

---

# Git

— Boston University CS 506 - Lance Galletti —

---

# GitHub vs Git

**GitHub** --> [[browser](#)]

GitHub is a **website** that allows you to upload your git repositories online. It allows you to have a **backup of your files online**, has a visual interface to navigate your repos, and it allows other people to be able to view, copy, or contribute to your repos.

**Git** --> [terminal]

Git is a **version control system**. It allows you to manage the history of your git repositories.

# Motivation

For each codebase (repository) I own, I want to write code where:

1. Progress loss is minimized
2. Iterating on different versions of the code is easy
3. Collaboration is productive

# Minimal Progress Loss

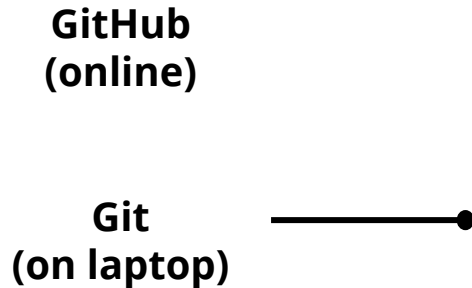
This is achieved by effectively “backing up your work”.

By creating regular save points (called **commits**) and **pushing** them up to GitHub (from your laptop), if your laptop is destroyed you will only lose the commits you did not upload to GitHub.

# Minimal Progress Loss

This is achieved by effectively “backing up your work”.

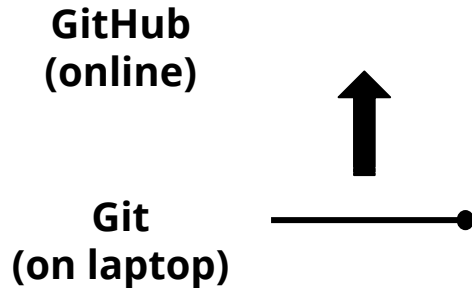
By creating regular save points (called **commits**) and pushing them up to GitHub (from your laptop), if your laptop is destroyed you will only lose the commits you did not upload to GitHub.



# Minimal Progress Loss

This is achieved by effectively “backing up your work”.

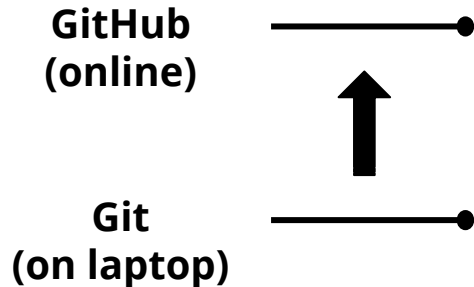
By creating regular save points (called **commits**) and pushing them up to GitHub (from your laptop), if your laptop is destroyed you will only lose the commits you did not upload to GitHub.



# Minimal Progress Loss

This is achieved by effectively “backing up your work”.

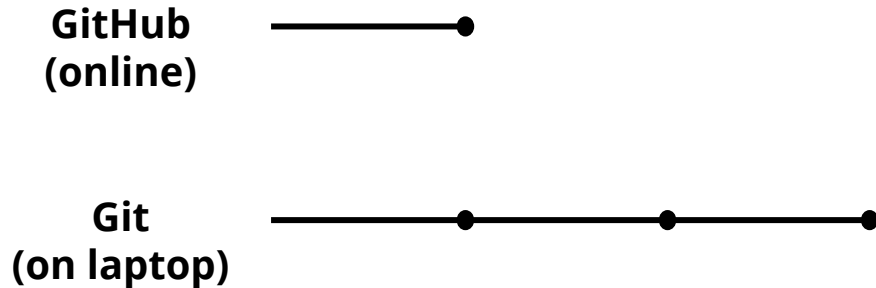
By creating regular save points (called **commits**) and pushing them up to GitHub (from your laptop), if your laptop is destroyed you will only lose the commits you did not upload to GitHub.



# Minimal Progress Loss

This is achieved by effectively “backing up your work”.

By creating regular save points (called **commits**) and pushing them up to GitHub (from your laptop), if your laptop is destroyed you will only lose the commits you did not upload to GitHub.

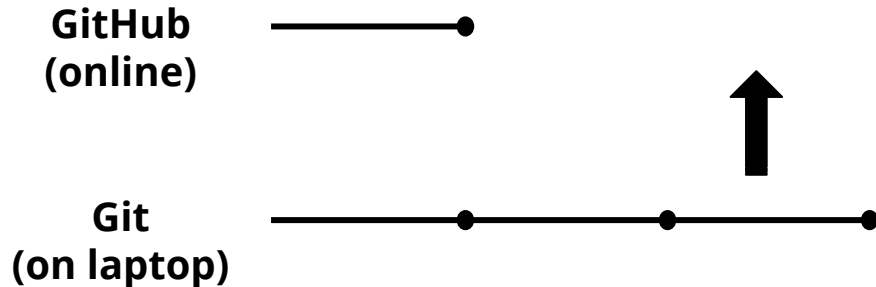




# Minimal Progress Loss

This is achieved by effectively “backing up your work”.

