Contents

1 Creation **2**
- 1.1 How to create a project using Git . . . 2
  - 1.1.1 Connect to a Remote Repository . . . 2
  - 1.1.2 Changing upstream . . . 2
  - 1.1.3 How to clone a repository . . . 2

2 The Basics **3**
- 2.1 File states . . . 3
  - 2.1.1 File tracking categories . . . 3
  - 2.1.2 Lifecycle States of Tracked Files . . . 3
- 2.2 Commit Lifecycle . . . 4
  - 2.2.1 File States . . . 4
  - 2.2.2 Tracking and Staging . . . 5
  - 2.2.3 Committing staged changes . . . 5
  - 2.2.4 Pushing to Remote . . . 5
- 2.3 Mistaken commit . . . 5
  - 2.3.1 Undo when pushed . . . 6

3 Working with Remotes **7**
- 3.1 Remotes . . . 7
- 3.2 Fetch & Pull . . . 7
- 3.3 push . . . 7
- 3.4 Renaming and Removing Remotes . . . 7

4 Branch Management **8**
- 4.1 Branching . . . 8
- 4.2 Branch vs. Tag . . . 8
  - 4.2.1 Taking changes to new branch . . . 9

5 Branch Integration **10**
- 5.1 Merge . . . 10
  - 5.1.1 What way does one merge using the base command? . . . 10
  - 5.1.2 Two people working on one branch . . . 10
  - 5.1.3 Undoing a merge . . . 10
- 5.2 Rebase . . . 11
  - 5.2.1 Rebasing up to a certain commit - step by step allowing for changes . . . 11

6 Solving Conflicts **12**
- 6.1 Visual Studio . . . 12

7 Tracking Path Changes **13**
- 7.1 Tracking Path Changes . . . 13
  - 7.1.1 Removing files locally . . . 13
  - 7.1.2 Removing files from remote . . . 13
  - 7.1.3 Remove tracking but keep locally . . . 13
  - 7.1.4 Batch removal . . . 13
  - 7.1.5 Renaming Files . . . 13
  - 7.1.6 Remove Remote File after local changes . . . 14

8 Git Configurations **15**
- 8.1 Git Aliases . . . 15

01 Chapter

Creation

1.1 How to create a project using Git

Create a project locally. Then run `$git init` – which creates a repository in the folder you are currently in. Alternativley you can use `$git init <name>` –which basically does `$mkdir <name>` and `$git init <name>` in one line and creates a folder in the directory you are in.

1.1.1 Connect to a Remote Repository

Once you have a git project set up locally, you will want to connect it to a remote host. Use the Git hosting service of your choosing and create a project. Then use the *url* (either SSH or HTTP) and connect it to your local.

```
$git remote add origin <url>
$git branch -M main
$git push -u origin main
```

1.1.2 Changing upstream

If you made a mistake or want to take part of the project to a different remote you can use this to set a new upstream location:

```
$git remote set-url origin <url>
```

1.1.3 How to clone a repository

Sometimes you want to work on an already existing project.

```
$git clone <url>
```

This creates a repository in a new dir in your cwd named after the project in your Git hosting service.
`$git clone <url> myFolder` Does the same, but replaces the project name with a specific one you set.
`$git clone <url> .` Creates the repository in your cwd.

02

Chapter

The Basics

2.1 File states

Every file in Git exists in a defined state — describing whether it is tracked and how far its changes have progressed in the versioning process.

2.1.1 File tracking categories

Each file in your working directory can be in one of two states: **tracked** or **untracked**. Tracked files are files that Git is aware of and were either in the last snapshot[1] or are newly staged files. Tracked files can be *unmodified*, *modified*, or *staged*.
While git knows about all files in your working directory, it will only *watch* and record the history of **tracked** files.

