information and set it to 'expire'  
git reflog expire --expire=now --all  
  
# Pack unpacked objects in a repository  
git repack -adl  
  
# Cleanup unnecessary files and optimize local repository and  
# prune all unreachable objects from the object database  
git gc --prune=now --aggressive
```

You should have enabled the `FormatOnSave` option in Visual Studio Code and use an automatic code formatter. But, if you have set `FormatOnSaveMode` to `file`, you should seriously consider switch it to `modificationsifAvailable`. Because it might render your commits unreadable and impossible to review.