

# Git Fundamentals

by Nicola Orecchini



# What you'll find in this presentation

## Chapter

## Content

### 1. What's Git?

High-level introduction to Git

### 2. How does Git work?

Overview of Git core components

### 3. Basic Git workflow

Illustration of basic usage of Git in a project lifecycle

### 4. Useful use-cases

Tested solutions to recurring tasks performed with Git

# 1. What's Git?

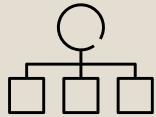
# Git is a **Version Control System**, a tool that helps you keep track of changes to your project throughout its lifecycle



## Version tracking

Keeps a complete and detailed **history of changes to files**, including:

- who did it
- what lines of code exactly changed
- when it changed
- why it was changed (a commit message is mandatory)



## Branching & Merging

- **Branching:** allows you to duplicate the source code for yourself and work on your local copy, without affecting the source code
- **Merging:** when you're done and your code passes all tests, it allows you to join your branch with the original source code



## Backup

- Allows to revert to **previous versions** of the code
- Can **restore lost or corrupted files**



## Collaboration

- Allows **multiple users** to **work on the same code**, without interfering with each other
- **Changes by different users** can be **merged together** after an approval, which can accept all, some, or none of the changes

# Git, GitHub, GitLab

			
<i>What is it?</i>	It's a <b>source code version control system</b> – not the only one, but the <b>most popular</b>	They're <b>Git repository<sup>1</sup> hosting services</b> , i.e., a <b>remote storage location</b> for <b>Git repositories</b> – again, not the only two but the <b>most popular</b>	
<i>Ok, now tell me what it really is</i>	It's a <b>software</b> that can run both <b>locally</b> on your PC and in the <b>cloud</b>	It's a <b>service</b> (accessible through a website) that <b>hosts</b> your <b>Git repository</b> in the <b>cloud</b>	
<i>And why should I use it?</i>	It manages your code base <sup>2</sup> providing the features described in the previous slide	They <b>facilitate Git usage</b> by providing <b>custom web UI's</b> to make it easier to browse and interact with Git repositories, e.g. track issues, perform code reviews, etc.	
<i>Can you make an example?</i>	You are working on a coding project on your laptop. You use Git commands (git add, git commit, git log) to save versions of your code and track changes locally	You push your local Git repository to GitLab. Your team members can then review your code with an UI, open issues, suggest changes, and automatically deploy updates	

1. Git Repository = the set of files created and used by Git to track different versions, changes, etc.

2. Code Base = the set of code files created by you and your team that make up your project

## 2. How does Git work?

# Git is based on 2 main components: commit & branch

## Commit

A commit is the location where the code and its changes are stored. It's the equivalent of saving a file and renaming it "v2", "v3" and so on



## Branch

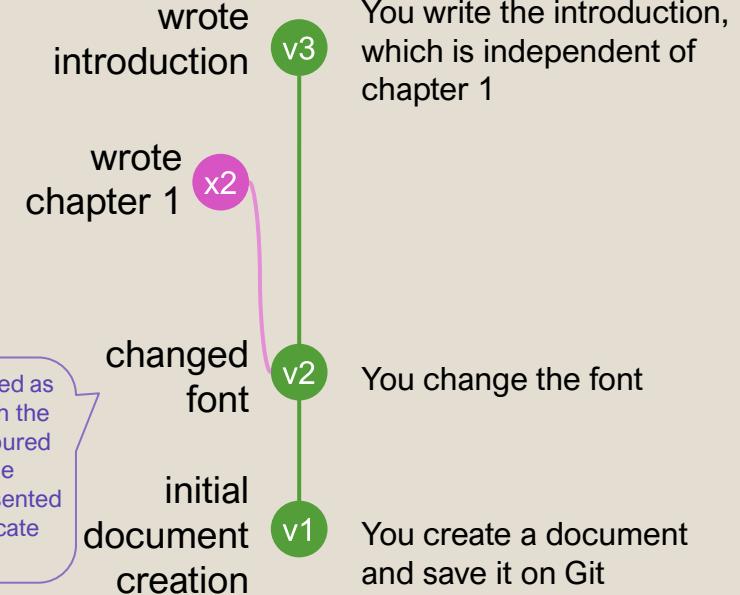
A branch is a representation of different isolated versions of code

Branch "V"

Branch "X"

Your colleague starts writing Chapter 1. To do this, she creates a **new branch**. Why? Because if she accidentally deletes something in the rest of the document, changes won't affect the main version

Conventionally, a branch is represented as a line connecting dots (commits). Both the line and the dots on it are usually coloured with the same color to indicate a single branch. Different branches are represented as differently coloured lines that bifurcate from a commit