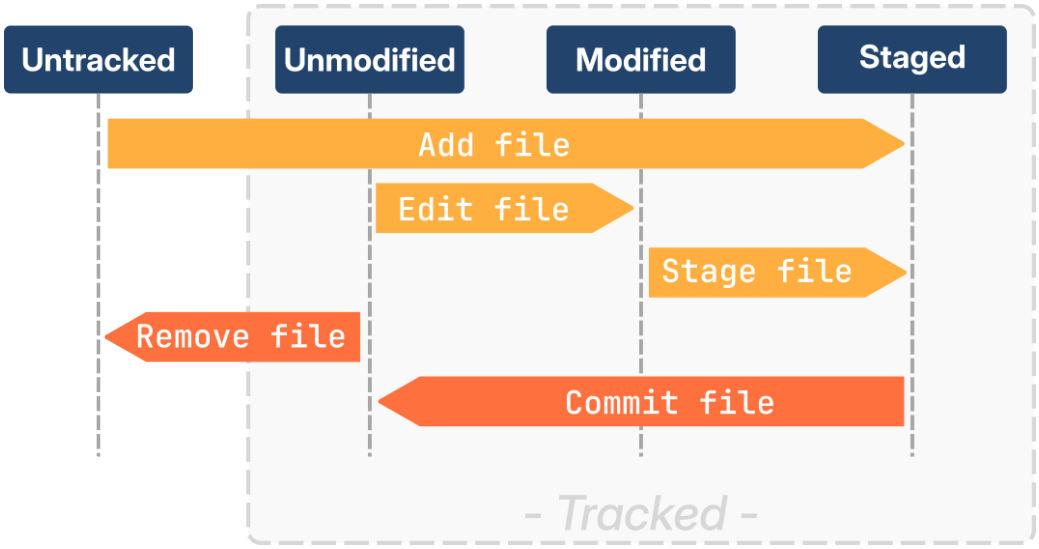

Figure 2.1: illustrates how files move through the different tracking categories.

2.1.2 Lifecycle States of Tracked Files

GGit records your work through several *storage stages*, each representing a deeper level of permanence:

1. **Stashed** — work in progress is temporarily stored aside, allowing you to return to a clean working state.
2. **Modified** — a tracked file has been changed in your **working directory** but not yet staged.
3. **Staged** — you have marked a modified file in its current version to go into your next *commit snapshot*.

4. **Committed** — the snapshot has been saved to your **local repository**.
5. **Pushed** — the commit has been transferred to a **remote repository**.

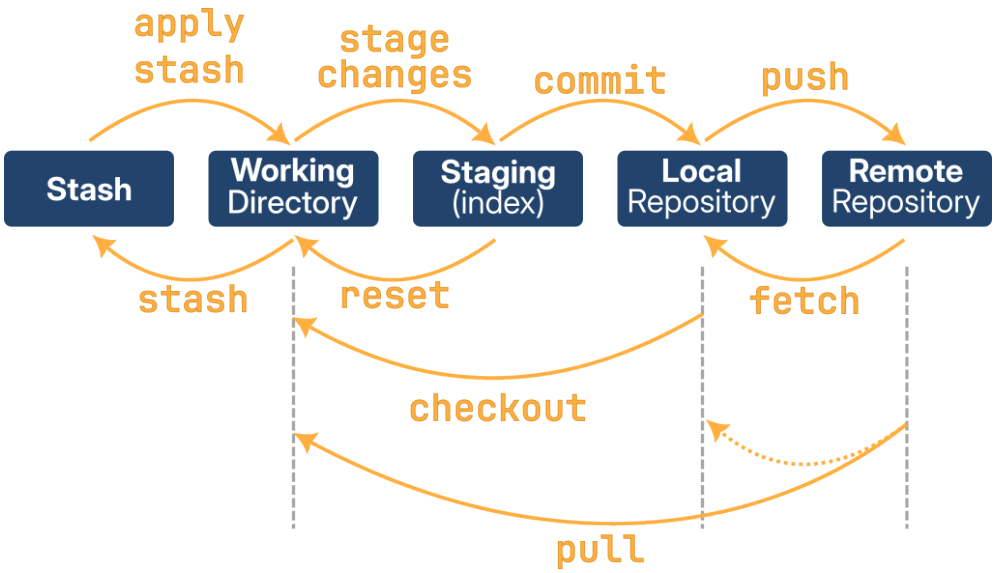

Figure 2.2: illustrates how files move through these stages during normal Git operations.

