

DataCamp_Intermediate-Git

Chapter 1: Working with branches

Video 1.1: Introduction to branches

What we will cover:

- Branches
- Remotes
- Conflicts

Branches

Branch ::: an individual **version** of a repo

Git uses **branches** to systematically track multiple versions of files

In each branch :: some files might be the same, others might be different, or some may not exist at all

Why use branches?

Live System	Feature development
default branch = main	might encounter issues during development and testing
	doesn't affect the live system

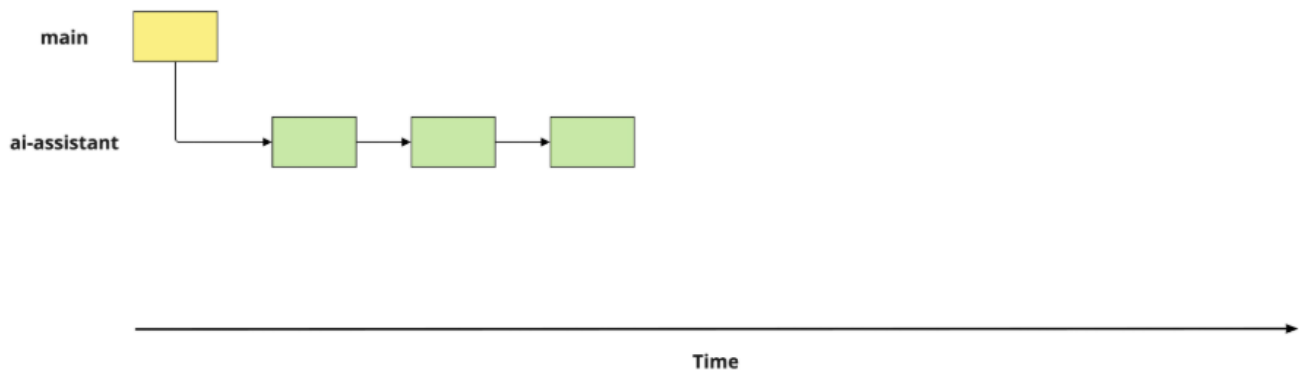
- Branches allow multiple developers **to work on a project simultaneously.**
- Git makes it easy to **compare the state** of a repo **between branches**

- Combine contents, pushing new features to a live system
- Each branch should have a specific purpose

Branching off

We create a new branch from main called ai-assistant. Each box in diagram represents a commit.

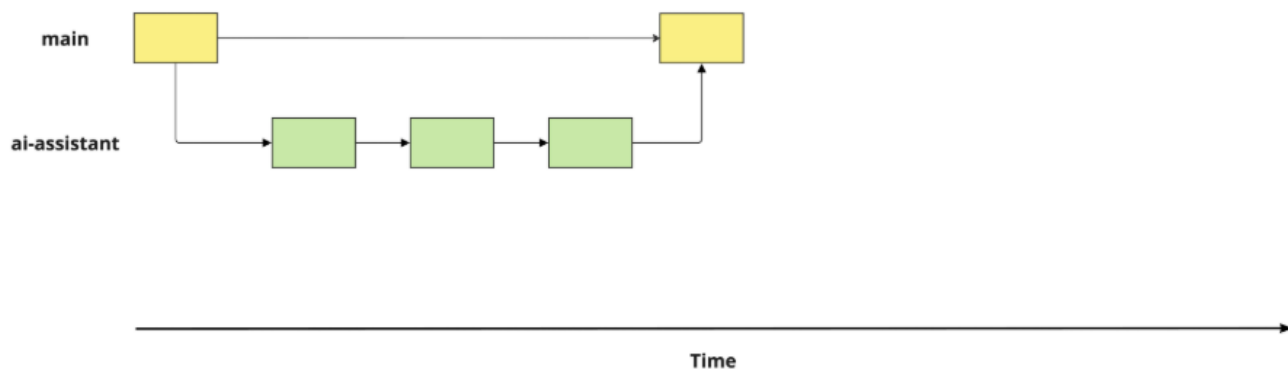
Branching off



Merging back into main

Once the new feature has been developed and tested, we merge our ai-assistant branch back into main, which combines the contents of the branches, making the new feature available to users. We'll discuss merging at the end of the chapter.