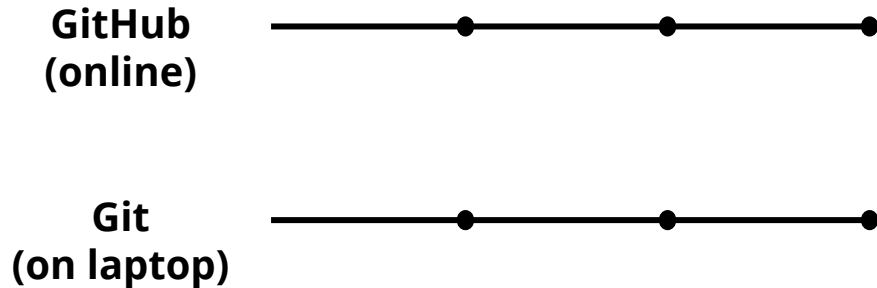
By creating regular save points (called **commits**) and pushing them up to GitHub (from your laptop), if your laptop is destroyed you will only lose the commits you did not upload to GitHub.



# Minimal Progress Loss

This is achieved by effectively “backing up your work”.

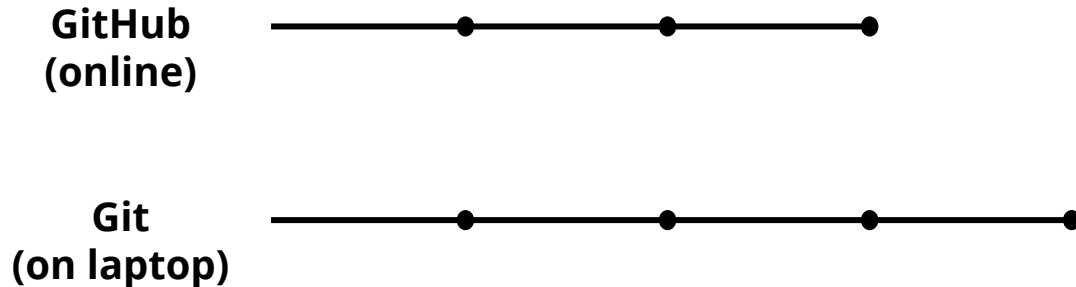
By creating regular save points (called **commits**) and pushing them up to GitHub (from your laptop), if your laptop is destroyed you will only lose the commits you did not upload to GitHub.



# Minimal Progress Loss

This is achieved by effectively “backing up your work”.

By creating regular save points (called **commits**) and pushing them up to GitHub (from your laptop), if your laptop is destroyed you will only lose the commits you did not upload to GitHub.



# Minimal Progress Loss

This is achieved by effectively “backing up your work”.

By creating regular save points (called **commits**) and pushing them up to GitHub (from your laptop), if your laptop is destroyed you will only lose the commits you did not upload to GitHub.

GitHub  
(online)



~~Git  
(on laptop)~~



**Demo**

# Iterating on Different Versions

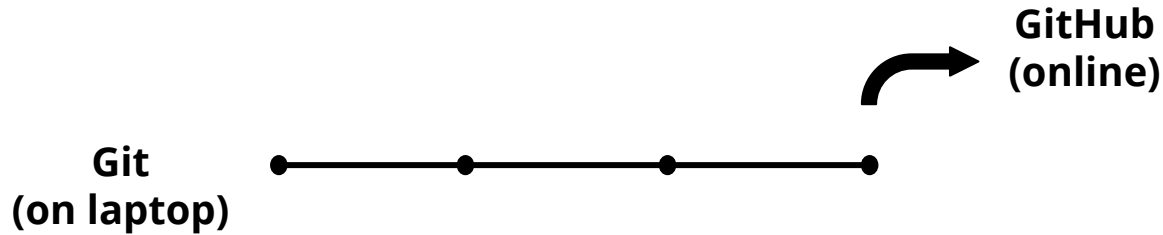
The ease or difficulty of adding a new feature to the code base may depend on the state / version of the codebase.

It may be easiest to add this feature at a specific commit.

# Iterating on Different Versions

The ease or difficulty of adding a new feature to the code base may depend on the state / version of the codebase.

It may be easiest to add this feature at a specific commit.



# Iterating on Different Versions

The ease or difficulty of adding a new feature to the code base may depend on the state / version of the codebase.