2.2 Commit Lifecycle

The commit lifecycle begins by checking the state of files, then adding the desired files to the staging area, and finally committing them to the repository. Commits can also be pushed to a remote repository.

2.2.1 File States

To check the states of our files we can use:

```
$git status
```

Adding the *short* flag[2] gives a more concise output, if the regular one is too long. The `$status -s` and `$tsatus --short` syntax are synonymous to one another.

M	modified but not staged
M	modified and staged
MM	modified, staged and then modified again
A	new files added to staging area
? ?	new files that aren't tracked

Table 2.1: Shorthand symbol meaning

If *git status* is too vague, use `$git diff` it shows changes between your working directory and the staging area. → **unstaged changes**
`$git diff --staged` and `$git diff --cached` [3] will shows changes between the staging area and the last commit. → **staged changes**

If you have a newly untracked directory with a bunch of untracked files `$git diff` will only show you the untracked directory, not all the files inside that aren't tracked either. To see these you can `$dit diff -uall`, which is short for `$git diff --untracked-files=all`.

2.2.2 Tracking and Staging

Track and stage new files using:

```
$git add <file>
```

The git add command takes a path name for either a file or a directory; if it's a directory, the command adds all the files in that directory recursively.

2.2.3 Committing staged changes

When the staging area is set up the way you want it, you can commit your changes with `$git commit`. This will however open a text editor for you to enter your commit message and if using the `$-v` verbose tag, will display more information on what will be commited.
Becasue we are lazy we can use:

```
$git commit -m "message"
```

This allows us to skip the editor step and type up our comment in the editor itself.

If you want to skip the staging are, one can add the `$-a` flag. This will also add all tracked files.

2.2.4 Pushing to Remote

If you are working with a remote repository, you will want to push your changes to it using `$git push`.

2.3 Mistaken commit

If you have made a mistake or forgot something either undo the changes or amend the commit.

1. `$git add` any files you forgot, and then:

   ```
   $git     commit --amend -m "your message"
   ```

   This will amend your commit. If you have already pushed you will need to `$git push --force-with-lease`.
   Caution: This rewrites commit history, because the amended commit gets a new hash! If others have already pulled the old commit, they'll need to sync manually.

2. This will simply reset your commit to right before you made it:[4]

   (a) Undo last commit but keep changes staged

   ```
   $git reset --soft HEAD~1
   ```

(b) Undo last commit and unstage changes

```
$git reset --mixed HEAD~1
```

(c) Undo last commit and discard changes

```
$git reset --hard HEAD~1
```

2.3.1 Undo when pushed

```
$git revert <commit>
```

Where <commit> is the commit hash. Use `$git log` or `$git log --oneline`. `$git --no-pager log` doesn't leave you in the pager, which you would normally equit with **q**.

[1] A Snapshot is your last commit, see Glossary for more details.

[2] A flag is a parameter you pass to a Git command to modify its behavior. There are short (-s) and long (--short) flags. Functionally *most* Git commands treats them the same.

[3] --chaced is older terminology

[4] HEAD points to your current commit (the one your working directory is based on)

Chapter 03

Working with Remotes

3.1 Remotes

- What is a remote: - showing Remotes - adding remotes + shortname

3.2 Fetch & Pull

git fetch updates your local repository's knowledge of the remote, not your working branch.
fetch -> make sure your local Git knows about the latest main on the server.