By storing each commit, Git maintains the **history of changes** to our document, allowing us to **examine/restore** any previous versions

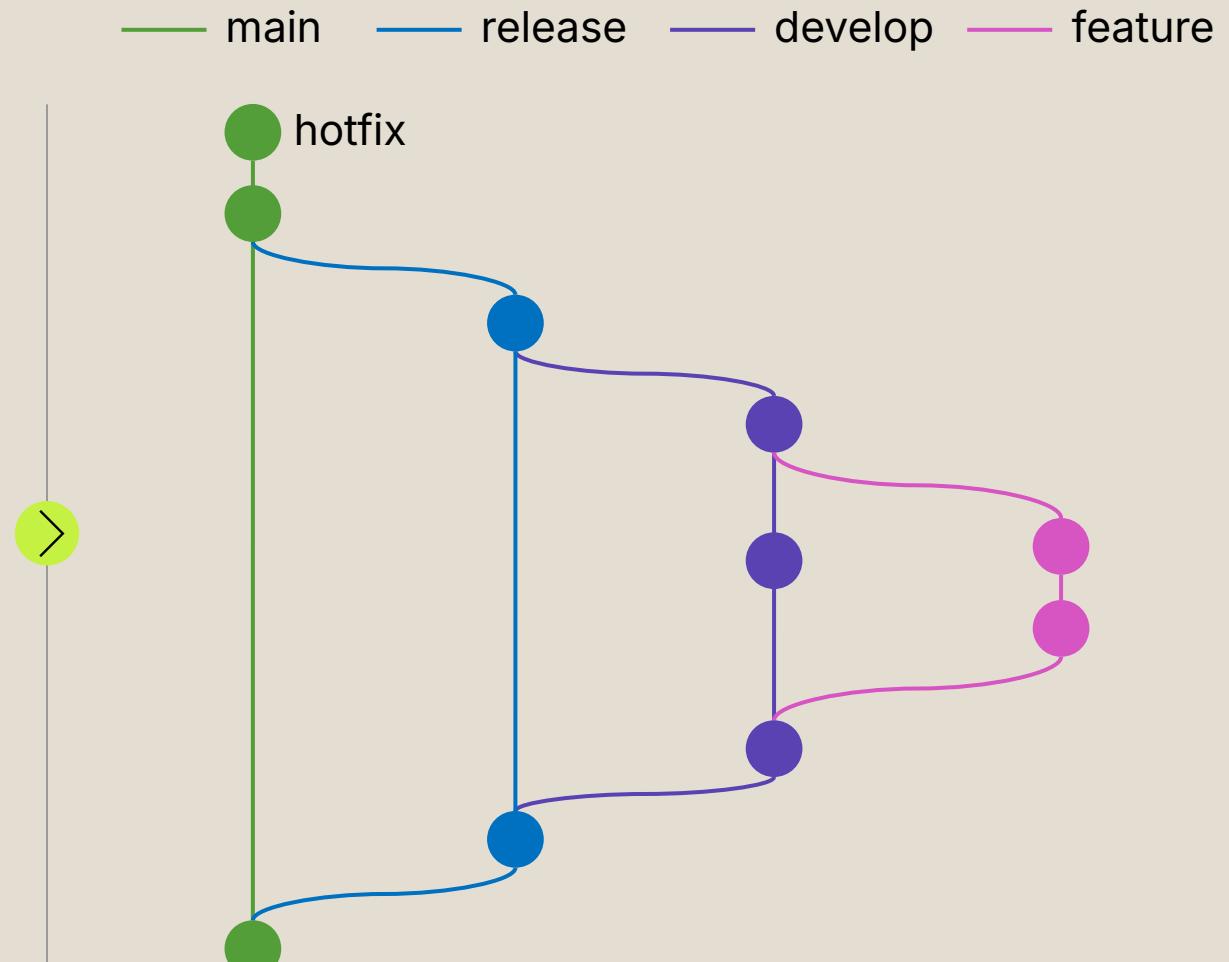
By organizing commits in branches, Git **allows multiple users to work on the same document** without worrying about conflicts

# Combine commits & branch to version-control your project

In a corporate setting, where multiple teams must work together and things must be formal, the branching model has some clear policies and is structured along 5 branches:

- main: stable version in production – all that is here can be deployed
- develop: integration branch – features ready to be deployed are merged here before deploying
- feature/\*: temporary branches to develop new features - for every new feature
- hotfix/\*: urgent fixes in production
- release/\*: prepare stable version