Merging back into main



Identifying branches

list all branches ::: `git branch`

```
main
* ai-assistant
```

* ::: indicates current branch

switching into `main` branch ::: `git switch main`

create a new branch called `speed-test` ::: `git branch speed-test`

move to the `speed-test` branch ::: `git switch speed-test`

create a new branch called `speed-test` and switch to it ::: `git switch -c speed-test`

Terminology

creating a new branch = "branching off"

creating `speed-test` from `main` = "branching off `main`"

Video 1.2: Modifying and comparing branches

Comparing branches

to compare `main` to `summary-statistics` ::: `git diff main summary-statistics`

This is the first half, with the green text in the output representing new content added in the second branch, `summary-statistics`.

```
diff --git a/bin/summary b/bin/summary
new file mode 100755
index 0000000..9d6e2fa
--- /dev/null
+++ b/bin/summary
@@ -0,0 +1,44 @@
+Summary statistics
+
+Age:
+Yes: 25
+No: 24
+
+treatment:
+Yes: 31
+No: 18
+
+work_interfere:
+Sometimes: 17
```

In the second half of the output we see that git is showing the diff for the summary.txt file within the results directory between the two branches. Hence, the only difference between the branches was the creation of this summary file and the addition of content to it. If there are multiple files that have been modified, or more extensive revisions to existing files, then the diff output will grow!

```

+
+benefits:
+Don't know: 17
+Yes: 17
+No: 15
+
+mental_health_interview:
+No: 41
+Maybe: 7
+No: 1
+
+mental_vs_physical:
+Don't know: 24
+Yes: 15
+No: 10
diff --git a/results/summary.txt b/results/summary.txt
new file mode 100644
index 0000000..e69de29

```

Navigating large git outputs

- can produce large outputs!
 - Press `space` to progress through
 - Press `q` to exit

Modifying branches

```
git branch
```

```
main
* feature_dev
```

- need another branch for a second new feature being developed
- Solution - rename `feature_dev` Renaming a branch from `feature_dev` to `chatbot` :: `git branch -m feature_dev chatbot``

Checking our branches

- confirm the change by running `git branch`

```
git branch
```

```
main
* chatbot
```

Deleting a branch

- Large projects can have many branches
- Delete branches once we are finished with them
- Delete the `chatbot` branch with the `-d` flag

delete the `chatbot` branch
??

```
git branch -d chatbot
```

Deleting a branch that hasn't been merged

- If `chatbot` hasn't been merged to `main`, `git branch -d chatbot` will produce an error.

```
error: The branch 'chatbot' is not fully merged.
If you are sure you want to delete it, run 'git branch -D
chatbot'.
```

- Delete with the `-D` flag if you are sure you want to delete.

```
git branch -D chatbot
```

Deleted branch chatbot (was 3edb989).

- Difficult, but not impossible, to recover deleted branches.
- Be sure we don't need the branch any more before deleting!

Summary:

`git diff main chatbot` ::: compare the state of the `main` and `chatbot` branches

`git branch` ::: list all branches

`git branch -m old_name new_name` ::: rename branch called `old_name` to `new_name`

`git branch -d chatbot` ::: delete `chatbot` branch, which **has** been merged.

`git branch -D chatbot` ::: delete `chatbot` branch, which ***has not*** been merged.

Video 1.3: Merging branches

The purpose of branches

- Each branch should **have a particular purpose**.
 - Developing a new feature
 - Debugging an error
- Once the task is complete, we incorporate the changes into production
 - Typically the `main` branch - "ground truth"
- When merging two branches:
 - the last commits from each branch are called **parent commits**
 - `source` = the branch we want to merge **from**.