Git pull syntax:
git pull <remote> <branch>
For pulling into a branch while on a different branch:
git fetch origin main:main
git fetch <remote> <src>:<dst>
- <remote> → which remote to fetch from (e.g. origin).
  - <src> → branch (or ref) on the remote.
  - <dst> → branch (or ref) in your local repo to update.

3.3 push

3.4 Renaming and Removing Remotes

04

Chapter

Branch Management

4.1 Branching

To create one locally you use the command:
$git checkout -b my-feature
This combines both:
$git branch my-feature → creates the branch
$git checkout my-feature → switches to it

After that it needs to be pushed to the remote via:
$git push -u origin my-feature

4.2 Branch vs. Tag

Differences between Git tags and branches

Tags and branches are both used for version control of your code base. Their functions complement each other, and they are designed to be used together.

A branch is often used for new features and fixing bugs. It allows you to work without impacting the main code base. Once your work is complete, you can incorporate your changes into the application by merging the branch back into the main code base. This allows multiple people to work on different aspects of the project simultaneously. It also provides a way to experiment with new ideas without risking the main code base's stability.

Unlike branches, tags are not intended for ongoing development. Tags mark a specific point in the repository's history to give developers an easy way to reference important milestones in the development timeline. When to use branches

Imagine that you want to add a new feature to a software project's code base but you aren't sure if the new feature will work as expected. You want to experiment without affecting the main code base.

In this scenario, use a Git branch to create a separate line of development for the new feature. This avoids affecting the main code base. Once the feature is complete and tested, you can merge the branch back into the main code base. Depending on your organization's process, you may choose to delete the branch to prevent clutter. When to use tags

Now, suppose that you complete and test the new feature. You want to release the new software version to the users. In this case, use a Git tag to mark the current state of the code base as a new version release.

You can name the tag to reflect the version number (such as v1.2.3) and include a brief description of the changes in the release. This allows you to easily reference the specific version of the released code base. It is also easy to roll back to the previous version if necessary. Using

4.2.1 Taking changes to new branch

Every now and then you may be working on a branch and have an idea. Start working on that idea and then realize that perhaps this idea should be its own branch. When this happens you can move your worked on files into a new branch. Here it's important to distinguish between

1. changes have been committed
2. changes are unstaged

No commits

1. Stash your changes: [1]

```
$git git stash push -m "work in progress"
```

2. Switch to the branch you want to branch off from: `$git switch <branch>`
3. Create a new branch and switch to it `$git switch -c <new_branch>`
4. Apply your stashed work: `$git stash pop`

Previous Commits

1. Stay on the wrong branch, note the last commit hash: `$git log`
2. Switch to main, create the new branch:

```
$git     git switch <branch>
$git     git switch -c <new_branch>
```

3. Cherry-pick your commits:

```
$git     git cherry-pick <commit-hash>
```

[1] `$git stash` is just a shorthand for `$git stash push`. It does however allow us to use flags like `$-m`

05

Chapter

Branch Integration

Whether you are working colaborativley with others and want to integrate thei work into whatever it is you are doing or you have finished working on a feature for you project and now want to integrate it into the rest of what you've built you will need to combine the two branches. You have two options when wanting to do this – you can either *merge* or *rebase*.
Generally you will **merge** your branches together. This combines two branches by creating a merge commit that ties their histories together. It preserves the true branching history. Any conflicts that may arrise are resolved at once.
Rebasing on the other hand reapplies your branch's commits on top of another branch, rewriting history. It produces alinear history, as tough your work had always been based on that initial snapshot. Conflicts are handles one commit at a time.

5.1 Merge

To merge a hotfix branch back into your master branch to deploy to production. You do this with the git merge command:

```
$git checkout master
$git merge hotfix
```

5.1.1 What way does one merge using the base command?

All you have to do is check out the branch you wish to merge into and then run the `$git merge` command

5.1.2 Two people working on one branch

If p1 pushes changes to a remote, and p2 has local changes and also wants to push they need to merge first. They can do this with `$git pull`, however this leads to a messy history. Ideally p2 would run

```
$git pull --rebase
```

This ensures a clean history, and p2 can then push their changes to the remote. If they still have unstaged changes in their working directory, these must be stashed first.

