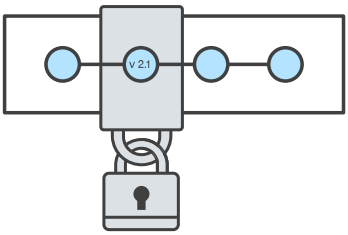
IT UND SOFTWARE PROJEKTE

SKRITUM

TABLE OF CONTENTS

WHAT IS VERSION CONTROL?



Version control, also known as source control, is the practice of tracking and managing changes to software code. Version control systems are software tools that help software teams manage changes to source code over time.

Version control software keeps track of every modification to the code in a special kind of database. If a mistake is made, developers can turn back the clock and compare earlier versions of the code to help fix the mistake while minimizing disruption to all team members.

Software developers working in teams are continually writing new source code and changing existing source code. The code for a project, app or software component is typically organized in a folder structure or "file tree". One developer on the team may be working on a new feature while another developer fixes an unrelated bug by changing code, each developer may make their changes in several parts of the file tree.

Software teams that do not use any form of version control often run into problems like not knowing which changes that have been made are available to users or the creation of incompatible changes between two unrelated pieces of work that must then be painstakingly untangled and reworked. If you're a developer who has never used version control you may have added versions to your files, perhaps with suffixes like "final" or "latest" and then had to later deal with a new final version. Perhaps you've commented out code blocks because you want to disable certain functionality without deleting the code, fearing that there may be a use for it later. Version control is a way out of these problems.

Version control software is an essential part of the every-day of the modern software team's professional practices. Individual software developers who are accustomed to working with a capable version control system in their teams typically recognize the incredible value version control also gives them even on small solo projects. Once accustomed to the powerful benefits of version control systems, many developers wouldn't consider working without it even for non-software projects.

BENEFITS

1. **A complete long-term change history of every file.** This means every change made by many individuals over the years. Changes include the creation and deletion of files as well as edits to their contents.
2. **Branching and merging.** Having team members work concurrently is a no-brainer, but even individuals working on their own can benefit from the ability to work on independent streams of changes. Creating a "branch" in VCS tools keeps multiple streams of work independent from each other while also providing the facility to merge that work back together, enabling developers to verify that the changes on each branch do not conflict.
3. **Traceability.** Being able to trace each change made to the software and connect it to project management and bug tracking, and being able to annotate each change with a message describing the purpose and intent of the change can help not only with root cause analysis and other forensics.

SOURCE CODE MANAGEMENT

Source code management (SCM) is used to track modifications to a source code repository. SCM tracks a running history of changes to a code base and helps resolve conflicts when merging updates from multiple contributors.

SCM == Version control

As software projects grow in lines of code and contributor head count, the costs of communication overhead and management complexity also grow. SCM is a critical tool to alleviate the organizational strain of growing development costs.

IMPORTANCE OF SCM

When multiple developers are working within a shared codebase it is a common occurrence to make edits to a shared piece of code. Separate developers may be working on a seemingly isolated feature, however this feature may use a shared code module. Therefore developer 1 working on Feature 1 could make some edits and find out later that Developer 2 working on Feature 2 has conflicting edits.

Before the adoption of SCM this was a nightmare scenario. Developers would edit text files directly and move them around to remote locations. Developer 1 would make edits and Developer 2 would unknowingly save over Developer 1's work and wipe out the changes. SCM's role as a protection mechanism against this specific scenario is known as Version Control.

Example:
Developer 1 adds sub in File strichrechnung.h
Developer 2 adds div in File punktrechnung.h
MERGE CONFLICT in main.c

main.c | strichrechnung.h | punktrechnung.h

Users > mg > Dropbox > Schule_HTL > IT_Projekte > Demo_Konflikt > main.c > main()

```c
#include "punktrechnung.h"
#include "strichrechnung.h"
#include <stdio.h>

int main() {
    int a, b, c;
    char op;
    printf("Bitte Zahl 1 eingeben: ");
    scanf("%d", &a);
    printf("Bitte Zahl 2 eingeben: ");
    scanf("%d", &b);
    printf("Bitte Operation eingeben: ");
    scanf("%c", &op);

    if (op == '+') {
        printf("Ergebnis: %d", add(a, b));
    } else if (op == '*') {
        printf("Ergebnis: %d", mult(a, b));
    } else {
        printf("UNKNOWN OPERATION!");
    }
}
```

main.c | strichrechnung.h | punktrechnung.h

Users > mg > Dropbox > Schule_HTL > IT_Projekte > Demo_Konflikt > strichrechnung.h > add(int, int)

```c
int add(int a, int b);

int add(int a, int b) {
    return a + b;
}
```

main.c | strichrechnung.h | punktrechnung.h

Users > mg > Dropbox > Schule_HTL > IT_Projekte > Demo_Konflikt > punktrechnung.h > mult(int, int)

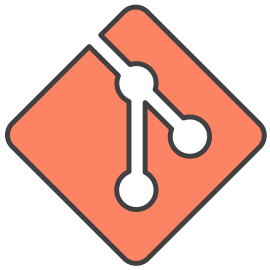
```c
int mult(int a, int b);

int mult(int a, int b) {
    return a * b;
}
```

BENEFITS

Once SCM has started tracking all the changes to a project over time, a detailed historical record of the projects life is created. This historical record can then be used to ‘undo’ changes to the codebase. The SCM can instantly revert the codebase back to a previous point in time. This is extremely valuable for preventing regressions on updates and undoing mistakes.

The SCM archive of every change over a project's life time provides valuable record keeping for a project's release version notes. A clean and maintained SCM history log can be used interchangeably as release notes.

WHAT IS GIT

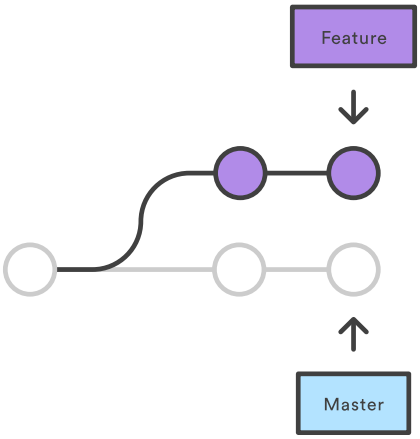


By far, the most widely used modern version control system in the world today is Git. Git is a mature, actively maintained open source project originally developed in 2005 by Linus Torvalds, the famous creator of the Linux operating system kernel.

Having a **distributed architecture**, Git is an example of a DVCS (hence Distributed Version Control System). Rather than have only one single place for the full version history of the software as is common in once-popular version control systems like CVS or Subversion (also known as SVN), in Git, every developer's working copy of the code is also a repository that can contain the full history of all changes.

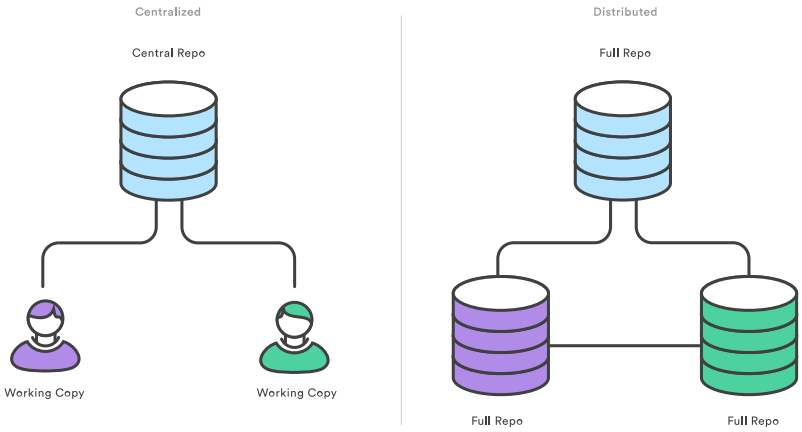
- **Performance**: The raw performance characteristics of Git are very strong when compared to many alternatives. Committing new changes, branching, merging and comparing past versions are all optimized for performance. The algorithms implemented inside Git take advantage of deep knowledge about common attributes of real source code file trees, how they are usually modified over time and what the access patterns are.
- **Security**: Git has been designed with the integrity of managed source code as a top priority. The content of the files as well as the true relationships between files and directories, versions, tags and commits, all of these objects in the Git repository are secured with a cryptographically secure hashing algorithm called SHA1.
- **Flexibility**: One of Git's key design objectives is flexibility. Git is flexible in several respects: in support for various kinds of nonlinear development workflows, in its efficiency in both small and large projects and in its compatibility with many existing systems and protocols.

BRANCHES



One of the biggest advantages of Git is its branching capabilities: Feature branches provide an isolated environment for every change to your codebase. When a developer wants to start working on something—no matter how big or small—they create a new branch. This ensures that the master branch always contains production-quality code.

DISTRIBUTED DEVELOPMENT

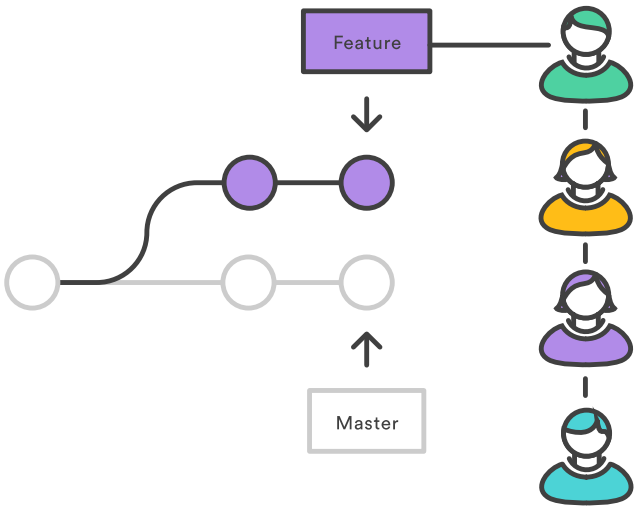


Git is a distributed version control system. Each developer gets their own local repository, complete with a full history of commits.

Having a full local history makes Git fast, since it means you don't need a network connection to create commits, inspect previous versions of a file, or perform diffs between commits.

Similar to feature branches, distributed development creates a more reliable environment. Even if a developer obliterates their own repository, they can simply clone someone else's and start anew.

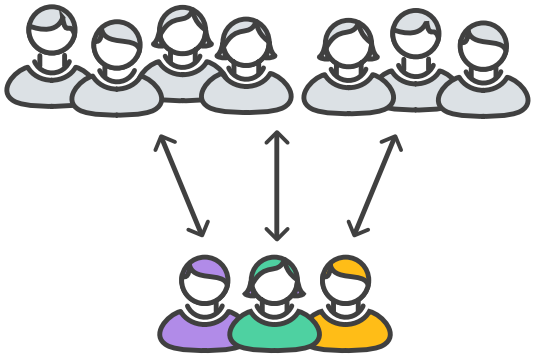
PULL / MERGE REQUESTS



Many source code management tools such as Bitbucket or Github enhance core Git functionality with pull or merge requests. A pull/merge request is a way to ask another developer to merge one of your branches into their repository. This not only makes it easier for project leads to keep track of changes, but also lets developers initiate discussions around their work before integrating it with the rest of the codebase.

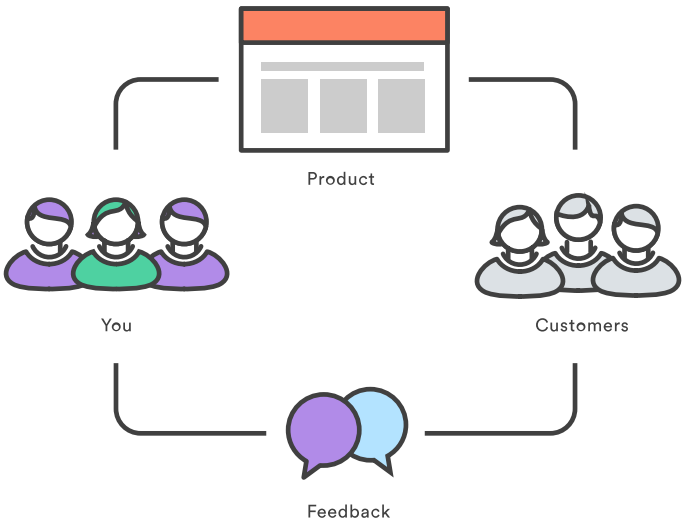
Since they're essentially a comment thread attached to a feature branch, pull requests are extremely versatile. When a developer gets stuck with a hard problem, they can open a pull request to ask for help from the rest of the team. Alternatively, junior developers can be confident that they aren't destroying the entire project by treating pull requests as a formal code review.

COMMUNITY



In many circles, Git has come to be the expected version control system for new projects. If your team is using Git, odds are you won't have to train new hires on your workflow, because they'll already be familiar with distributed development.

FASTER RELEASE CYCLE



The ultimate result of feature branches, distributed development, pull requests, and a stable community is a faster release cycle. These capabilities facilitate an agile workflow where developers are encouraged to share smaller changes more frequently. In turn, changes can get pushed down the deployment pipeline faster than the monolithic releases common with centralized version control systems.

GIT – GETTING STARTED

INSTALL GIT ON MAX OS X

There are several ways to install Git on a Mac. In fact, if you've installed XCode (or it's Command Line Tools), Git may already be installed.

Install Command Line Tools: `xcode-select --install`

To find out, open a terminal and enter `git --version`.