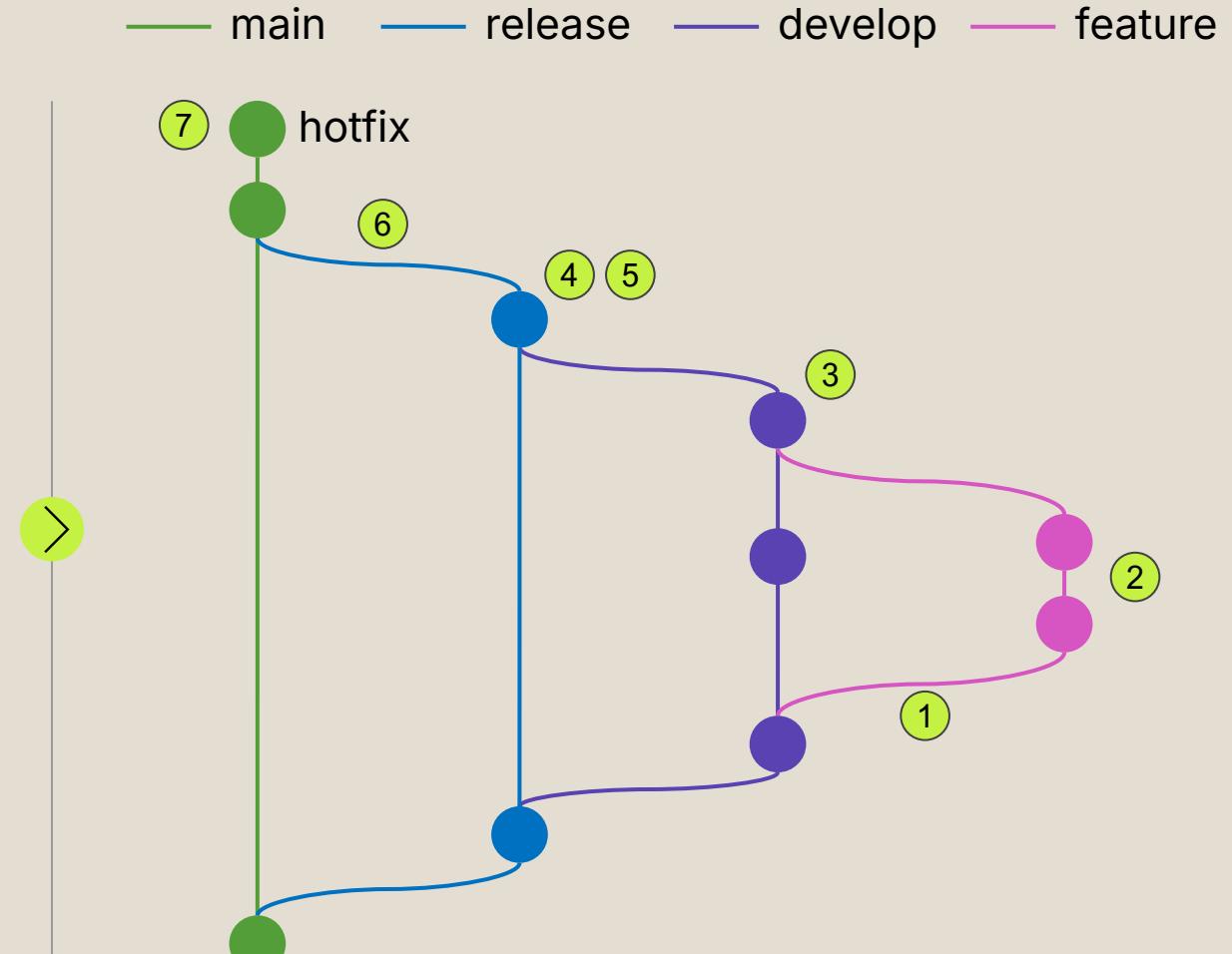
\*The slash (/) in the name is only a convention to organize names. Git doesn't create folders, it's only for visual clarity