- `destination` = the branch we want to merge **into**.
- When merging `ai-assistant` into `main`:
 - `ai-assistant` = `source`
 - `main` = `destination`

Merging branches

move to the destination branch:

??

```
git switch main
```

- syntax: `git merge source`

From `main`, to merge `ai-assistant` into `main`:

??

```
git merge ai-assistant
```

From another branch: `git merge source destination`

??

```
git merge ai-assistant main
```

Git merge output

```
Updating 7964fe1..d7b2310
Fast-forward
 source/main.py | 11 ++++++++
 1 file changed, 11 insertions(+)
 create mode 100644 source/main.py
```


Commit hashes

```
Updating 7964fe1..d7b2310
Fast-forward
 source/main.py | 11 ++++++++
 1 file changed, 11 insertions(+)
 create mode 100644 source/main.py
```

- Parent commits

Type of merge

```
Updating 7964fe1..d7b2310
Fast-forward
 source/main.py | 11 ++++++++
 1 file changed, 11 insertions(+)
 create mode 100644 source/main.py
```

- **Linear commit history**: branched off `main` to create `ai-assistant`
- **Fast-forward**: point `main` to the last commit in the `ai-assistant` branch

Number of lines changed

```
Updating 7964fe1..d7b2310
Fast-forward
 source/main.py | 11 ++++++++
 1 file changed, 11 insertions(+)
 create mode 100644 source/main.py
```

Files modified

```
Updating 7964fe1..d7b2310
Fast-forward
 source/main.py | 11 ++++++++
 1 file changed, 11 insertions(+)
 create mode 100644 source/main.py
```

Chapter 2: Collaborating with Git

Video 2.1: Merge Conflicts

conflict ::: the inability to resolve differences in the contents of one or more files between branches

- edit the same file in two branches
- try to merge
- git doesn't know what version to keep
- conflict!

Conflicting versions of README.md

documentation branch	main branch
Contents and usage This repo contains source code for the DataCamp website. It also contains source code for an AI-Assistant (recommendation system) that takes prompts from learners and returns suggested content that they might be interested in. It is for internal use only, external access is prohibited.	Contents and usage This repo contains source code for the DataCamp website. It is for internal use only, external access is prohibited.

Merging

From the `main` branch,

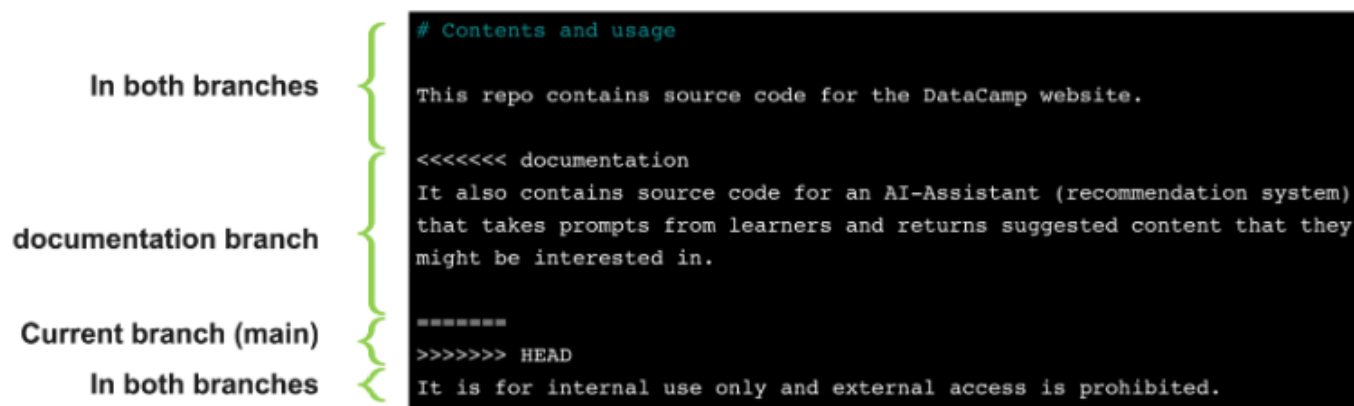
```
git merge documentation
```

```
Auto-merging README.md
CONFLICT (add/add): Merge conflict in README.md
Automatic merge failed; fix conflicts and then commit the
result.
```