Alternative: https://git-scm.com/download/mac

INSTALL GIT ON WINDOWS

Download and complete wizard: http://gitforwindows.org

CONFIGURE GIT

The git config command is a convenience function that is used to set Git configuration values on a global or local project level. These configuration levels correspond to `.gitconfig` text files. Executing git config will modify a configuration text file.

USAGE

The most basic use case for `git config` is to invoke it with a configuration name, which will display the set value at that name. Configuration names are dot delimited strings composed of a 'section' and a 'key' based on their hierarchy. For example: `user.email`

```
git config user.email
```

Expanding on what we already know about git config, let's look at an example in which we write a value:

```
git config --global user.email "your_email@example.com"
```

This example writes the value `your_email@example.com` to the configuration name `user.email`. It uses the `--global` flag so this value is set for the current operating system user.

CONFIG LEVEL

The git config command can accept arguments to specify which configuration level to operate on. The following configuration levels are available:

`--local`

By default, git config will write to a local level if no configuration option is passed. Local level configuration is applied to the context repository git config gets invoked in.

`--global`

Global level configuration is user-specific, meaning it is applied to an operating system user.

`--system`

System-level configuration is applied across an entire machine.

The most important configuration is the git username and email:

```
git config --global user.name "Emma Paris"

git config --global user.email "eparis@atlassian.com"
```

CONFIGURE SSH

For Windows: use Git Bash

An **SSH** key is an access credential for the SSH (secure shell) network protocol. This authenticated and encrypted secure network protocol is used for remote communication between machines on an unsecured open network. SSH is used for remote file transfer, network management, and remote operating system access. The SSH acronym is also used to describe a set of tools used to interact with the SSH protocol.

SSH uses a pair of keys to initiate a secure handshake between remote parties. The key pair contains a public and private key. The private vs public nomenclature can be confusing as they are both called keys. It is more helpful to think of the public key as a "lock" and the private key as the "key". You give the public 'lock' to remote parties to encrypt or 'lock' data. This data is then opened with the 'private' key which you hold in a secure place.

SSH keys are generated through a public key cryptographic algorithm, the most common being RSA or DSA.

1. Execute the following to begin the key creation
   `ssh-keygen -t rsa -b 4096 -C "your_email@example.com"`
   This command will create a new SSH key using the email as a label
2. You will then be prompted to "Enter a file in which to save the key." You can specify a file location or press “Enter” to accept the default file location.
3. The next prompt will ask for a secure passphrase. A passphrase will add an additional layer of security to the SSH and will be required anytime the SSH key is used. If someone gains access to the computer that private keys are stored on, they could also gain access to any system that uses that key. Adding a passphrase to keys will prevent this scenario.
4. Add the new SSH key to the ssh-agent.
   The ssh-agent is another program that is part of the SSH toolsuite. The ssh-agent is responsible for holding private keys. Think of it like a keychain. In addition to holding private keys it also brokers requests to sign SSH requests with the private keys so that private keys are never passed around unsecurly.
   Before adding the new SSH key to the ssh-agent first ensure the ssh-agent is running by executing:
   `eval "$(ssh-agent -s)"`
   `> Agent pid 59566`
   Once the ssh-agent is running the following command will add the new SSH key to the local SSH agent.
   `ssh-add -K /Users/you/.ssh/id_rsa`
   The new SSH key is now registered and ready to use!

WORKING WITH GIT

A **Git repository** is a virtual storage of your project. It allows you to save versions of your code, which you can access when needed.

GIT INIT

Summary

To create a new repo, you'll use the `git init` command. `git init` is a one-time command you use during the initial setup of a new repo. Executing this command will create a new `.git` subdirectory in your current working directory. This will also create a new `master` branch.

Example: This example assumes you already have an existing project folder that you would like to create a repo within. You'll first `cd` to the `root` project folder and then execute the `git init` command.

```
cd /path/to/your/existing/code

git init
```

The git init command creates a new Git repository. It can be used to convert an existing, unversioned project to a Git repository or initialize a new, empty repository.

Executing git init creates a .git subdirectory in the current working directory, which contains all of the necessary Git metadata for the new repository. This metadata includes subdirectories for objects, refs, and template files. A HEAD file is also created which points to the currently checked out commit.

USAGE

All you have to do is cd into your project subdirectory and run git init, and you'll have a fully functional Git repository.

```
git init
```

Transform the current directory into a Git repository. This adds a .git subdirectory to the current directory and makes it possible to start recording revisions of the project.

EXAMPLE

Create a new git repository for an existing code base:

```
cd /path/to/code
git init
git add .
git commit
```

GIT CLONE

Summary

If a project has already been set up in a central repository, the clone command is the most common way for users to obtain a local development clone. Like `git init`, cloning is generally a one-time operation. Once a developer has obtained a working copy, all version control operations are managed through their local repository.

```
git clone <repo url>
```

`git clone` is used to create a copy or clone of remote repositories. You pass git clone a repository URL. Git supports a few different network protocols and corresponding URL formats. In this example, we'll be using the Git SSH protocol. Git SSH URLs follow a template of: `git@HOSTNAME:USERNAME/REPONAME.git`

An example Git SSH URL would be: `git@bitbucket.org:rhyolight/javascript-data-store.git` where the template values match:

`HOSTNAME: bitbucket.org`

`USERNAME: rhyolight`

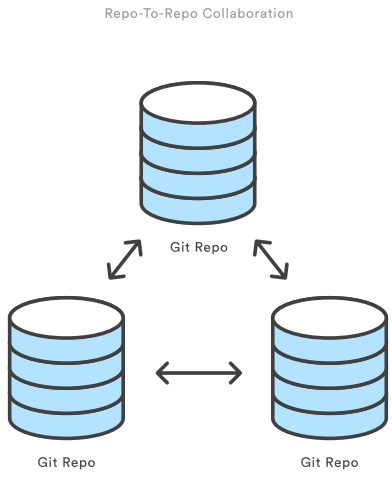
`REPONAME: javascript-data-store`

When executed, the latest version of the remote repo files on the master branch will be pulled down and added to a new folder. The new folder will be named after the `REPONAME` in this case `javascript-data-store`. The folder will contain the full history of the remote repository and a newly created master branch.

A quick note: `git init` and `git clone` can be easily confused. At a high level, they can both be used to "initialize a new git repository." However, `git clone` is dependent on `git init`. `git clone` is used to create a copy of an existing repository. Internally, `git clone` first calls `git init` to create a new repository. It then copies the data from the existing repository, and checks out a new set of working files.

`git clone` is a Git command line utility which is used to target an existing repository and create a clone, or copy of the target repository.

If a project has already been set up in a central repository, the git clone command is the most common way for users to obtain a development copy. Like git init, cloning is generally a one-time operation. Once a developer has obtained a working copy, all version control operations and collaborations are managed through their local repository.



USAGE

git clone is primarily used to point to an existing repo and make a clone or copy of that repo at in a new directory, at another location. The original repository can be located on the local filesystem or on remote machine accessible supported protocols. The git clone command copies an existing Git repository.

As a convenience, cloning automatically creates a remote connection called "origin" pointing back to the original repository. This makes it very easy to interact with a central repository.

The example below demonstrates how to obtain a local copy of a central repository stored on a server accessible at example.com using the SSH username john:

```
git clone ssh://john@example.com/path/to/my-project.git
cd my-project
```

Cloning to a specific folder

```
git clone <repo> <directory>
```

Clone the repository located at ＜repo＞ into the folder called ~＜directory＞! on the local machine.

Cloning a specific tag

```
git clone --branch <tag> <repo>
```

Clone the repository located at ＜repo＞ and only clone the ref for ＜tag＞.

Shallow clone

```
git clone -depth=1 <repo>
```

Clone the repository located at ＜repo＞ and only clone the history of commits specified by the option depth=1. In this example a clone of ＜repo＞ is made and only the most recent commit is included in the new cloned Repo. Shallow cloning is most useful when working with repos that have an extensive commit history. An extensive commit history may cause scaling problems such as disk space usage limits and long wait times when cloning. A Shallow clone can help alleviate these scaling issues.

git clone -branch

The -branch argument lets you specify a specific branch to clone instead of the branch the remote HEAD is pointing to, usually the master branch. In addition you can pass a tag instead of branch for the same effect.

```
git clone -branch new_feature git://remoterepository.git
```

This above example would clone only the new_feature branch from the remote Git repository. This is purely a convince utility to save you time from downloading the HEAD ref of the repository and then having to additionally fetch the ref you need.

SAVING CHANGES TO THE REPOSITORY

The following example assumes you have set up a project at `/path/to/project`. The steps being taken in this example are:

- Change directories to `/path/to/project`
- Create a new file `CommitTest.txt` with contents `~"test content for git tutorial"~`
- `git add CommitTest.txt` to the repository staging area
- Create a new commit with a message describing what work was done in the commit

```
cd /path/to/project
echo "test content for git tutorial" >> CommitTest.txt
git add CommitTest.txt
git commit -m "added CommitTest.txt to the repo"
```

After executing this example, your repo will now have `CommitTest.txt` added to the history and will track future updates to the file.

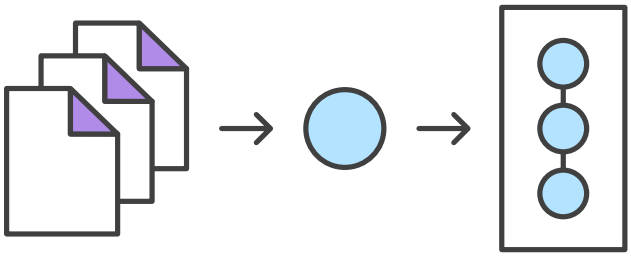
This example introduced two additional git commands: `add` and `commit`. Another common use case for git add is the `--all` option. Executing `git add --all` will take any changed and untracked files in the repo and add them to the repo and update the repo's working tree.

GIT ADD

The git add command adds a change in the working directory to the staging area. It tells Git that you want to include updates to a particular file in the next commit. However, git add doesn't really affect the repository in any significant way—changes are not actually recorded until you run git commit.

The `git add` and `git commit` commands compose the fundamental Git workflow. Developing a project revolves around the basic edit/stage/commit pattern:

1. Edit your files in the working directory.
2. When you're ready to save a copy of the current state of the project, you stage changes with git add.
3. After you're happy with the staged snapshot, you commit it to the project history with git commit.



In addition to git add and git commit, a third command git push is essential for a complete collaborative Git workflow. git push is utilized to send the committed changes to remote repositories for collaboration. This enables other team members to access a set of saved changes.

Staging area: The primary function of the git add command, is to promote pending changes in the working directory, to the git staging area. Instead of committing all of the changes you've made since the last commit, the stage lets you group related changes into highly focused snapshots before actually committing it to the project history. This means you can make all sorts of edits to unrelated files, then go back and split them up into logical commits by adding related changes to the stage and commit them piece-by-piece.

The following command stages all changes in the directory:

```
git add .
```

The following command stages a specific file:

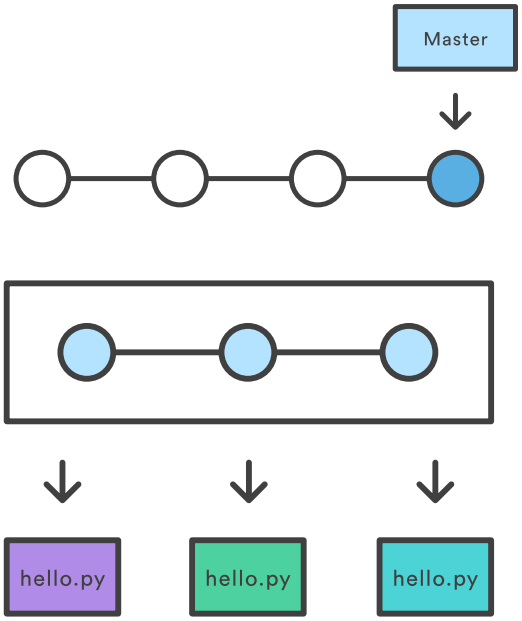
```
git add hello.py
```

GIT COMMIT

The `git commit` command captures a snapshot of the project's currently staged changes. Committed snapshots can be thought of as “safe” versions of a project—Git will never change them unless you explicitly ask it to.

At a high-level, Git can be thought of as a timeline management utility. Commits are the core building block units of a Git project timeline. Commits can be thought of as snapshots or milestones along the timeline of a Git project. Commits are created with the git commit command to capture the state of a project at that point in time. Git Snapshots are always committed to the local repository.

Git records the entire contents of each file in every commit.



- `git commit`
  Commit the staged snapshot. This will launch a text editor prompting you for a commit message. After you’ve entered a message, save the file and close the editor to create the actual commit.
- `git commit -a`
  Commit a snapshot of all changes in the working directory. This only includes modifications to tracked files (those that have been added with git add at some point in their history).
- `git commit -m "commit message"`
  A shortcut command that immediately creates a commit with a passed commit message. By default, git commit will open up the locally configured text editor, and prompt for a commit message to be entered. Passing the -m option will forgo the text editor prompt in-favor of an inline message.
- `git commit --amend`
  This option adds another level of functionality to the commit command. Passing this option will modify the last commit. Instead of creating a new commit, staged changes will be added to the previous commit. This command will open up the system's configured text editor and prompt to change the previously specified commit message.