# Having a clearly defined branching structure facilitates work

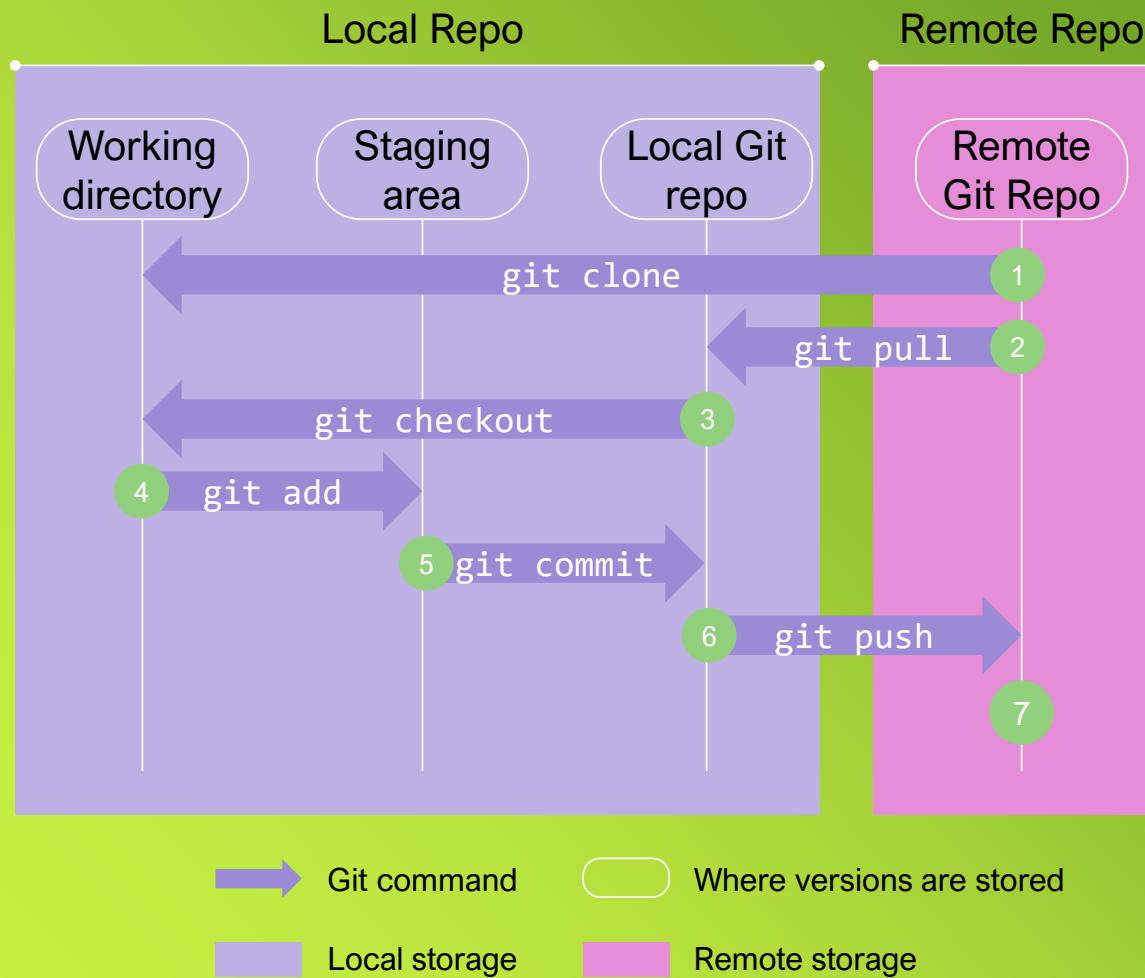
Workflow for the creation of a new feature:

- ① Create new branch «/feature-xyz» from develop
- ② Code the feature and push
- ③ Open merge request into develop. If accepted, feature is merged into develop
- ④ Release: create new branch release/x.y from develop
- ⑤ Test and fix feature in release
- ⑥ Merge release into main (and in develop, if needed)
- ⑦ If a quick fix is needed, create new branch hotfix/x from main, then merge on main and in develop