Opening the file

to open the text editor on `README.md` :: `nano README.md`

Git conflict syntax



Video 1.2: Introduction to remotes

Local repo ::: repo stored on your local computer

remote repo ::: a repo stored in the cloud through an online repo hosting service like GitHub

2 benefits of remote repos :: (1) everything is backed up, (2) collaboration, regardless of location

local remotes ::: repo copies on our local computer

cloning ::: making a copy of a repo

- Cloning a repo is useful for collaboration where cloud storage is prohibited, such as in enterprises with strict data privacy rules.

to clone a repo

??

```
git clone path-to-project-repo
```

cloning a local project

??

```
git clone /home/george/repo
```

cloning and naming a new project

??

```
git clone /home/george/repo new_repo
```

Cloning a remote

- Remote repos are generally stored in an online hosting service, e.g. GitHub, Bitbucket, or GitLab
- If we have an account, we can clone a remote repo onto our local computer

To clone a url:

??

```
git clone URL
```

```
git clone https://github.com/datacamp/project
```

Identifying a remote

- When cloning a repo, git remembers where the original was

- Git stores a remote **tag** in the new repo's configuration

List all remotes associated with the repo

??

```
git remote
```

Creating a remote

- When cloning, git will automatically name the remote **origin**.

General syntax to clone a remote with a specific name:

??

```
git remote add name URL
```

Create a remote called **george** :

??

```
git remote add george https://github.com/george_datacamp/repo
```

- Defining remote names is useful for **merging**.

Getting more information

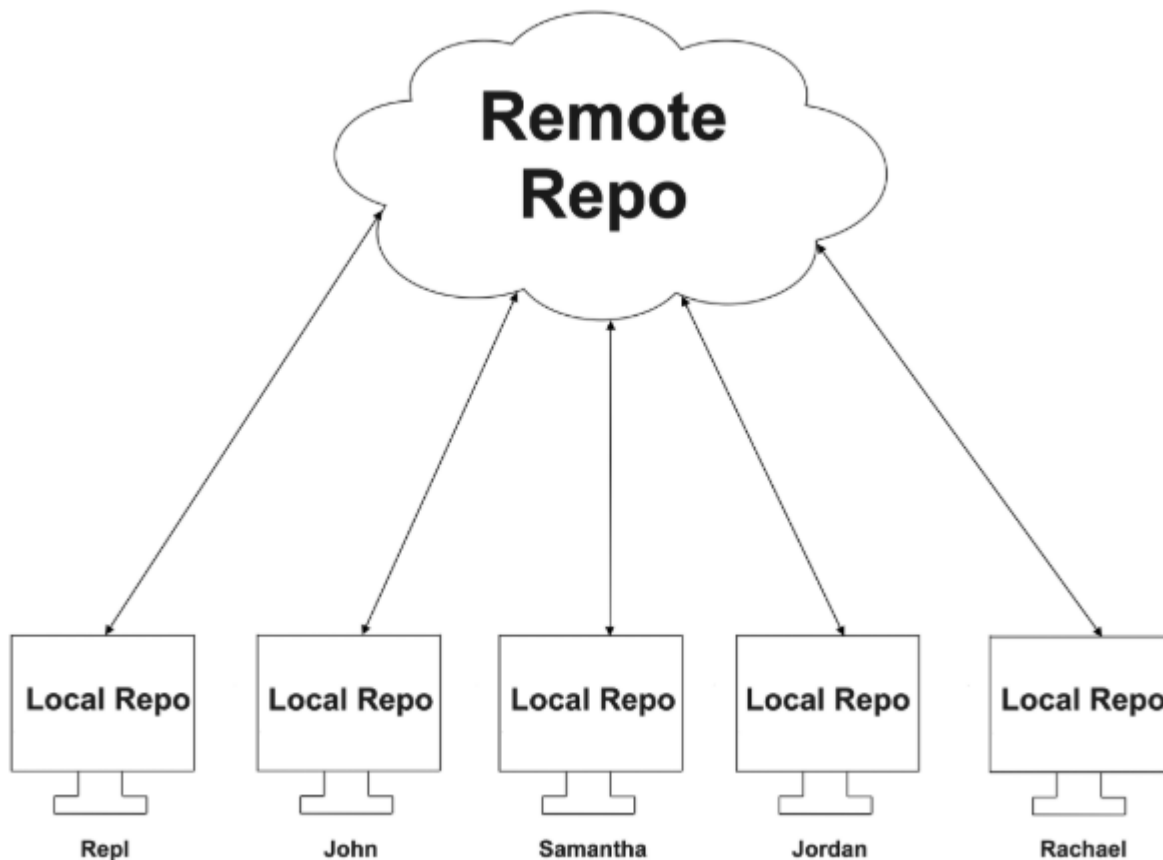
get more information about the remote(s):