EXAMPLE 1

The following example assumes you've edited some content in a file called hello.py on the current branch, and are ready to commit it to the project history. First, you need to stage the file with `git add`, then you can `commit` the staged snapshot.

```
git add hello.py
```

This command will add hello.py to the Git staging area. We can examine the result of this action by using the git status command.

```
git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
   new file: hello.py

git commit
```

This will open a text editor (customizable via git config) asking for a commit log message, along with a list of what's being committed:

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Changes to be committed:
# (use "git reset HEAD ..." to unstage)
#
#modified: hello.py
```

Git doesn't require commit messages to follow any specific formatting constraints, but the canonical format is to summarize the entire commit on the first line in less than 50 characters, leave a blank line, then a detailed explanation of what's been changed. For example:

```
Change the message displayed by hello.py


- Update the sayHello() function to output the user's name
- Change the sayGoodbye() function to a friendlier message
```

EXAMPLE 2

To continue with the hello.py example above. Let's make further updates to hello.py and execute the following:

```
git add hello.py
git commit --amend
```

This will once again, open up the configured text editor. This time, however, it will be pre-filled with the commit message we previously entered. This indicates that we are not creating a new commit, but editing the last.

GIT DIFF

Diffing is a function that takes two input data sets and outputs the changes between them. git diff is a multi-use Git command that when executed runs a diff function on Git data sources. These data sources can be commits, branches, files and more.

By default git diff will show you any uncommitted changes since the last commit.

```
git diff
```

GIT STASH

`git stash` temporarily shelves (or stashes) changes you've made to your working copy so you can work on something else, and then come back and re-apply them later on. Stashing is handy if you need to quickly switch context and work on something else, but you're mid-way through a code change and aren't quite ready to commit.

Stashing your work: The git stash command takes your uncommitted changes (both staged and unstaged), saves them away for later use, and then reverts them from your working copy. For example:

```
$ git status
On branch master
Changes to be committed:

    new file:   style.css

Changes not staged for commit:
    modified:   index.html

$ git stash
Saved working directory and index state WIP on master: 5002d47 our new homepage
HEAD is now at 5002d47 our new homepage

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

At this point you're free to make changes, create new commits, switch branches, and perform any other Git operations; then come back and re-apply your stash when you're ready.

Note that the stash is local to your Git repository; stashes are not transferred to the server when you push.

Re-apply stashed changes:

You can reapply previously stashed changes with git stash pop:

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

```
$ git stash pop
On branch master
Changes to be committed:

    new file:   style.css

Changes not staged for commit:

    modified:   index.html

Dropped refs/stash@{0} (32b3aa1d185dfe6d57b3c3cc3b32cbf3e380cc6a)
```

Popping your stash removes the changes from your stash and reapplies them to your working copy.

Alternatively, you can reapply the changes to your working copy and keep them in your stash with git stash apply:

```
$ git stash apply
On branch master
Changes to be committed:

    new file:   style.css

Changes not staged for commit:

    modified:   index.html
```

This is useful if you want to apply the same stashed changes to multiple branches.

.GITIGNORE

Git sees every file in your working copy as one of three things:

- *tracked* - a file which has been previously staged or committed;
- *untracked* - a file which has not been staged or committed; or
- *ignored* - a file which Git has been explicitly told to ignore.

Ignored files are usually build artifacts and machine generated files that can be derived from your repository source or should otherwise not be committed. Some common examples are:

- *dependency caches*, such as the contents of /node_modules or /packages
- *compiled code*, such as .o, .pyc, and .class files
- *build output directories*, such as /bin, /out, or /target
- *files generated at runtime*, such as .log, .lock, or .tmp
- *hidden system files*, such as .DS_Store or Thumbs.db
- *personal IDE config files*, such as .idea/workspace.xml

Ignored files are tracked in a special file named `.gitignore` that is checked in at the root of your repository. There is no explicit git ignore command: instead the `.gitignore` file must be edited and committed by hand when you have new files that you wish to ignore. `.gitignore` files contain patterns that are matched against file names in your repository to determine whether or not they should be ignored.

Some examples are:

`*/logs`	`logs/debug.log`
	`logs/monday/foo.bar`
	`build/logs/debug.log`
		You can prepend a pattern with a double asterisk to match directories anywhere in the repository.

`**/logs/debug.log`	`logs/debug.log`
	`build/logs/debug.log`
	but not
	`logs/build/debug.log`
		You can also use a double asterisk to match files based on their name and the name of their parent directory.

`*.log`	`debug.log`
	`foo.log`
	`.log`
	`logs/debug.log`
		An asterisk is a wildcard that matches zero or more characters.

`logs`	`logs`
	`logs/debug.log`
	`logs/latest/foo.bar`
	`build/logs`
	`build/logs/debug.log`
		If you don't append a slash, the pattern will match both files and the contents of directories with that name. In the example matches on the left, both directories and files named logsare ignored

`logs/`	`logs/debug.log`
	`logs/latest/foo.bar`
	`build/logs/foo.bar`
	`build/logs/latest/debug.log`
		Appending a slash indicates the pattern is a directory. The entire contents of any directory in the repository matching that name – including all of its files and subdirectories – will be ignored

INSPECTING A REPOSITORY

GIT STATUS

The git status command displays the state of the working directory and the staging area. It lets you see which changes have been staged, which haven't, and which files aren't being tracked by Git.

USAGE

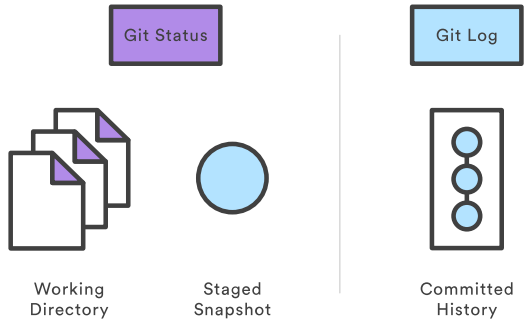
```
git status
```

The git status command is a relatively straightforward command. It simply shows you what's been going on with git add and git commit. Status messages also include relevant instructions for staging/unstaging files. Sample output showing the three main categories of a git status call is included below:

```
# On branch master
# Changes to be committed:
# (use "git reset HEAD <file>..." to unstage)
#
#modified: hello.py
#
# 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: main.py
#
# Untracked files:
# (use "git add <file>..." to include in what will be committed)
#
#hello.pyc
```

GIT LOG

The git log command displays committed snapshots. It lets you list the project history, filter it, and search for specific changes. While git status lets you inspect the working directory and the staging area, git log only operates on the committed history.



USAGE

git log
Display the entire commit history using the default formatting. If the output takes up more than one screen, you can use Space to scroll and q to exit.

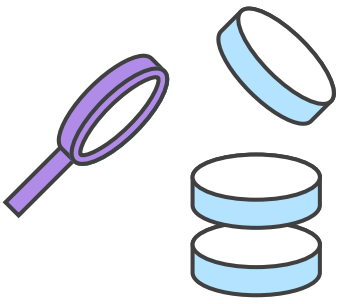
- `git log -n <limit>`
  Limit the number of commits by . For example, git log -n 3 will display only 3 commits.
- `git log --oneline`
  Condense each commit to a single line. This is useful for getting a high-level overview of the project history.
- `git log -stat`
  Along with the ordinary git log information, include which files were altered and the relative number of lines that were added or deleted from each of them.
- `git log -p`
  Display the patch representing each commit. This shows the full diff of each commit, which is the most detailed view you can have of your project history.
- `git log --author="<pattern>"`
  Search for commits by a particular author. The argument can be a plain string or a regular expression.
- `git log --grep="<pattern>"`
  Search for commits with a commit message that matches , which can be a plain string or a regular expression.
- `git log <since>..<until>`
  Show only commits that occur between < since > and < until >. Both arguments can be either a commit ID, a branch name, HEAD, or any other kind of revision reference.
- `git log <file>`
  Only display commits that include the specified file. This is an easy way to see the history of a particular file.
- `git log --graph --decorate –oneline`
  A few useful options to consider. The --graph flag that will draw a text based graph of the commits on the left hand side of the commit messages. --decorate adds the names of branches or tags of the commits that are shown. --oneline shows the commit information on a single line making it easier to browse through commits at-a-glance.

EXAMPLE

```
git log --author="John Smith" -p hello.py
```

This will display a full diff of all the changes John Smith has made to the file hello.py.

GIT TAG



Tags are ref's that point to specific points in Git history. Tagging is generally used to capture a point in history that is used for a marked version release (i.e. v1.0.1). A tag is like a branch that doesn't change. Unlike branches, tags, after being created, have no further history of commits.

CREATING A TAG

To create a new tag execute the following command:

```
git tag <tagname>
```

Annotated tags are stored as full objects in the Git database. To reiterate, they store extra meta data such as: the tagger name, email, and date.

```
git tag -a v1.4
```

Executing this command is similar to the previous invocation, however, this version of the command is passed the -m option and a message.

```
git tag -a v1.4 -m "my version 1.4"
```

To list stored tags in a repo execute the following:

```
git tag
```

This will output a list of tags:

```
v0.10.0
    v0.10.0-rc1
    v0.11.0
    v0.11.0-rc1
    v0.11.1
    v0.11.2
    v0.12.0
    v0.12.0-rc1
    v0.12.1
    v0.12.2
    v0.13.0
    v0.13.0-rc1
    v0.13.0-rc2
```

TAGGING OLD COMMITS

The previous tagging examples have demonstrated operations on implicit commits. By default, git tag will create a tag on the commit that HEAD is referencing. Alternatively git tag can be passed as a ref to a specific commit. This will tag the passed commit instead of defaulting to HEAD. To gather a list of older commits execute the git log command.

```
$ git log --pretty=oneline
    15027957951b64cf874c3557a0f3547bd83b3ff6 Merge branch 'feature'
    a6b4c97498bd301d84096da251c98a07c7723e65 add update method for thing
    0d52aaab4479697da7686c15f77a3d64d9165190 one more thing
    6d52a271eda8725415634dd79daabbc4d9b6008e Merge branch 'experiment'
```

Executing git log will output a list of commits. In this example we will pick the top most commit Merge branch 'feature' for the new tag. We will need to reference to the commit SHA hash to pass to Git:

```
git tag -a v1.2 15027957951b64cf874c3557a0f3547bd83b3ff6
```

Executing the above git tag invocation will create a new annotated commit identified as v1.2 for the commit we selected in the previous git log example.

PUSHING TAGS TO REMOTE

Sharing tags is similar to pushing branches. By default, git push will not push tags. Tags have to be explicitly passed to git push.

```
$ git push origin v1.4
    Counting objects: 14, done.
    Delta compression using up to 8 threads.
    Compressing objects: 100% (12/12), done.
    Writing objects: 100% (14/14), 2.05 KiB | 0 bytes/s, done.
    Total 14 (delta 3), reused 0 (delta 0)
    To git@bitbucket.com:atlasbro/gittagdocs.git
     * [new tag]         v1.4 -> v1.4
```

CHECKING OUT TAGS

You can view the state of a repo at a tag by using the git checkout command.

```
git checkout v1.4
```

The above command will checkout the v1.4 tag. This puts the repo in a detached HEAD state. This means any changes made will not update the tag. They will create a new detached commit. This new detached commit will not be part of any branch and will only be reachable directly by the commits SHA hash. Therefore it is a best practice to create a new branch anytime you're making changes in a detached HEAD state.

DELETING TAGS

Deleting tags is a straightforward operation. Passing the -d option and a tag identifier to git tag will delete the identified tag.

```
$ git tag
    v1
    v2
    v3

$ git tag -d v1

$ git tag
    v2
    v3
```

GIT BLAME

The git blame command is a versatile troubleshooting utility that has extensive usage options. The high-level function of git blame is the display of author metadata attached to specific committed lines in a file. This is used to examine specific points of a file's history and get context as to who the last author was that modified the line. This is used to explore the history of specific code and answer questions about what, how, and why the code was added to a repository.

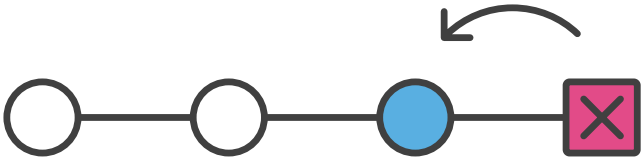
git blame only operates on individual files. A file-path is required for any useful output. The default execution of git blame will simply output the commands help menu. For this example, we will operate on the README.MD file.

```
git blame README.MD
```

- `git blame -L 1,5 README.md`
  The -L option will restrict the output to the requested line range. Here we have restricted the output to lines 1 through 5.
- `git blame -e README.md`
  The -e option shows the authors email address instead of username.
- `git blame -w README.md`
  The -w option ignores whitespace changes. If a previous author has modified the spacing of a file by switching from tabs to spaces or adding new lines this, unfortunately, obscures the output of git blame by showing these changes.
- `git blame -M README.md`
  The -M option detects moved or copied lines within in the same file. This will report the original author of the lines instead of the last author that moved or copied the lines.
- `git blame -C README.md`
  The -C option detects lines that were moved or copied from other files. This will report the original author of the lines instead of the last author that moved or copied the lines.