# 3. Basic Git workflow

# How a typical development workflow looks like with Git

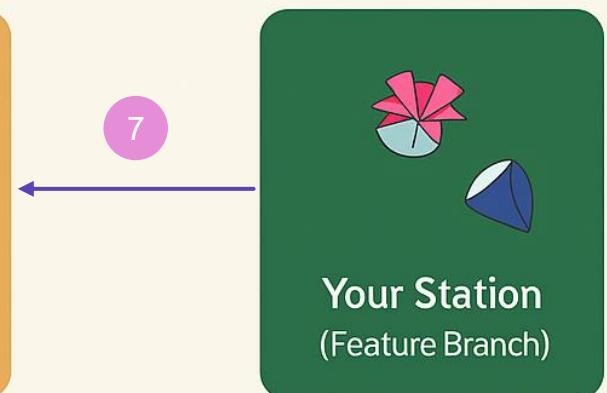


- 1 **git clone**  
Clone the repository to your local machine
- 2 **git pull origin master**  
Pull down from the remote repository to make sure you have up to date information. Changes could have been made even in the time you were cloning the repo
- 3 **git checkout -b new\_branch\_name**
  - Create your own branch to have your own place to experiment with code without immediately affecting changes on the master branch
  - You might not need to create a new branch if you are intentionally editing an existing branch. In that case, be sure you are on the correct branch and not the master prior to making any changes
- 4 **git add file\_name**  
After your code has been tested and is complete, add your changes to the staging area. This means that your files are tracked by Git, changes are complete, and code is prepped, and ready to be added back to the repository
- 5 **git status**  
Confirm that your files have been added to the staging area. If the name of the file is listed in green - you're good to go! If not, try adding again
- 6 **git commit -m "valuable\_but\_short\_message\_here"**  
Commit your changes back to the repository. Add a brief, but detailed comment about the changes that were made (e.g., code that was written, or bug that was fixed)
- 7 **git push branch\_name**  
You have now transferred your code up to GitHub. The code you've pushed up lives in the current branch you've worked on. If you accidentally push up to the master you run the risk of overwriting code in the GitHub master branch
- 8 **merge\_request**  
When you are ready to merge your code with the master branch, you notify the owner of the repository through a merge request.  
The owner should review your code prior to finalizing the merge

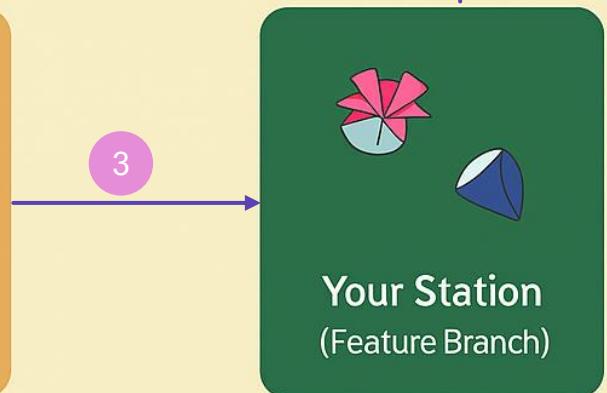
# Another way of thinking about Git workflow is by imagining to replicate a kitchen cooking station

- 1 git clone
- 2 git pull origin master
- 3 git checkout -b new\_branch\_name
- 4 git add file\_name
- 5 git commit
- 6 git push branch\_name
- 7 merge request

## RAMSAYS KITCHEN (Remote Repo)



## TEST KITCHEN (Local Repo)

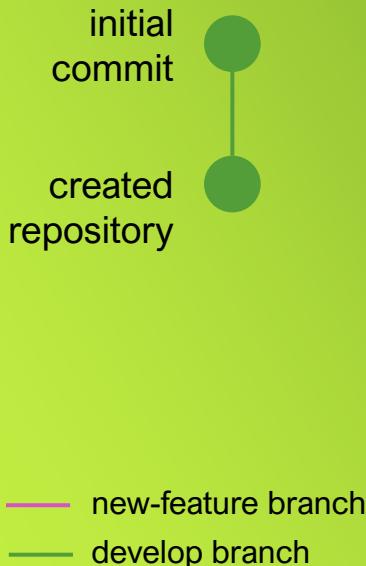


## 4. Useful use-cases

# Create & push new branch from local work

## Scenario

You made some changes in local, and now you want to upload them into a new branch



1. Save changes in local

Check whether your changes are in the staging area

```
git status
```

If not, add them

```
git add .
```

Then, commit

```
git commit -m "implemented  
new feature"
```

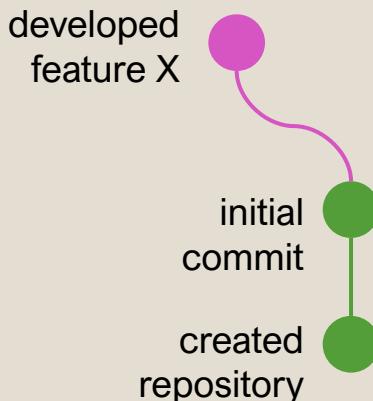
2. Create a new branch inside /feature

From the branch with your changes:

```
git checkout -b feature/new-  
feature
```

3. Push new branch to remote, inside origin

```
git push -u origin  
feature/new-feature
```



✓ At the end:

✓ your code is on the new branch, ready for review or pull request

# Two people working on the same branch

## Scenario

You and your colleague worked on the same branch. You committed A1\_bob locally without pushing to remote. Your colleague, instead, has committed & pushed their changes to remote, A1\_alice.

Now, if you push your A1\_bob, Git checks whether it can update remote branch with a fast-forward (i.e. only adding commits over the last one). But it can't, as remote is one step ahead. So, the push is rejected.