??

```
git remote -v
```

- the **-v** stands for **verbose**.

Video 2.3: Pulling from remotes



If several people are collaborating on a project then they will access the remote, work locally, save files, and synchronize their changes between the remote and local repos. This means that the remote repo should be the project's source of truth, where the latest versions of files that are not drafts can be located.

Fetching from a remote

- To synchronize the files in a remote to a local repo, we need to fetch branches from the remote.

Fetch from the `origin` remote:
??

```
git fetch origin
```

- Fetch all remote branches

- Might create new local branches if they only existed in the `remote`
- Doesn't merge the remote's contents into local repo

Fetching a remote branch

Fetch only from the `origin` remote's `main` branch:
??

```
git fetch origin main
```

```
From https://github.com/datacamp/project
* branch                main          -> FETCH_HEAD
```

Synchronizing content

After fetching, we need to synchronize content between the remote and local repos. To do this, we perform a merge of the remote into our local repo.

Merge `origin` remote's default branch (`main`) into the local repo's current branch:
??

```
git merge origin
```

The output is the same as if we merged two local branches:

```
Updating 9dcf4e5..887da2d
Fast-forward
 tests/tests.py | 13 ++++++++
 README.md      | 10 +++++
 2 files changed, 23 insertions (+)
```

Pulling from a remote

- Local and remote synchronization is a common workflow
- Git simplifies this process for us!
Fetch and merge from the remote's default (`main`) into the local repo's current branch:
??

```
git pull origin
```

Pulling a remote branch

Pull from the `origin` remote's `dev` branch:
??

```
git pull origin dev
```

- still merges into the local branch we are located in

Git pull output

```
From https://github.com/datacamp/project
* branch                dev          -> FETCH_HEAD
Updating 9dcf4e5..887da2d
Fast-forward
 tests/tests.py | 13 ++++++++
 README.md      | 10 ++++++++ 2 files changed, 23 insertions (+)
```

- output is a combination of `git pull` and `git merge`

A word of caution


```
git pull origin
```

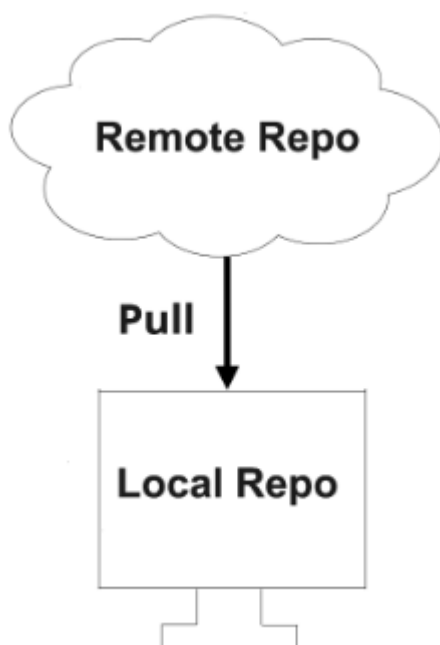
Note that if we have been working locally and not yet committed our changes, then Git won't allow us to pull from a remote. We are instructed to commit our changes and told that the pull command was aborted. Therefore, it's important to save our work locally before we pull from a remote.

```
Updating 9dcf4e5..887da2d
error: Your local changes to the following files would be
overwritten by merge:
      README.md
Please commit your changes or stash them before you merge.
Aborting
```