UNDOING CHANGES

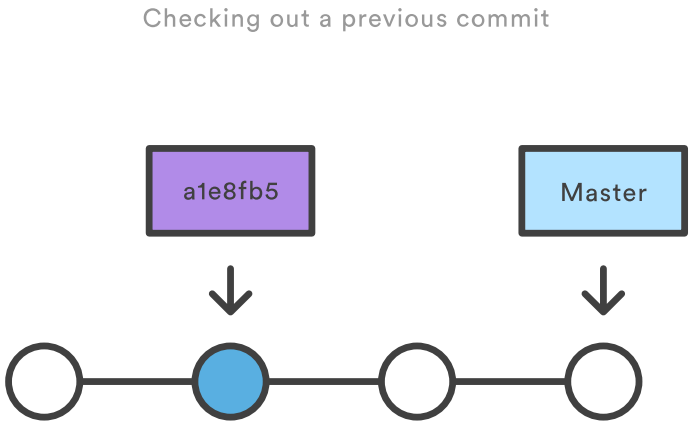
we will discuss the available 'undo' Git strategies and commands. It is first important to note that Git does not have a traditional 'undo' system like those found in a word processing application.



GIT CHECKOUT

Each commit has a unique SHA-1 identifying hash. These IDs are used to travel through the committed timeline and revisit commits.

When you have found a commit reference to the point in history you want to visit, you can utilize the `git checkout` command to visit that commit. `Git checkout` is an easy way to “load” any of these saved snapshots onto your development machine. During the normal course of development, the `HEAD` usually points to `master` or some other local branch, but when you check out a previous commit, `HEAD` no longer points to a branch—it points directly to a commit. This is called a “detached `HEAD`” state, and it can be visualized as the following:



Checking out an old file does not move the `HEAD` pointer. It remains on the same branch and same commit, avoiding a 'detached head' state. You can then commit the old version of the file in a new snapshot as you would any other changes. So, in effect, this usage of `git checkout` on a file, serves as a way to revert back to an old version of an individual file.

UNDOING A COMMITTED SNAPSHOT

There are technically several different strategies to 'undo' a commit. The following examples will assume we have a commit history that looks like:

```
git log --oneline
872fa7e Try something crazy
a1e8fb5 Make some important changes to hello.txt
435b61d Create hello.txt
9773e52 Initial import
```

We will focus on undoing the `872fa7e` Try something crazy commit. Maybe things got a little too crazy.

Using the git checkout command we can checkout the previous commit, `a1e8fb5`, putting the repository in a state before the crazy commit happened. Checking out a specific commit will put the repo in a "detached `HEAD`" state. This means you are no longer working on any branch. In a detached state, any new commits you make will be orphaned when you change branches back to an established branch. Orphaned commits are up for deletion by Git's garbage collector.

From the detached `HEAD` state, we can execute

```
git checkout -b new_branch_without_crazy_commit
```

This will create a new branch named `new_branch_without_crazy_commit` and switch to that state. The repo is now on a new history timeline in which the `872fa7e` commit no longer exists. At this point, we can continue work on this new branch in which the `872fa7e` commit no longer exists and consider it 'undone'.

Unfortunately, if you need the previous branch, maybe it was your master branch, this undo strategy is not appropriate. Let's look at some other 'undo' strategies.

GIT REVERT

Let's assume we are back to our original commit history example. The history that includes the `872fa7e` commit. This time let's try a `revert` 'undo'. If we execute

```
git revert HEAD
```

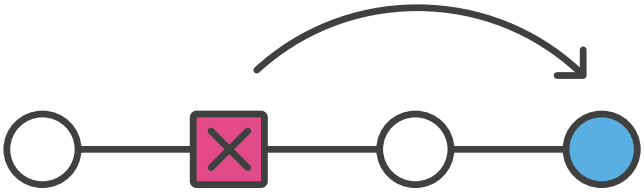
Git will create a new commit with the inverse of the last commit. This adds a new commit to the current branch history and now makes it look like:

```
git log --oneline
e2f9a78 Revert "Try something crazy"
872fa7e Try something crazy
a1e8fb5 Make some important changes to hello.txt
435b61d Create hello.txt
9773e52 Initial import
```

At this point, we have again technically 'undone' the `872fa7e` commit. Although `872fa7e` still exists in the history, the new `e2f9a78` commit is an inverse of the changes in `872fa7e`. Unlike our previous checkout strategy, we can continue using the same branch. This solution is a satisfactory undo. This is the ideal 'undo' method for working with public shared repositories.

The git revert command can be considered an 'undo' type command, however, it is not a traditional undo operation. Instead of removing the commit from the project history, it figures out how to invert the changes introduced by the commit and appends a new commit with the resulting inverse content.

Reverting should be used when you want to apply the inverse of a commit from your project history. This can be useful, for example, if you're tracking down a bug and find that it was introduced by a single commit. Instead of manually going in, fixing it, and committing a new snapshot, you can use git revert to automatically do all of this for you.



A revert operation will take a specified commit, inverse the changes from that commit, and create a new "revert commit". The ref pointers are then updated to point at the new revert commit making it the tip of the branch.

```
$ mkdir git_revert_test
$ cd git_revert_test/
$ git init .
Initialized empty Git repository in /git_revert_test/.git/
$ touch demo_file
$ git add demo_file
$ git commit -am"initial commit"
[master (root-commit) 299b15f] initial commit
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 demo_file
$ echo "initial content" >> demo_file
$ git commit -am"add new content to demo file"
[master 3602d88] add new content to demo file
n 1 file changed, 1 insertion(+)
$ echo "prepended line content" >> demo_file
$ git commit -am"prepend content to demo file"
[master 86bb32e] prepend content to demo file
 1 file changed, 1 insertion(+)
$ git log -oneline
86bb32e prepend content to demo file
3602d88 add new content to demo file
299b15f initial commit
```

With the repo in this state, we are ready to initiate a git revert.

```
$ git revert HEAD
[master b9cd081] Revert "prepend content to demo file"
1 file changed, 1 deletion(-)
```

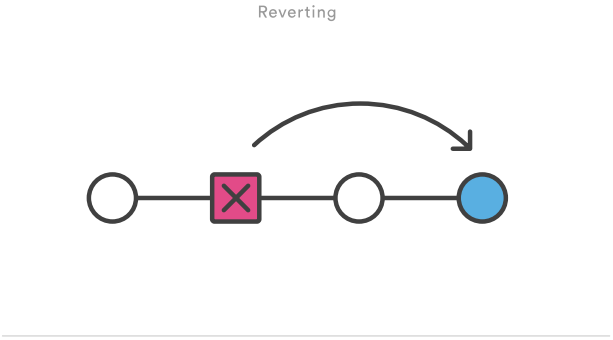
Git revert expects a commit ref was passed in and will not execute without one. Here we have passed in the `HEAD` ref. This will revert the latest commit. This is the same behavior as if we reverted to commit `3602d8815dbfa78cd37cd4d189552764b5e96c58`

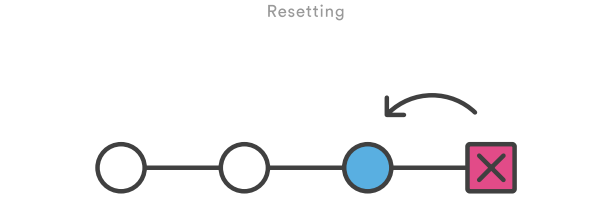
We can now examine the state of the repo using git log and see that there is a new commit added to the previous log:

```
$ git log --oneline
1061e79 Revert "prepend content to demo file"
86bb32e prepend content to demo file
3602d88 add new content to demo file
299b15f initial commit
```

- `-e or –edit`
  This is a default option and doesn't need to be specified. This option will open the configured system editor and prompts you to edit the commit message prior to committing the revert
- `--no-edit`
  This is the inverse of the -e option. The revert will not open the editor.
- `-n or --no-commit`
  Passing this option will prevent git revert from creating a new commit that inverses the target commit. Instead of creating the new commit this option will add the inverse changes to the Staging Index and Working Directory.

It's important to understand that git revert undoes a single commit—it does not "revert" back to the previous state of a project by removing all subsequent commits.





Reverting has two important advantages over resetting. First, it doesn’t change the project history, which makes it a “safe” operation for commits that have already been published to a shared repository.

Second, git revert is able to target an individual commit at an arbitrary point in the history, whereas git reset can only work backward from the current commit.

GIT RESET

For this undo strategy we will continue with our working example. `git reset` is an extensive command with multiple uses and functions. If we invoke

```
git reset --hard a1e8fb5
```

the commit history is reset to that specified commit. Examining the commit history with git log will now look like:

```
git log --oneline
a1e8fb5 Make some important changes to hello.txt
435b61d Create hello.txt
9773e52 Initial import
```

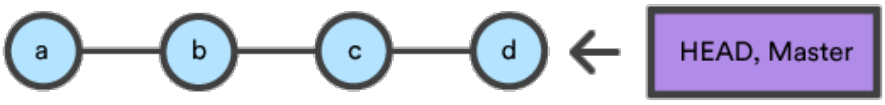
The log output shows the `e2f9a78` and `872fa7e` commits no longer exist in the commit history. At this point, we can continue working and creating new commits as if the 'crazy' commits never happened. This method of undoing changes has the cleanest effect on history. Doing a reset is great for local changes however it adds complications when working with a shared remote repository. If we have a shared remote repository that has the `872fa7e` commit pushed to it, and we try to git push a branch where we have reset the history, Git will catch this and throw an error. Git will assume that the branch being pushed is not up to date because of it's missing commits. In these scenarios, git revert should be the preferred undo method.

The `git reset` command is a complex and versatile tool for undoing changes. It has three primary forms of invocation. These forms correspond to command line arguments `--soft`, `--mixed`, `--hard`. The three arguments each correspond to Git's three internal state management mechanism's, The Commit Tree (HEAD), The Staging Index, and The Working Directory.

To properly understand git reset usage, we must first understand Git's internal state management systems. Sometimes these mechanisms are called Git's "three trees".

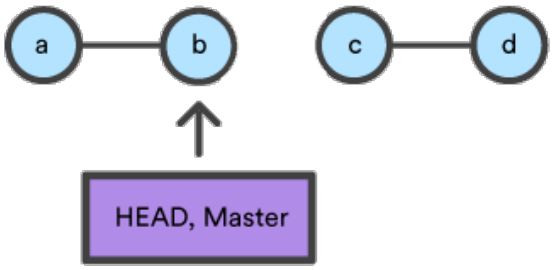
- **The working directory:** The first tree we will examine is "The Working Directory". This tree is in sync with the local filesystem and is representative of the immediate changes made to content in files and directories.
- **Staging index:** Next up is the 'Staging Index' tree. This tree is tracking Working Directory changes, that have been promoted with git add, to be stored in the next commit.
- **Commit history:** The final tree is the Commit History. The git commit command adds changes to a permanent snapshot that lives in the Commit History.

Consider the following example:



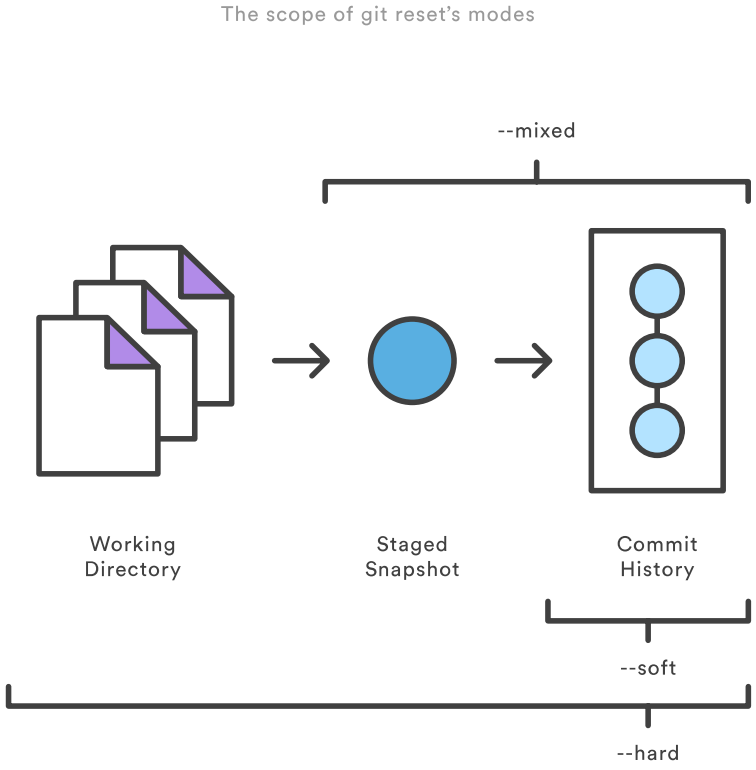
This example demonstrates a sequence of commits on the master branch. The HEAD ref and master branch ref currently point to commit d.

git reset, moves both the HEAD and branch refs to the specified commit.



In addition to updating the commit ref pointers, git reset will modify the state of the three trees. The ref pointer modification always happens and is an update to the third tree, the Commit tree. The command line arguments `--soft`, `--mixed`, and `--hard` direct how to modify the Staging Index, and Working Directory trees.

MAIN OPTIONS



HARD

This is the most direct, DANGEROUS, and frequently used option. When passed --hard The Commit History ref pointers are updated to the specified commit. Then, the Staging Index and Working Directory are reset to match that of the specified commit. Any previously pending changes to the Staging Index and the Working Directory gets reset to match the state of the Commit Tree. This means any pending work that was hanging out in the Staging Index and Working Directory will be lost.