5.1.3 Undoing a merge

1. not commited: `$git merge --abort`
2. Commmited but not pushed `$git reset --hard HEAD~1`

3. pushed and commited `$git revert -m 1 <merge_commit_hash>`

5.2 Rebase

Why would you do this though?

> Often, you'll do this to make sure your commits apply cleanly on a remote branch – perhaps in a project to which you're trying to contribute but that you don't maintain.

NOTE: Do not rebase commits that exist outside your repository and that people may have based work on.

Think of git rebase as lifting your commits off your current branch and then replaying them one by one on top of a new base (in our case, the updated main).
- main has new commits from person A.
- You are working on a feature branch that has commits of its own.
- Rebase takes your commits, temporarily removes them, updates your branch to look like main.
- Then tries to apply your commits on top, as if you branched off after person A's changes were already made.

This gives you a clean, linear history.

When you run `$git rebase main`, Git takes all your commits (from where your branch diverged from main) and replays them one by one on top of the updated main.

If Git tries applying a commit that touched a file that also changed in main, you get a conflict.

Once you fix it and `$git rebase --continue`, Git will move on to the next commit, and so on, until it finally reapplies your most recent work.

So:

You're not "losing" your latest changes — they just haven't been replayed yet.

You'll resolve conflicts in the order your commits were originally made.

At the end of the rebase, your branch will contain all your commits, sitting neatly on top of the updated main.

```
$git push --force-with-lease
```

must be used afterwards, because the new commit history is different than the one on the remote after rebasing.

5.2.1 Rebasing up to a certain commit - step by step allowing for changes

```
$git git rebase -i <commit hash>
```

06 Chapter

Solving Conflicts

6.1 Visual Studio

In Visual Studio you need to go to the Git Changes window and from there you should see Merge Conflicts

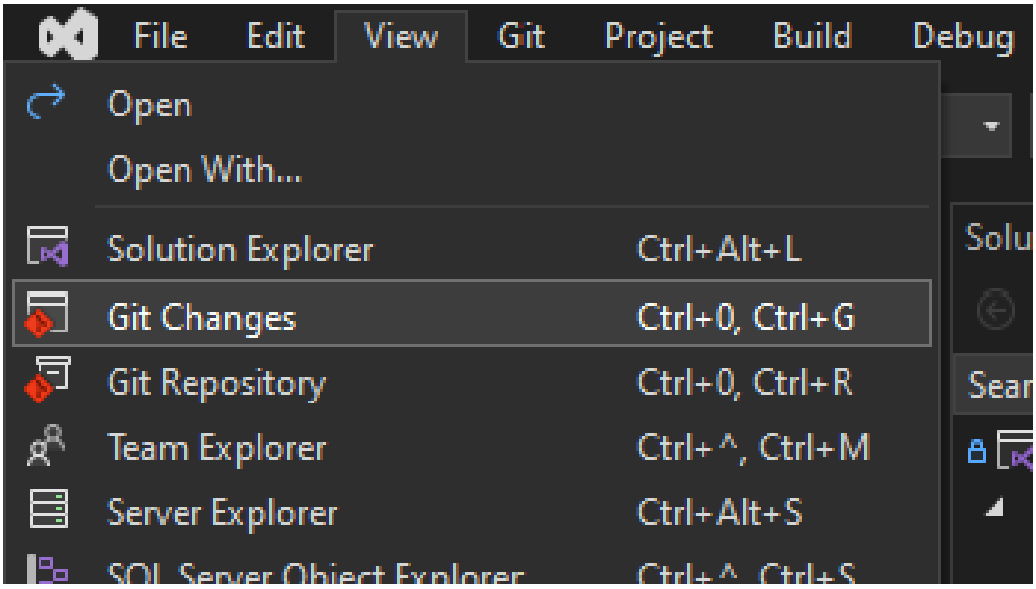

Figure 6.1: where to find Git Changes menu

using the editor you can then resolve these conflicts.
Once you have resolved them you can close VS and then stage and commit all your changes.