You have first to integrate the missing commit A1\_alice in your local branch

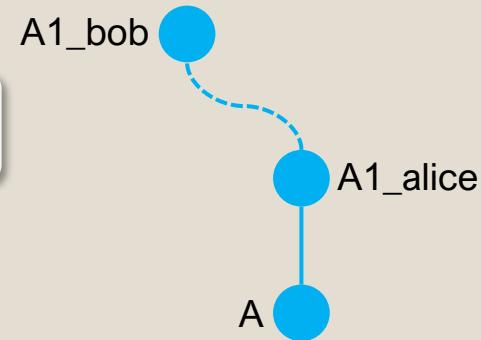


1. **Update the branch** from remote, to include all changes

```
git fetch origin  
git checkout develop/my-feat
```

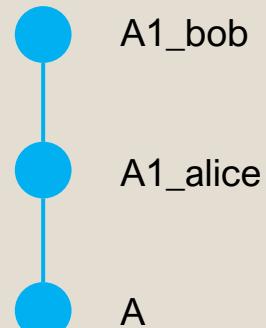
2. **Rebase**

```
git rebase origin/develop/my-feat
```



3. **Push your commit**

```
git push
```

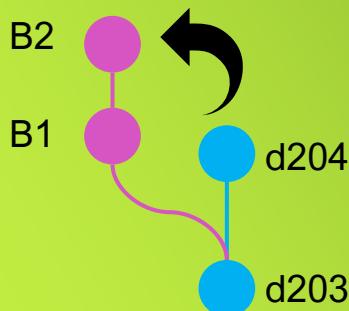


# Integrate advancements from main into your branch

## Scenario

You created the *feature-B* branch starting from commit *d203* of the *develop* branch. Meanwhile you were working on your feature, more commits have been pushed into *develop*, i.e., *d204*.

In order to work on the latest version of the code, you want to pick up the new stuff they pushed into *develop* and integrate it into your branch.



—●— develop branch    —●— feature-B branch

## 1. Update the develop branch from remote, to include all changes

```
git fetch origin  
git checkout develop  
git pull
```

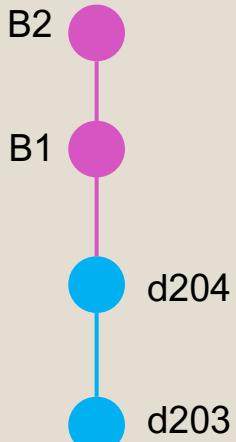
## 2A. Rebase

```
git checkout feature/my-feat  
git rebase origin/develop
```

This moves your commits over the last *develop* and cleans up history. If there are **conflicts**, git will tell you. Resolve them, and then do:

```
git add .  
git rebase --continue
```

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



view after step 3: linear history

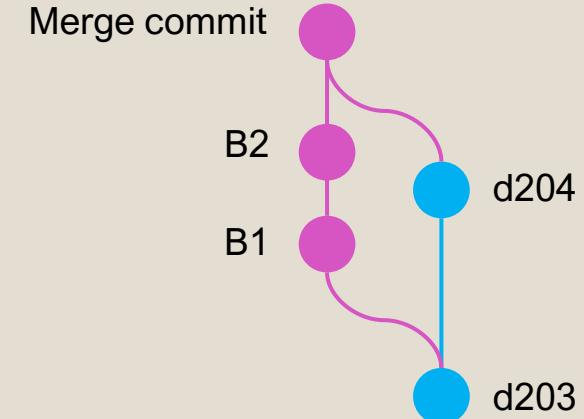
## 2B. Merge

```
git checkout feature/my-feat  
git merge origin/develop
```

This keeps your commits as they are, and **adds a merge commit** on top of the *develop* branch. If there are **conflicts**, git will tell you. Resolve them, and then do:

```
git add .  
git merge --continue
```

```
git push -f -u origin  
feature/my-feat
```



view after step 3: spaghetti history

## 3. Push

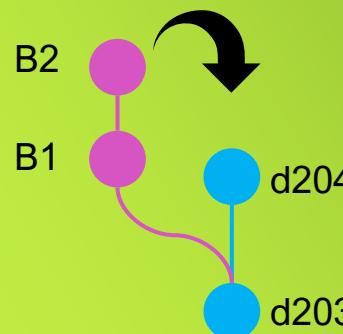
# Integrate your developments into main

## Scenario

You created the *feature-B* branch starting from commit *d203* of the *develop* branch.

Now you are finished with your developments and are ready to integrate them in the *develop* branch to make them available in the main codebase.

Additionally, meanwhile you were working on your feature, more commits have been pushed into *develop*, i.e., *d204*.