MIXED

This is the default operating mode. The ref pointers are updated. The Staging Index is reset to the state of the specified commit. Any changes that have been undone from the Staging Index are moved to the Working Directory. Let us continue.

SOFT

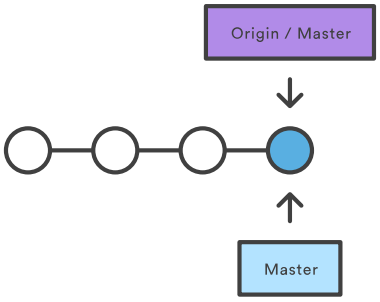
When the --soft argument is passed, the ref pointers are updated and the reset stops there. The Staging Index and the Working Directory are left untouched.

DON'T RESET PUBLIC HISTORY

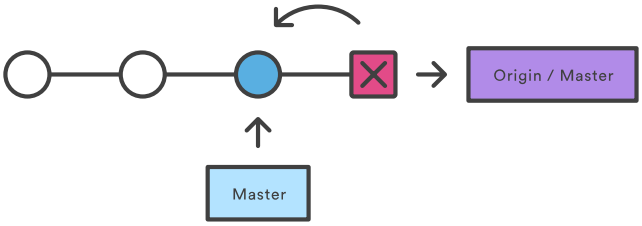
You should never use git reset when any snapshots after have been pushed to a public repository. After publishing a commit, you have to assume that other developers are reliant upon it.

Removing a commit that other team members have continued developing poses serious problems for collaboration. When they try to sync up with your repository, it will look like a chunk of the project history abruptly disappeared. The sequence below demonstrates what happens when you try to reset a public commit. The origin/master branch is the central repository's version of your local master branch.

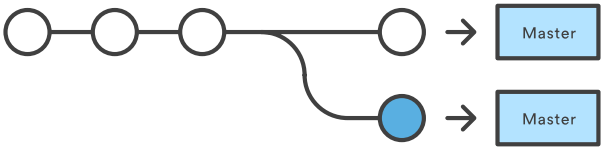
Resetting



After Resetting



After Committing



As soon as you add new commits after the reset, Git will think that your local history has diverged from origin/master, and the merge commit required to synchronize your repositories is likely to confuse and frustrate your team.

The point is, make sure that you're using git reset on a local experiment that went wrong—not on published changes. If you need to fix a public commit, the git revert command was designed specifically for this purpose.

GIT COMMIT AMEND

In some cases, you might not need to remove or reset the last commit. Maybe it was just made prematurely. In this case you can `amend` the most recent commit. Once you have made more changes in the working directory and staged them for commit by using git add, you can execute

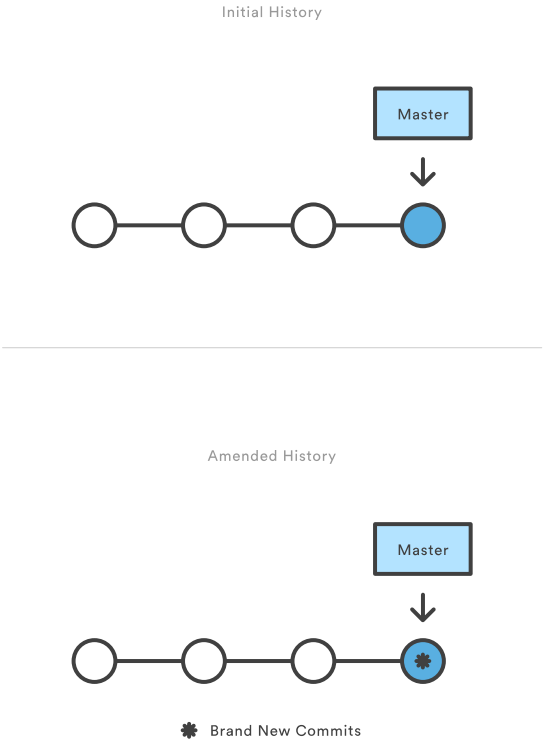
```
git commit --amend
```

This will have Git open the configured system editor and let you modify the last commit message. The new changes will be added to the amended commit.

REWRITING HISTORY

GIT COMMIT AMEND

The git commit --amend command is a convenient way to modify the most recent commit. It lets you combine staged changes with the previous commit instead of creating an entirely new commit. It can also be used to simply edit the previous commit message without changing its snapshot.



```
git commit --amend
```

Let's say you just committed and you made a mistake in your commit log message. Running this command when there is nothing staged lets you edit the previous commit's message without altering its snapshot.

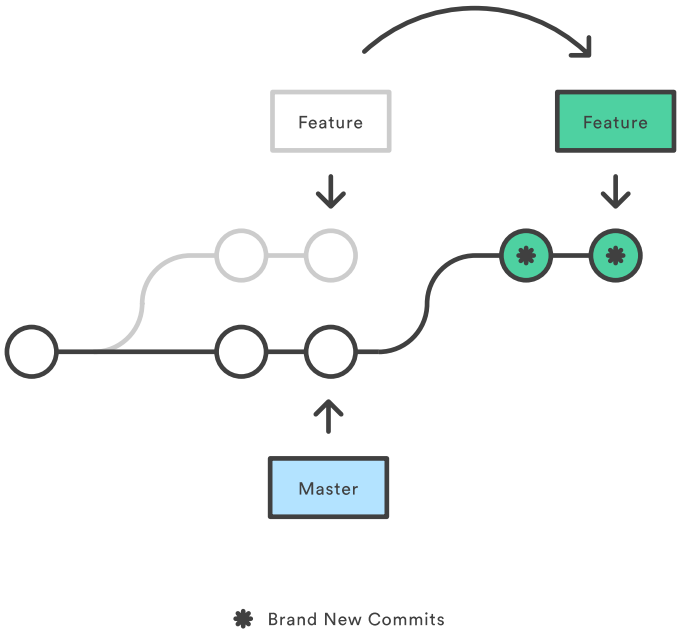
```
git commit --amend -m "an updated commit message"
```

Adding the -m option allows you to pass in a new message from the command line without being prompted to open an editor.

Don't amend public commits: Amended commits are actually entirely new commits and the previous commit will no longer be on your current branch. This has the same consequences as resetting a public snapshot.

GIT REBASE

Rebasing is the process of moving or combining a sequence of commits to a new base commit. Rebasing is most useful and easily visualized in the context of a feature branching workflow. The general process can be visualized as the following:



Commits auf akt. Stand bringen

From a content perspective, rebasing is changing the base of your branch from one commit to another making it appear as if you'd created your branch from a different commit. Internally, Git accomplishes this by creating new commits and applying them to the specified base. It's very important to understand that even though the branch looks the same, it's composed of entirely new commits.

Don't rebase public history As we've discussed previously in rewriting history, you should never rebase commits once they've been pushed to a public repository. The rebase would replace the old commits with new ones and it would look like that part of your project history abruptly vanished.

GIT REBASE STANDARD VS. GIT REBASE INTERACTIVE

Git rebase interactive is when git rebase accepts an -- i argument. This stands for "Interactive." Without any arguments, the command runs in standard mode.

Git rebase in **standard mode** will automatically take the commits in your current working branch and apply them to the head of the passed branch.

```
git rebase <base>
```

This automatically rebases the current branch onto , which can be any kind of commit reference (for example an ID, a branch name, a tag, or a relative reference to HEAD).

Running git rebase with the -i flag begins an **interactive rebasing session**. Instead of blindly moving all of the commits to the new base, interactive rebasing gives you the opportunity to alter individual commits in the process. This lets you clean up history by removing, splitting, and altering an existing series of commits.

```
git rebase --interactive <base>
```

This rebases the current branch onto but uses an interactive rebasing session. This opens an editor where you can enter commands for each commit to be rebased. These commands determine how individual commits will be transferred to the new base. You can also reorder the commit listing to change the order of the commits themselves. Once you've specified commands for each commit in the rebase, Git will begin playing back commits applying the rebase commands. The rebasing edit commands are as follows:

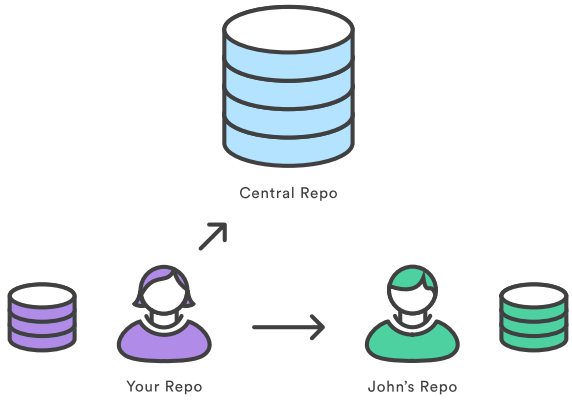
```
pick 2231360 some old commit
pick ee2adc2 Adds new feature

# Rebase 2cf755d..ee2adc2 onto 2cf755d (9 commands)
#
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
# x, exec = run command (the rest of the line) using shell
# d, drop = remove commit
```

SYNCING

GIT REMOTE

The git remote command lets you create, view, and delete connections to other repositories. Instead of providing real-time access to another repository, they serve as convenient names that can be used to reference a not-so-convenient URL.

For example, the following diagram shows two remote connections from your repo into the central repo and another developer's repo. Instead of referencing them by their full URLs, you can pass the origin and john shortcuts to other Git commands.



- `git remote`
  List the remote connections you have to other repositories.
- `git remote -v`
  Same as the above command, but include the URL of each connection.
- `git remote add <name> <url>`
  Create a new connection to a remote repository. After adding a remote, you'll be able to use as a convenient shortcut for in other Git commands.
- `git remote rm <name>`
  Remove the connection to the remote repository called .
- `git remote rename <old-name> <new-name>`
  Rename a remote connection from to .

GIT FETCH

The git fetch command downloads commits, files, and refs from a remote repository into your local repo. Fetching is what you do when you want to see what everybody else has been working on. Fetched content has to be explicitly checked out using the git checkout command. This makes fetching a safe way to review commits before integrating them with your local repository.

When downloading content from a remote repo, git pull and git fetch commands are available to accomplish the task. You can consider git fetch the 'safe' version of the two commands. It will download the remote content but not update your local repo's working state, leaving your current work intact. git pull is the more aggressive alternative; it will download the remote content for the active local branch and immediately execute git merge to create a merge commit for the new remote content.

Behind the scenes, Git stores all commits, local and remote. Git keeps remote and local branch commits distinctly separate through the use of branch refs.

```
git branch -r
# origin/master
# origin/feature1
# origin/debug2
# remote-repo/master
# remote-repo/other-feature
```

You can inspect remote branches with the usual git checkout and git log commands. If you approve the changes a remote branch contains, you can merge it into a local branch with a normal `git merge`.

- `git fetch <remote>`
  Fetch all of the branches from the repository. This also downloads all of the required commits and files from the other repository.
- `git fetch <remote> <branch>`
  Same as the above command, but only fetch the specified branch.
- `git fetch –all`
  A power move which fetches all registered remotes and their branches:
- `git fetch --dry-run`
  The --dry-run option will perform a demo run of the command. It will output examples of actions it will take during the fetch but not apply them.

EXAMPLE

In this example, let us assume there is a central repo origin from which the local repository has been cloned from using the git clone command. Let us also assume an additional remote repository named coworkers_repo that contains a feature_branch which we will configure and fetch. With these assumptions set let us continue the example.

```
git remote add coworkers_repo git@bitbucket.org:coworker/coworkers_repo.git
```

Here we have created a reference to the coworker's repo using the repo URL. We will now pass that remote name to git fetch to download the contents.

```
git fetch coworkers_repo coworkers/feature_branch
fetching coworkers/feature_branch
```

We now locally have the contents of coworkers/feature_branch we will need the integrate this into our local working copy. We begin this process by using the git checkout command to checkout the newly downloaded remote branch.

```
git checkout coworkers/feature_branch
Note: checking out coworkers/feature_branch'.
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>
```

The output from this checkout operation indicates that we are in a detached HEAD state. This is expected and means that our HEAD ref is pointing to a ref that is not in sequence with our local history. Being that HEAD is pointed at the coworkers/feature_branch ref, we can create a new local branch from that ref. The 'detached HEAD' output shows us how to do this using the git checkout command:

```
git checkout -b local_feature_branch
```

Here we have created a new local branch named local_feature_branch. This puts updates HEAD to point at the latest remote content and we can continue development on it from this point.