Chapter 07

Tracking Path Changes

7.1 Tracking Path Changes

7.1.1 Removing files locally

To remove a file from Git, you have to remove it from your tracked files (more accurately, remove it from your staging area) and then commit. The `$git rm` command does that, and also removes the file from your working directory so you don't see it as an untracked file the next time around.
This is different than just doing `$rm <file>`, because then git will just mark the file as "Changes not staged for commit"

7.1.2 Removing files from remote

7.1.3 Remove tracking but keep locally

```
$git rm --cached <file>
```

Keeps the file in your working tree but removes it from your staging area.

7.1.4 Batch removal

```
$git rm log/\*.log
```

Removes all log files in the log/ directory.

7.1.5 Renaming Files

Unlike many other VCSs, Git doesn't explicitly track file movement.
If you want to rename a file you can do:

```
$git mv <file_from> <file_to>
```

This works the same way as the Linux command `$mv dir1 dir2` In this scenario if the dir2 directory exists, the command will move dir1 inside dir2. If dir2 doesn't exist, dir1 will be renamed to dir2.
The same happens with `$git mv dir1 dir2`, the only difference is that the changes are staged automatically.

[0] we need to escape the * character

7.1.6 Remove Remote File after local changes

Chapter 08

Git Configurations

8.1 Git Aliases

Command List

SETUP

Download an existing repository

```
$ git clone <name>
```

Create new Local repository in current directory

```
$git init
```

Create new Local repository in *project name* directory

```
$git init <project name>
```

Add upstream for local repository.

```
$ git remote add origin <link to Git hosting service>
```

SNAPSHOT

Download an existing repository

```
$ git status
```

Create new Local repository in current directory

```
$git diff
```

- staged	staged changes vs last commit
HEAD	working directory vs last commit

BRANCHES & TAGS

Show local branches

```
$ git branch
```

- a	also shows remote branches
- v	additionally shows last commit on each branch
<*newbranch*>	creates a new branch based on curretn HEAD
- d <*branch*>	deletes a local branch

Switch HEAD branch

```
$git switch <branch>
```

- track origin/feature-x	creates local branch feature-x that tracks origin/feature-x. **NOTE**: It does not create a remote branch!
- c <*newbranch*>	create a new branch and switch to it

NOTE: *git checkout* can perform these tasks as well, however works slightly differently and is generally overloaded with a bunch of other features.

Mark the current commit with a tag

```
$ git tag <tag-name>
```

Push a tag to a remote

```
$ git push origin tag <tag-name>
```

LOCAL CHANGES

adds files to be tracked

```
$ git add <file>
```

.	adds all untracked files
-p <file>	???

Commit changes and add a message

```
$git commit -m "<message>"
```

-am "<message>"	add & commit in one go
-ammend	change something in a commit ???

Git remote add <alias> <url> git fetch <alias> Git fetch <alias>/<branch> git pull git push <alias>/<branch>

MERGE & REBASE

merge current branch into target branch???

```
$ git merge <branch>
```

Change the history...

```
$git rebase <branch>
```

Git stash Git stash list Git stash apply Git stash pop Git stash drop

TRACKING PATH CHANGES

removes stated file

```
$ git rm <file>
```

???

```
$git mv <stig-path> <new-path>
```

UNDO

???

```
$ git reset <file>
```

???

```
$git reset –hard <file>
```

Here is some information text inside a styled box. Line breaks and math also work: $E = mc^2$.

Glossary

Snaphsot - Every time you perform a commit, you're recording a snapshot of your project that you can revert to or compare to later.
Flag - A flag (also called an option or switch) is a parameter you pass to a Git command to modify its behavior.