—●— develop branch    —●— feature-B branch

## 1. Update the *develop* branch from remote, to include all changes

```
git fetch origin  
git checkout develop  
git pull origin develop
```

## 2A. Rebase

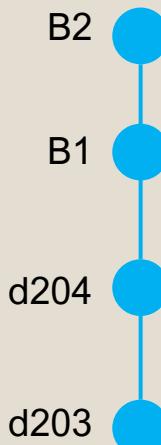
```
git checkout feature-B  
git rebase origin/develop
```

This moves your commits over the last *develop* and cleans up history. If there are **conflicts**, git will tell you. Resolve them, and then do:

```
git add .  
git rebase --continue
```

## 3. Merge

```
git checkout develop  
git merge feature-B
```



view after step 3: linear history

## 2B. Merge

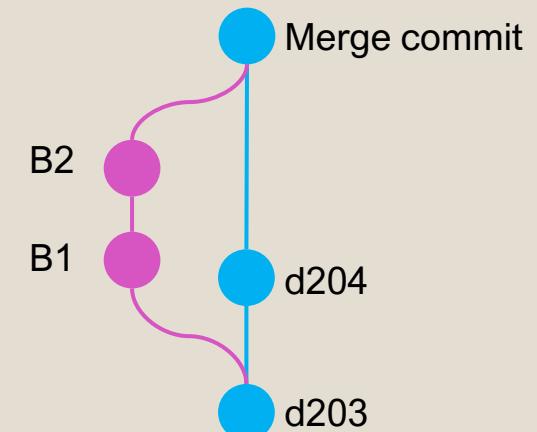
```
git merge feature-B
```

This keeps your commits as they are, and **adds a merge commit** on top of the *develop* branch. If there are **conflicts**, git will tell you. Resolve them, and then do:

```
git add .  
git merge --continue
```

## 3. Push new branch

```
git push origin develop
```



view after step 3: spaghetti history

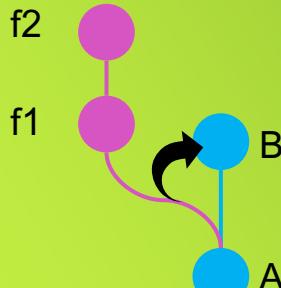
# Periodic rebase

## Scenario

Intermediate commits in a feature branch are worthless to everyone not working on that feature. Nobody cares about your "WIP", "fix typo", or "please work" commits.

Your job: **rewrite your feature branch into exactly one commit**, then **fast-forward** main to it.

- **Fast-forward:** Pretend the branch always lived on main
- **Squash:** Pretend the branch was always one commit

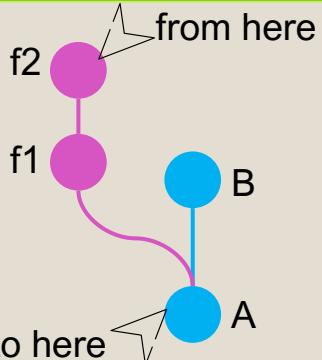


● origin/develop    ● origin/feature-X

- 1 Move branch pointer back to fork point and keep all changes staged

```
git reset -soft $(git merge-base  
-fork-point origin/develop)
```

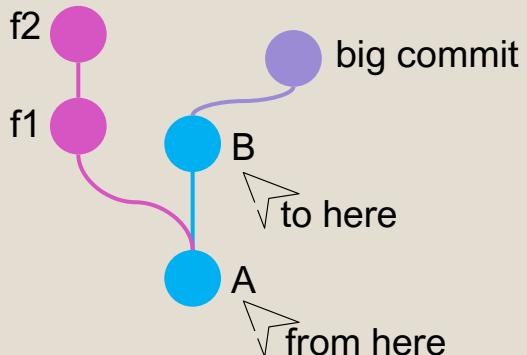
The part in parentheses finds the commit id where your branch originally diverged from origin/develop, i.e., where your branch started.



- 3 Rebase onto latest main

```
git fetch origin  
git rebase origin/develop
```

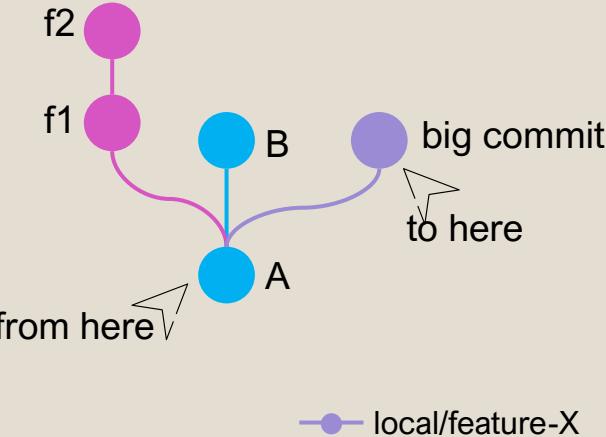
Now git only needs to replay one commit, so if there is a conflict, you resolve it once



- 2 Create 1 new commit containing everything

```
git commit -am 'big commit'
```

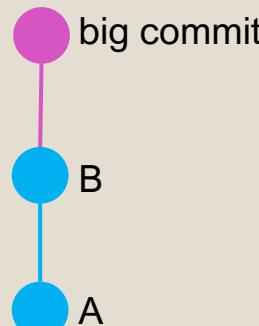
-a automatically stages all tracked and modified files before the commit (equivalent to git add .)