EXAMPLE 2

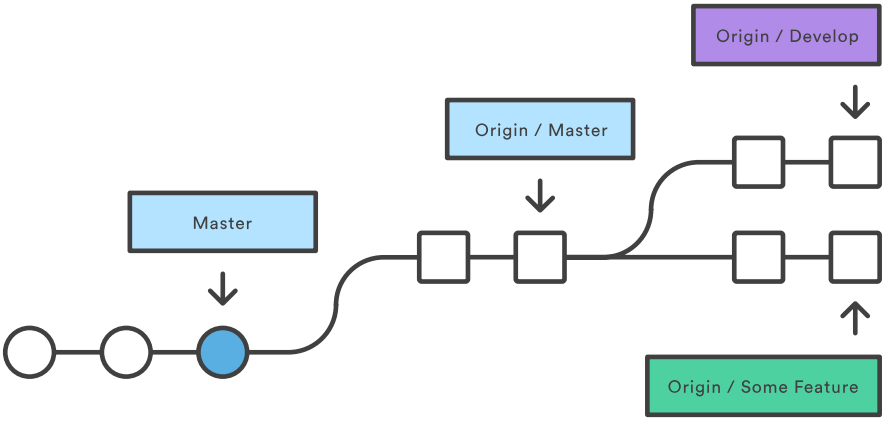
The following example walks through the typical workflow for synchronizing your local repository with the central repository's master branch.

```
git fetch origin
```

This will display the branches that were downloaded:

```
a1e8fb5..45e66a4 master -> origin/master
a1e8fb5..9e8ab1c develop -> origin/develop
* [new branch] some-feature -> origin/some-feature
```

The commits from these new remote branches are shown as squares instead of circles in the diagram below. As you can see, git fetch gives you access to the entire branch structure of another repository.



To see what commits have been added to the upstream master, you can run a git log using origin/master as a filter:

```
git log --oneline master..origin/master
```

To approve the changes and merge them into your local master branch use the following commands:

```
git checkout master
git log origin/master
```

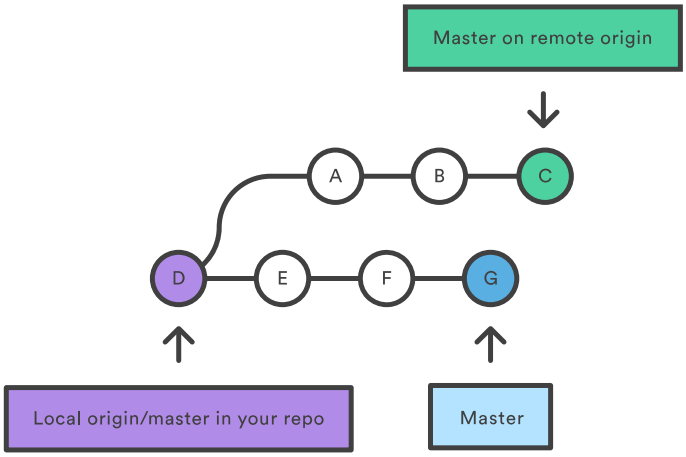
Then we can use git merge origin/master:

```
git merge origin/master
```

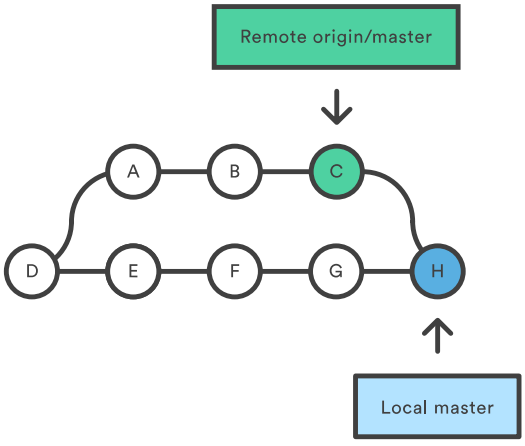
The origin/master and master branches now point to the same commit, and you are synchronized with the upstream developments.

GIT PULL

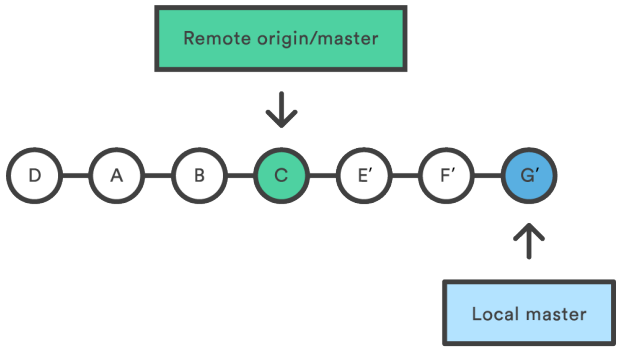
The `git pull` command first runs `git fetch` which downloads content from the specified remote repository. Then a `git merge` is executed to merge the remote content refs and heads into a new local merge commit.



In this scenario, git pull will download all the changes from the point where the local and master diverged. In this example, that point is E. git pull will fetch the diverged remote commits which are A-B-C. The pull process will then create a new local merge commit containing the content of the new diverged remote commits.



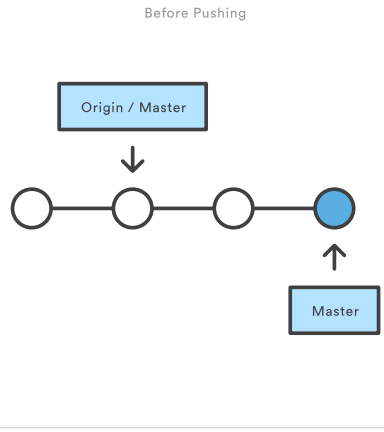
In the above diagram, we can see the new commit H. This commit is a new merge commit that contains the contents of remote A-B-C commits and has a combined log message. This example is one of a few `git pull` merging strategies. A `--rebase` option can be passed to `git pull` to use a rebase merging strategy instead of a merge commit. The next example will demonstrate how a rebase pull works. Assume that we are at a starting point of our first diagram, and we have executed `git pull --rebase`.

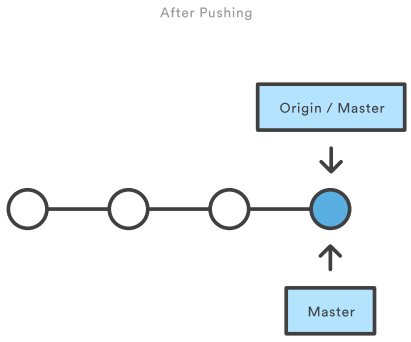


- `git pull <remote>`
  Fetch the specified remote's copy of the current branch and immediately merge it into the local copy. This is the same as git fetch ＜remote＞ followed by git merge origin/＜current-branch＞.
- `git pull --no-commit <remote>`
  Similar to the default invocation, fetches the remote content but does not create a new merge commit.
- `git pull --rebase <remote>`
  Same as the previous pull Instead of using git merge to integrate the remote branch with the local one, use git rebase.
- `git pull –verbose`
  Gives verbose output during a pull which displays the content being downloaded and the merge details.

GIT PUSH

The git push command is used to upload local repository content to a remote repository. Pushing is how you transfer commits from your local repository to a remote repo. It's the counterpart to git fetch, but whereas fetching imports commits to local branches, pushing exports commits to remote branches.



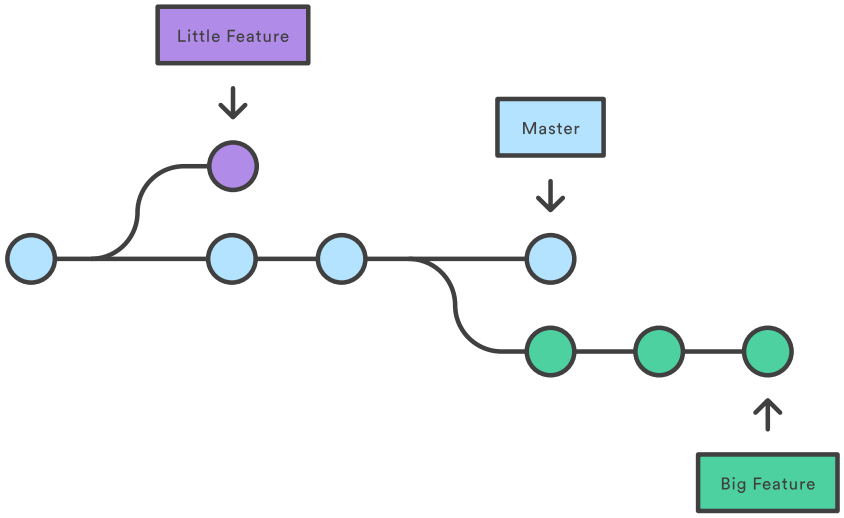


- `git push <remote> <branch>`
  Push the specified branch to , along with all of the necessary commits and internal objects. This creates a local branch in the destination repository. To prevent you from overwriting commits, Git won't let you push when it results in a non-fast-forward merge in the destination repository.
- `git push <remote> --force`
  Same as the above command, but force the push even if it results in a non-fast-forward merge. Do not use the --force flag unless you're absolutely sure you know what you're doing.
- `git push <remote> --all`
  Push all of your local branches to the specified remote.
- `git push <remote> --tags`
  Tags are not automatically pushed when you push a branch or use the --all option. The --tags flag sends all of your local tags to the remote repository.

BRANCHES

GIT BRANCH

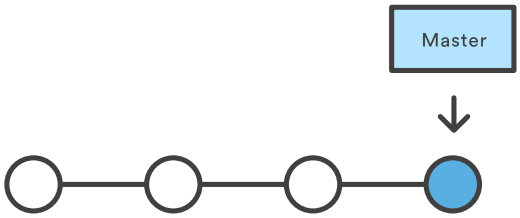
A branch represents an independent line of development. Branches serve as an abstraction for the edit/stage/commit process. You can think of them as a way to request a brand new working directory, staging area, and project history. New commits are recorded in the history for the current branch, which results in a fork in the history of the project.



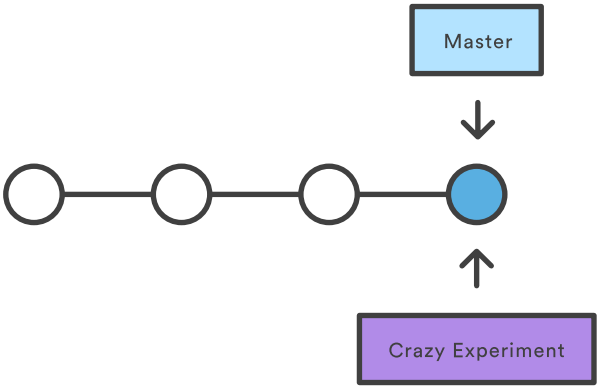
The diagram above visualizes a repository with two isolated lines of development, one for a little feature, and one for a longer-running feature. By developing them in branches, it's not only possible to work on both of them in parallel, but it also keeps the main master branch free from questionable code.

The git branch command lets you create, list, rename, and delete branches.

- `git branch`
  List all of the branches in your repository. This is synonymous with git branch --list.
- `git branch <branch>`
  Create a new branch called ＜branch＞. This does not check out the new branch.
- `git branch -d <branch>`
  Delete the specified branch. This is a "safe" operation in that Git prevents you from deleting the branch if it has unmerged changes.
- `git branch -D <branch>`
  Force delete the specified branch, even if it has unmerged changes. This is the command to use if you want to permanently throw away all of the commits associated with a particular line of development.
- `git branch -m <branch>`
  Rename the current branch to ＜branch＞.
- `git branch -a`
  List all remote branches.




```
git branch crazy-experiment
```



GIT CHECKOUT

The git checkout command lets you navigate between the branches created by git branch. Checking out a branch updates the files in the working directory to match the version stored in that branch, and it tells Git to record all new commits on that branch.

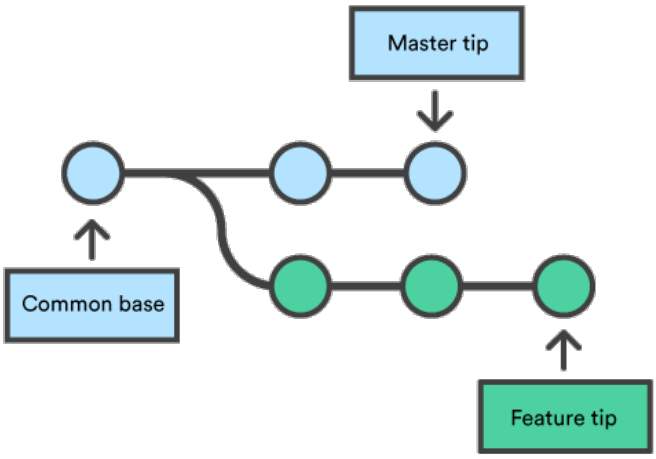
```
git branch
master
another_branch
feature_inprogress_branch

git checkout feature_inprogress_branch
```

The above example demonstrates how to view a list of available branches by executing the git branch command, and switch to a specified branch, in this case, the feature_inprogress_branch.

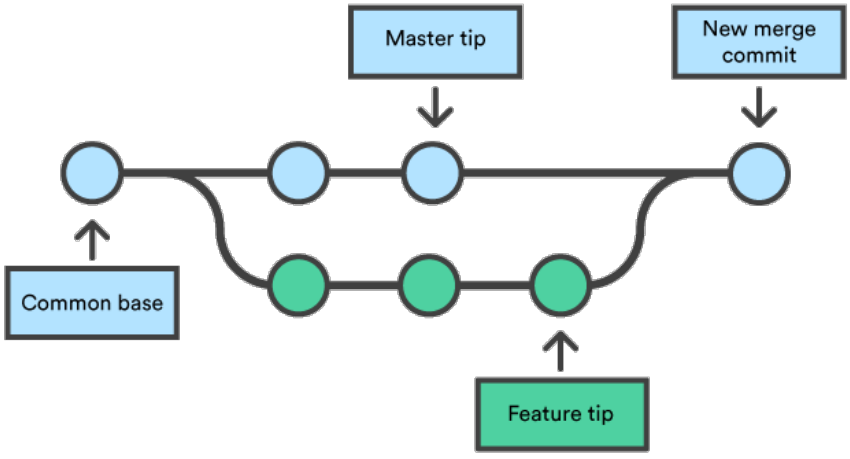
GIT MERGE

Git merge will combine multiple sequences of commits into one unified history. In the most frequent use cases, git merge is used to combine two branches. The following examples in this document will focus on this branch merging pattern. In these scenarios, git merge takes two commit pointers, usually the branch tips, and will find a common base commit between them. Once Git finds a common base commit it will create a new "merge commit" that combines the changes of each queued merge commit sequence.