- important to save locally before pulling from a `remote`

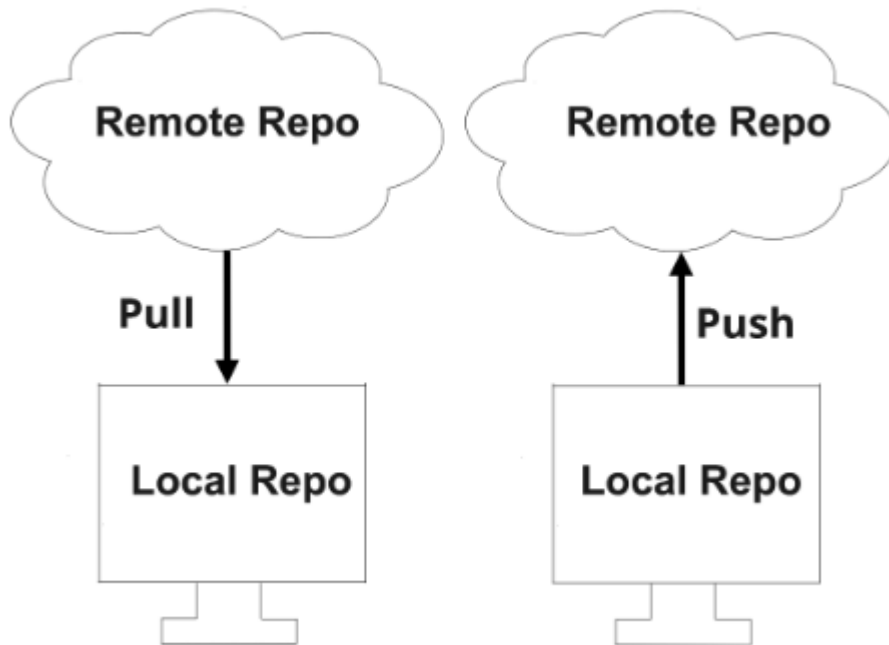
Video 2.4: Pushing to a remote

Pulling from a remote



Pushing to a remote

- the opposite of pulling is pushing, where we merge our local's repo content into a remote



git push

- We need to commit local changes first before we can push to the remote!

general push syntax:

??

```
git push remote local-branch
```

- Push *into* remote **from** local_branch
- 2 arguments for `git push` :: 1) the remote and 2) the local branch