● local/feature-X

- 4 Push (rewrite history)

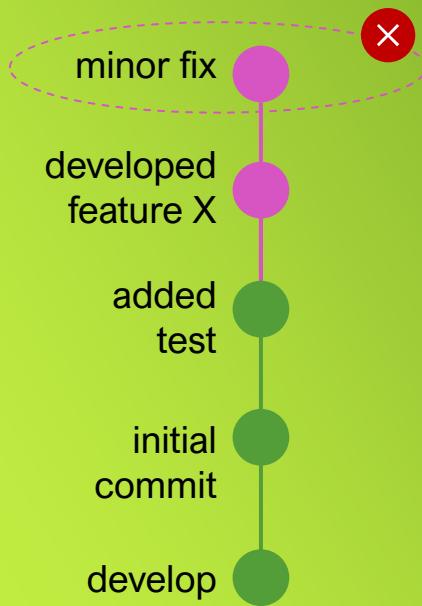
```
git push -force-with-lease
```



# Undo last commit

## Scenario

You did a commit but realized immediately after that you messed up. Don't you worry, you can «undo» that!



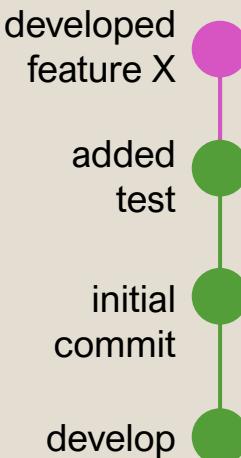
— new-feature branch  
— develop branch

1. Go back to the previous commit

Delete last commit and take the branch back at the previous commit

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

Now you removed the commit from **local**. But you must also remove it from **remote**, so follow step 2.



2. Update remote branch with a forced push

```
git push --force
```

## WARNING!

- hard **deletes**:
- all local uncommitted changes
  - last committed changes

# Clean up local branch

## Scenario

You made some changes in local but in a wrong branch. You don't have yet committed, so you switched to another branch. Now, you would like to clean up **locally** (as you haven't committed) the branch where you initially (and wrongly) made your changes

Go on the wrongly changed branch and remove all local changes you did there (because now everything is on the new branch):

```
git checkout develop  
git reset --hard origin/develop
```

### WARNING!

This erases all local changes that have not been committed, so use safely

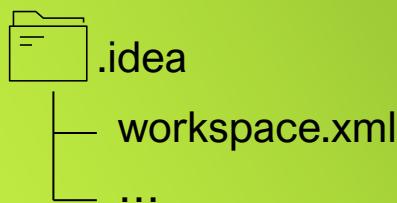
#### At the end:

- ✓ the branch where you wrongly worked at the beginning is now clean

# Remove .idea/ folder from Git tracking

## Scenario

You pushed into a commit your local folder «.idea/». But this is a folder containing local configurations of your IDE, and thus shouldn't be pushed. Moreover, if everyone in a team pushes it, conflicts will arise. Thus, you need to remove it from the Git tracking



1. Remove from Git tracking (not from your disk) all files contained in the .idea folder

```
git rm -cached -r -idea/
```

Now, these files are no longer considered part of the repository. They will still continue to exist on your local folder, so your IDE will continue working correctly

2. Create or update the .gitignore file adding the folder

```
echo ".idea/" >> .gitignore
```

3. Commit changes (removal + gitignore)

```
git add .gitignore  
git commit -m "Removed .idea from repository  
and added to .gitignore"
```

4. Push to remote

```
git push
```

# Stash

## Scenario

You have some uncommitted changes on a branch and want to perform some action that would lead to losing them (e.g., move to another branch). But for now you don't want to commit them, so you put your changes into a temporary folder

### 1. Put aside temporary changes

```
git stash push -m "temp changes"
```

Now your changes are stored in the stash area, and can be retrieved from any branch in the repository

**Note:** changes you want to push into the stash must be changes to tracked files

### 2. Retrieve & apply stashed changes

```
git stash apply
```

This takes your changes and copies them into the current branch; it also keeps a copy of them in the stash area

```
git stash pop
```

This takes your changes and copies them into the current branch; it removes them from the stash area

There could be conflicts when applying stashed changes, and you can resolve them similarly to merge

### 3. If you don't need stashed changes anymore, drop them

```
git stash drop
```