Say we have a new branch feature that is based off the master branch. We now want to merge this feature branch into master.



Invoking this command will merge the specified branch feature into the current branch, we'll assume master. Git will determine the merge algorithm automatically.

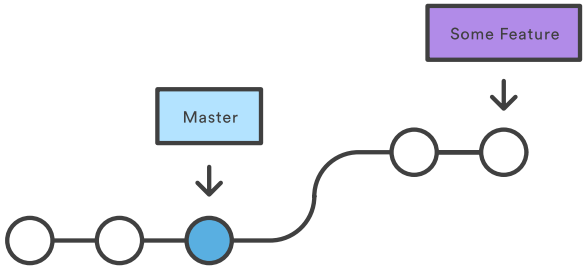


Merge commits are unique against other commits in the fact that they have two parent commits. When creating a merge commit Git will attempt to auto magically merge the separate histories for you. If Git encounters a piece of data that is changed in both histories **it will be unable to automatically combine them**.

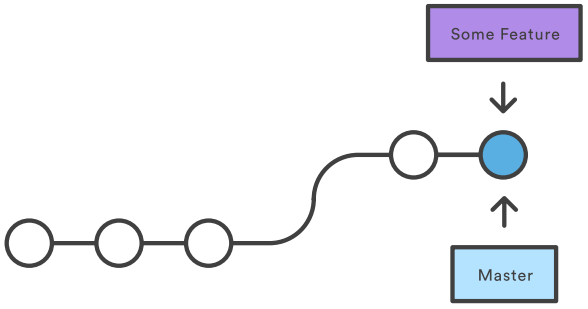
FAST FORWARD MERGE

A fast-forward merge can occur when there is a linear path from the current branch tip to the target branch. Instead of “actually” merging the branches, all Git has to do to integrate the histories is move (i.e., “fast forward”) the current branch tip up to the target branch tip. This effectively combines the histories, since all of the commits reachable from the target branch are now available through the current one.

Before Merging



After a Fast-Forward Merge

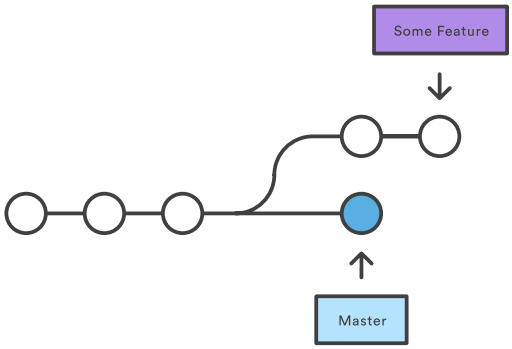



```
# Start a new feature
git checkout -b new-feature master
# Edit some files
git add <file>
git commit -m "Start a feature"
# Edit some files
git add <file>
git commit -m "Finish a feature"
# Merge in the new-feature branch
git checkout master
git merge new-feature
git branch -d new-feature
```

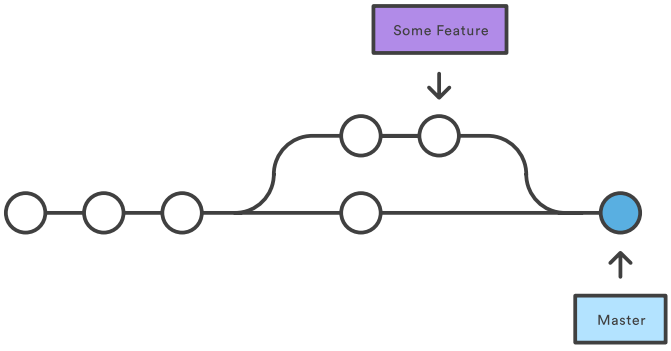
THREE WAY MERGE

However, a fast-forward merge is not possible if the branches have diverged. When there is not a linear path to the target branch, Git has no choice but to combine them via a 3-way merge.

Before Merging



After a 3-way Merge




```
Start a new feature
git checkout -b new-feature master
# Edit some files
git add <file>
git commit -m "Start a feature"
# Edit some files
git add <file>
git commit -m "Finish a feature"
# Develop the master branch
git checkout master
# Edit some files
git add <file>
git commit -m "Make some super-stable changes to master"
# Merge in the new-feature branch
git merge new-feature
git branch -d new-feature
```

MERGE CONFLICTS

If the two branches you're trying to merge both changed the same part of the same file, Git won't be able to figure out which version to use. When such a situation occurs, it stops right before the merge commit so that you can resolve the conflicts manually.

When Git encounters a conflict during a merge, It will edit the content of the affected files with visual indicators that mark both sides of the conflicted content. These visual markers are: `<<<<<<<`, `=======`, and `>>>>>>>`. Its helpful to search a project for these indicators during a merge to find where conflicts need to be resolved.

```
here is some content not affected by the conflict
<<<<<<< master
this is conflicted text from master
=======
this is conflicted text from feature branch
>>>>>>> feature branch;
```

Generally the content before the `=======` marker is the receiving branch and the part after is the merging branch.

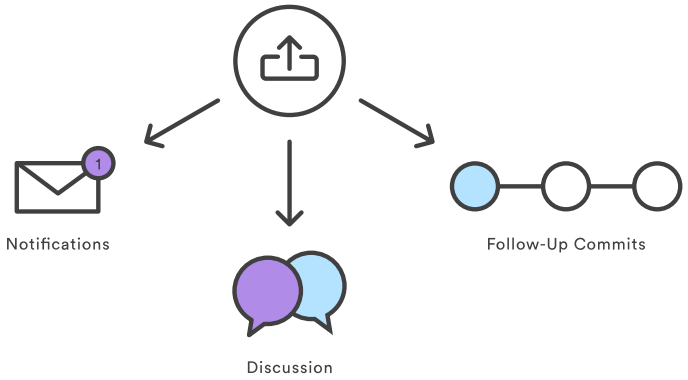
Once you've identified conflicting sections, you can go in and fix up the merge to your liking. When you're ready to finish the merge, all you have to do is run `git add` on the conflicted file(s) to tell Git they're resolved. Then, you run a normal `git commit` to generate the merge commit. It's the exact same process as committing an ordinary snapshot, which means it's easy for normal developers to manage their own merges.

MAKING A PULL/MERGE REQUEST

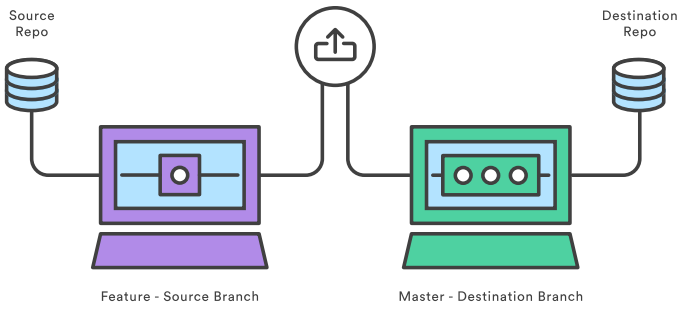
Pull requests are a feature that makes it easier for developers to collaborate using Bitbucket. They provide a user-friendly web interface for discussing proposed changes before integrating them into the official project.

In their simplest form, pull requests are a mechanism for a developer to notify team members that they have completed a feature. Once their feature branch is ready, the developer files a pull request via their Bitbucket account. This lets everybody involved know that they need to review the code and merge it into the master branch.

But, the pull request is more than just a notification—it's a dedicated forum for discussing the proposed feature. If there are any problems with the changes, teammates can post feedback in the pull request and even tweak the feature by pushing follow-up commits. All of this activity is tracked directly inside of the pull request.



When you file a pull request, all you're doing is requesting that another developer (e.g., the project maintainer) pulls a branch from your repository into their repository. Or, in other words, merges your branch into their repository. This means that you need to provide 4 pieces of information to file a pull request: the source repository, the source branch, the destination repository, and the destination branch.

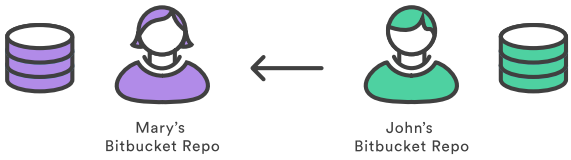


The general process is as follows:

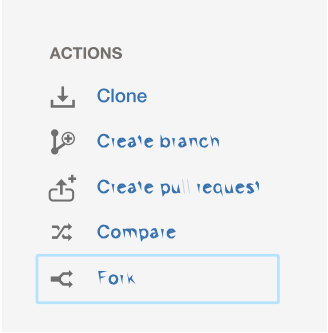
1. A developer creates the feature in a dedicated branch in their local repo.
2. The developer pushes the branch to a public repository.
3. The developer files a pull request.
4. The rest of the team reviews the code, discusses it, and alters it.
5. The project maintainer merges the feature into the official repository and closes the pull request.

EXAMPLE

In the example, Mary is a developer, and John is the project maintainer. Both of them have their own public Bitbucket repositories, and John's contains the official project.



To start working in the project, Mary first needs to fork John's Bitbucket repository.

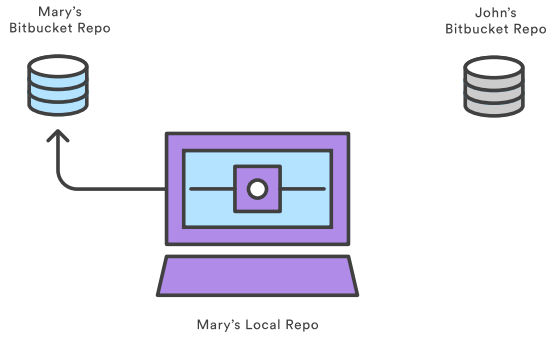


Next, Mary needs to clone the Bitbucket repository that she just forked. This will give her a working copy of the project on her local machine.

```
git clone https://user@bitbucket.org/user/repo.git
```

Before she starts writing any code, Mary needs to create a new branch for the feature. This branch is what she will use as the source branch of the pull request.

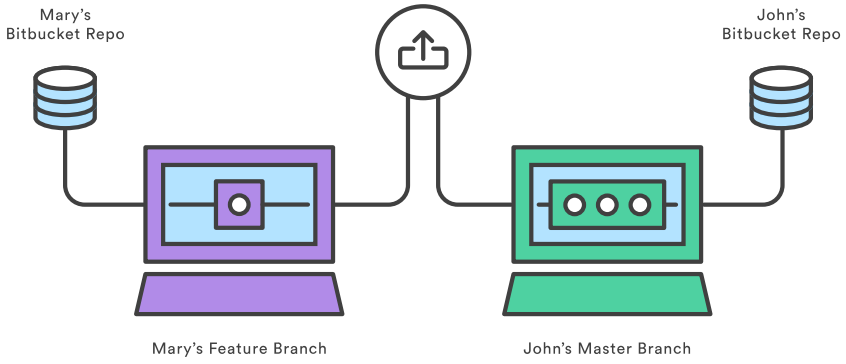
```
git checkout -b some-feature
# Edit some code
git commit -a -m "Add first draft of some feature"
```



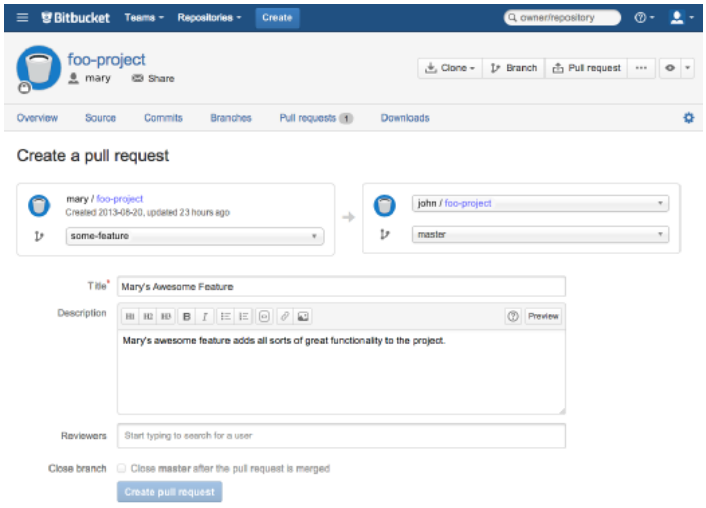
After her feature is complete, Mary pushes the feature branch to her own Bitbucket repository (not the official repository) with a simple git push:

```
git push origin some-branch
```

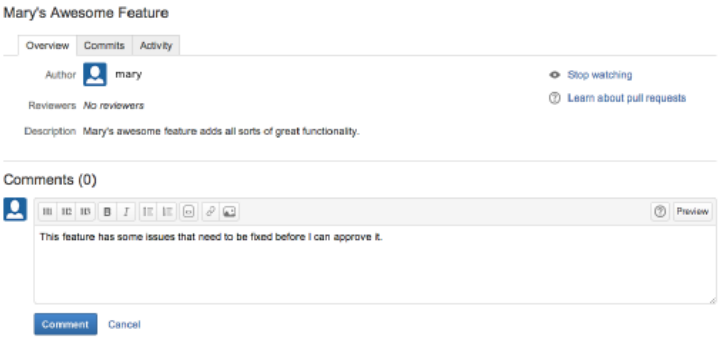
This makes her changes available to the project maintainer (or any collaborators who might need access to them).