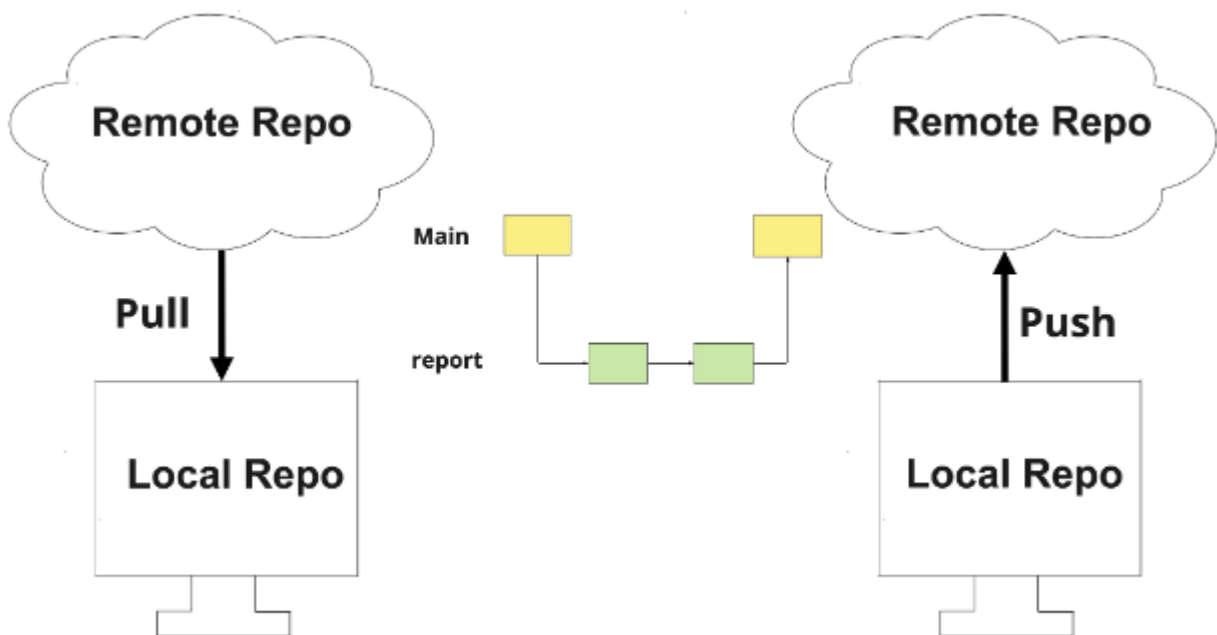
Push changes into `origin` from the local repo's `main` branch:

??

```
git push origin main
```

Push/pull workflow

- The typical push/pull workflow starts with pulling the remote into our local repo.
- We then work on our project locally, committing changes as we go.
- Lastly, we push our updated local repo to the remote.
- This workflow is repeated throughout our time working on the project.




Pushing first

What happens if we don't start the workflow by pulling from the remote?


Remote/local conflicts

Remote



```
To https://github.com/datacamp/project
! [rejected]        main -> main (non-fast-forward)
error: failed to push some refs to 'https://github.com/datacamp/project'
hint: Updates were rejected because the tip of your branch is behind
hint: its remote counterpart. Integrate the remote changes (e.g.
hint: 'git pull ..') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

Outcome



```
To https://github.com/datacamp/project
! [rejected]        main -> main (non-fast-forward)
error: failed to push some refs to 'https://github.com/datacamp/project'
hint: Updates were rejected because the tip of your branch is behind
hint: its remote counterpart. Integrate the remote changes (e.g.
hint: 'git pull ..') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

Reason(s) and suggestion(s)

```
To https://github.com/datacamp/project
! [rejected]        main -> main (non-fast-forward)
error: failed to push some refs to 'https://github.com/datacamp/project'
hint: Updates were rejected because the tip of your branch is behind
hint: its remote counterpart. Integrate the remote changes (e.g.
hint: 'git pull ..') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

Avoiding a conflict

- Pull from the remote first:

```
git pull origin main
```

```
branch 'master' of www.github.com/datacamp/project

# Please enter a commit message to explain why this merge is necessary,
# especially if it merges an updated upstream into a topic branch.
#
# Lines starting with '#' will be ignored, and an empty message aborts
# the commit.
```

Previously we saw that git pull used a fast-forward merge. In this case it's slightly different, as there are different commits on the local and remote repos, so Git can't just bring them in line with one another. It will use a **recursive merge**. So, when we try to pull, Git will automatically open the

nano text editor and ask us to add a message for the merge. We leave a message that we are pulling the latest report from the remote, then save and exit nano.

Pulling without editing

```
git pull --no-edit origin main
```

- Not recommended, unless we are very confident in the history of our project!

Pushing a new local branch

- Working in `hotfix` branch locally
- `hotfix` does not exist in the remote

Creating a new remote branch

`hotfix` only exists locally - create a new remote branch:
??

```
git push origin hotfix
```

```
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 8 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 349 bytes | 349.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
remote:
remote: Create a pull request for 'hotfix' on GitHub by visiting:
remote:   https://github.com/datacamp/project/pull/new/hotfix
remote:
To https://github.com/datacamp/project
* [new branch]      hotfix -> hotfix
```