After pushing her feature branch, she can create a pull/merge request.



John reviews the pull request:



Marry can fix/add/change code by pushing follow-up commits.

Once John accepts the pull request, he merges the branch into master.

WORKFLOW

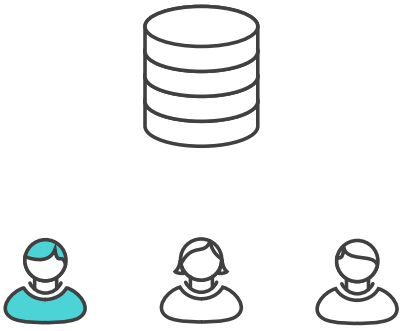
A Git Workflow is a recipe or recommendation for how to use Git to accomplish work in a consistent and productive manner. Git workflows encourage users to leverage Git effectively and consistently.

CENTRALIZED WORKFLOW

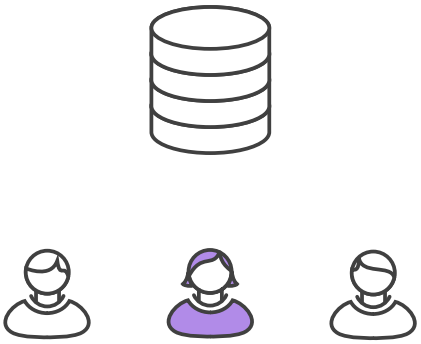
The Centralized Workflow uses a central repository to serve as the single point-of-entry for all changes to the project. The default development branch is called master and all changes are committed into this branch. This workflow doesn't require any other branches besides master.

EXAMPLE

John works on his feature: In his local repository, John can develop features using the standard Git commit process: edit, stage, and commit.

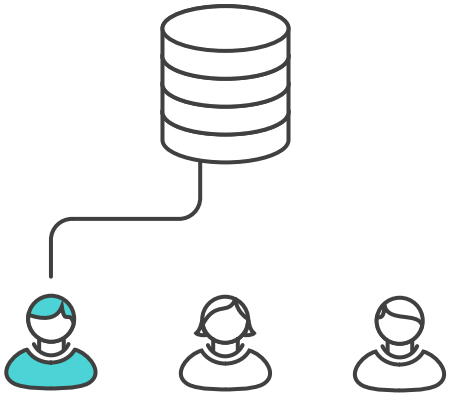


Mary works on her feature: Meanwhile, Mary is working on her own feature in her own local repository using the same edit/stage/commit process. Like John, she doesn't care what's going on in the central repository, and she really doesn't care what John is doing in his local repository, since all local repositories are private.



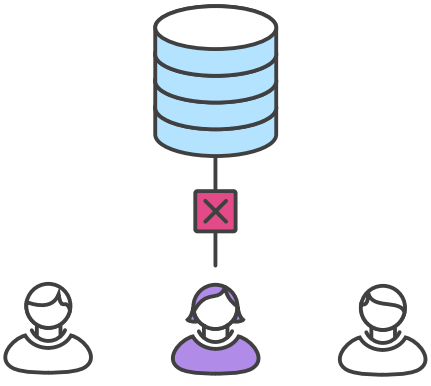
John publishes his feature: Once John finishes his feature, he should publish his local commits to the central repository so other team members can access it.

```
git push origin master
```



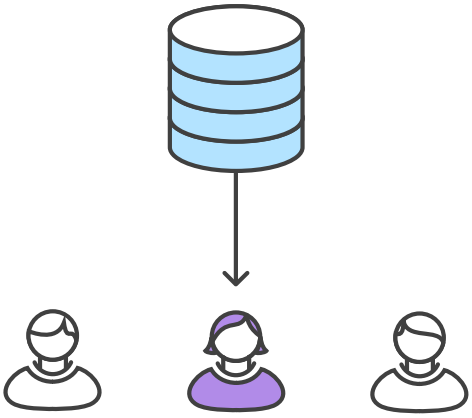
Mary tires to publish her feature: But, since her local history has diverged from the central repository, Git will refuse the request.

```
git push origin master
error: failed to push some refs to '/path/to/repo.git'
hint: Updates were rejected because the tip of your current branch is behind
hint: its remote counterpart. Merge the remote changes (e.g. 'git pull')
hint: before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

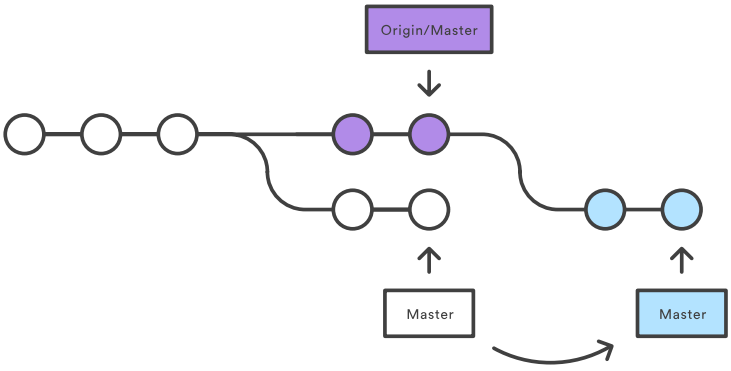


Mary rebases on top of John's commit(s): Mary can use git pull to incorporate upstream changes into her repository.

```
git pull --rebase origin master
```

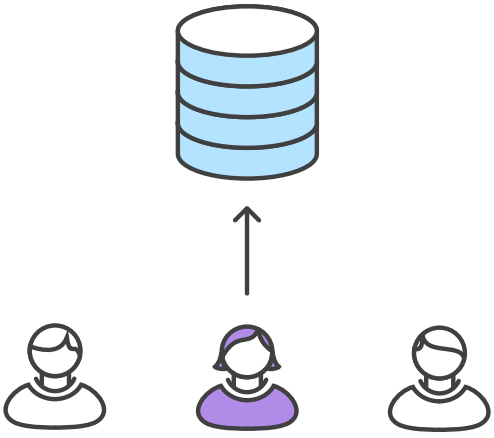


Mary's Repository



Mary successfully publishes her feature:

```
git push origin master
```



FEATURE BRANCH WORKFLOW

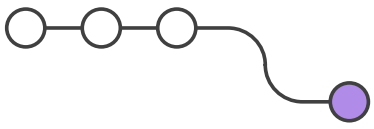
The core idea behind the Feature Branch Workflow is that all feature development should take place in a dedicated branch instead of the master branch. This encapsulation makes it easy for multiple developers to work on a particular feature without disturbing the main codebase. It also means the master branch will never contain broken code, which is a huge advantage for continuous integration environments.

HOW IT WORKS

1. **Start with the master branch:** All feature branches are created off the latest code state of a project.
   ```
   git checkout master
   git pull origin master
   ```
2. **Create a new branch:** Use a separate branch for each feature or issue you work on.
   git checkout -b new-feature
3. **Update, add, commit, and push changes:** On this branch, edit, stage, and commit changes in the usual fashion, building up the feature with as many commits as necessary. Work on the feature and make commits like you would any time you use Git.
   git status
   git add <some-file>
   git commit
4. **Push feature branch to remote:**
   git push -u origin new-feature
5. **Optional:** get feedback / discuss / pull/merge requests
6. **Merge into master**

EXAMPLE

Mary begins a new feature in a new branch:

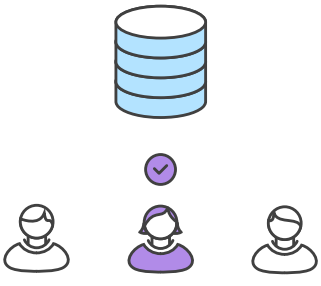


```
git checkout -b marys-feature master
```

Mary codes:

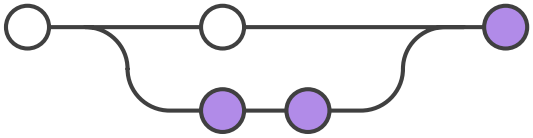
```
git status
git add <some-file>
git commit
```

Mary finishes her feature:



```
git push
```

Bill or Mary merge the feature into the master branch:



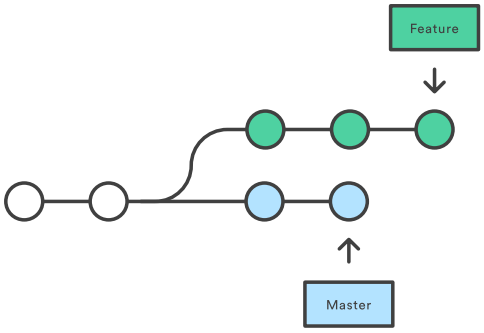
```
git checkout master
git pull
git pull origin marys-feature
git push
```

This process often results in a merge commit. Some developers like this because it's like a symbolic joining of the feature with the rest of the code base. But, if you're partial to a linear history, it's possible to rebase the feature onto the tip of master before executing the merge, resulting in a fast-forward merge.

MERGE VS. REBASE

The first thing to understand about git rebase is that it solves the same problem as git merge. Both of these commands are designed to integrate changes from one branch into another branch.

A forked commit history



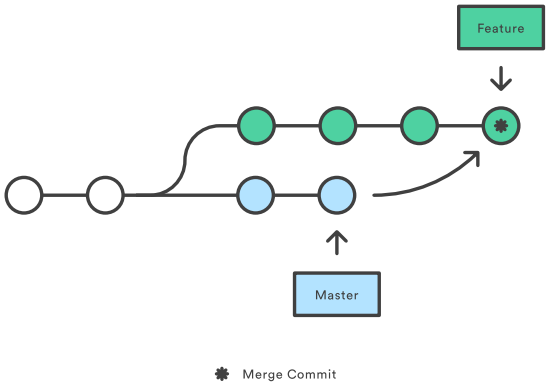
THE MERGE

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

```
git checkout feature
git merge master
```

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

Merging master into the feature branch

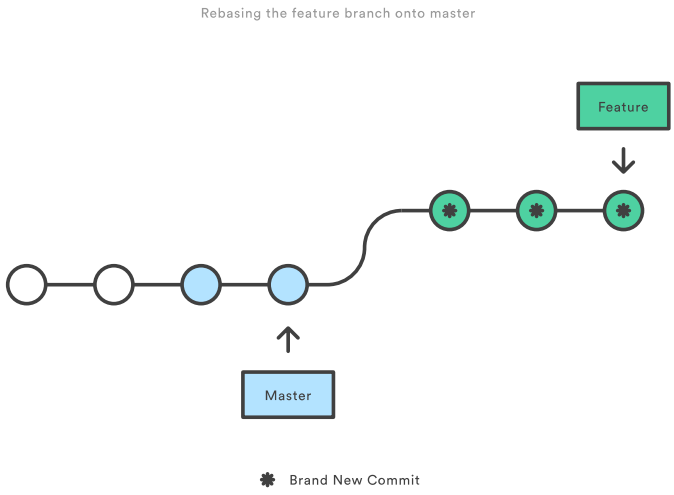


THE REBASE

As an alternative to merging, you can rebase the feature branch onto master branch using the following commands:

```
git checkout feature
git rebase master
```

This moves the entire feature branch to begin on the tip of the master branch, effectively incorporating all of the new commits in master. But, instead of using a merge commit, rebasing re-writes the project history by creating brand new commits for each commit in the original branch.



BEST PRACTICES

- **Commit often**: Commits are cheap and easy to make. They should be made frequently to capture updates to a code base. Each commit is a snapshot that the codebase can be reverted to if needed. Frequent commits give many opportunities to revert or undo work.
- **Ensure you're working from latest version**: SCM enables rapid updates from multiple developers. It's easy to have a local copy of the codebase fall behind the global copy. Make sure to git pull or fetch the latest code before making updates. This will help avoid conflicts at merge time.
- **Make detailed notes:** Each commit has a corresponding log entry. At the time of commit creation, this log entry is populated with a message. It is important to leave descriptive explanatory commit log messages. These commit log messages should explain the "why" and "what" that encompass the commits content. These log messages become the canonical history of the project's development and leave a trail for future contributors to review.
- **Review changes before committing**: SCM's offer a 'staging area'. The staging area can be used to collect a group of edits before writing them to a commit. The staging area can be used to manage and review changes before creating the commit snapshot. Utilizing the staging area in this manner provides a buffer area to help refine the contents of the commit.
- **Use Branches:** Branching is a powerful SCM mechanism that allows developers to create a separate line of development. Branches should be used frequently as they are quick and inexpensive. Branches enable multiple developers to work in parallel on separate lines of development. These lines of development are generally different product features. When development is complete on a branch it is then merged into the master line of development.
- **Agree on a Workflow:** By default SCMs offer very free form methods of contribution. It is important that teams establish shared patterns of collaboration. SCM workflows establish patterns and processes for merging branches. If a team doesn't agree on a shared workflow it can lead to inefficient communication overhead when it comes time to merge branches.

LICENSE