It may be easiest to add this feature at a specific commit.

**GitHub**  
(online)



**Git**  
(on laptop)






# Iterating on Different Versions

The ease or difficulty of adding a new feature to the code base may depend on the state / version of the codebase.


It may be easiest to add this feature at a specific commit.

**GitHub**  
(online)



A horizontal line with four black dots, representing a sequence of four commits on GitHub.

**Git**  
(on laptop)



A horizontal line with two black dots, representing a sequence of two commits on a local Git repository.

# Iterating on Different Versions

The ease or difficulty of adding a new feature to the code base may depend on the state / version of the codebase.

It may be easiest to add this feature at a specific commit.

**GitHub  
(online)**

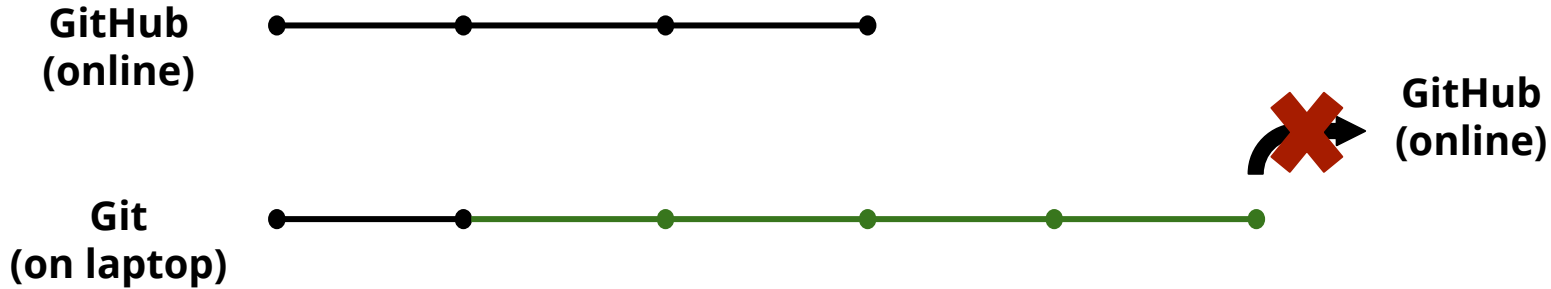


**Git  
(on laptop)**



# Iterating on Different Versions

What happens now?



# Iterating on Different Versions


Looks like we need:

1. A way to preserve both versions of history
2. A way to overwrite history if we choose (this is dangerous as we will lose that history)


# Iterating on Different Versions

Let's try that again!

**GitHub**  
(online)



**Git**  
(on laptop)



# Iterating on Different Versions

We will **branch** off of that particular commit

GitHub  
(online)

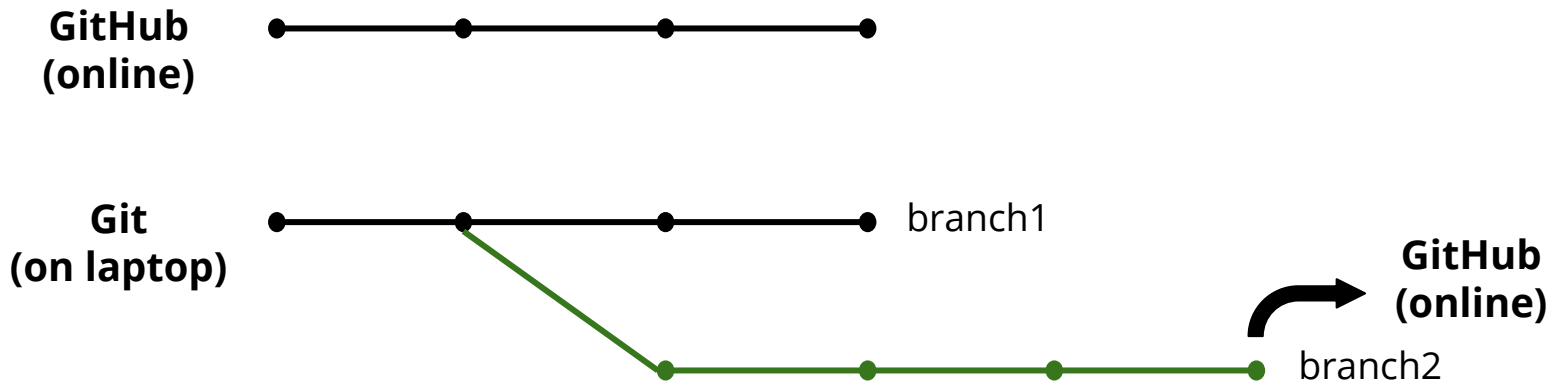


Git  
(on laptop)



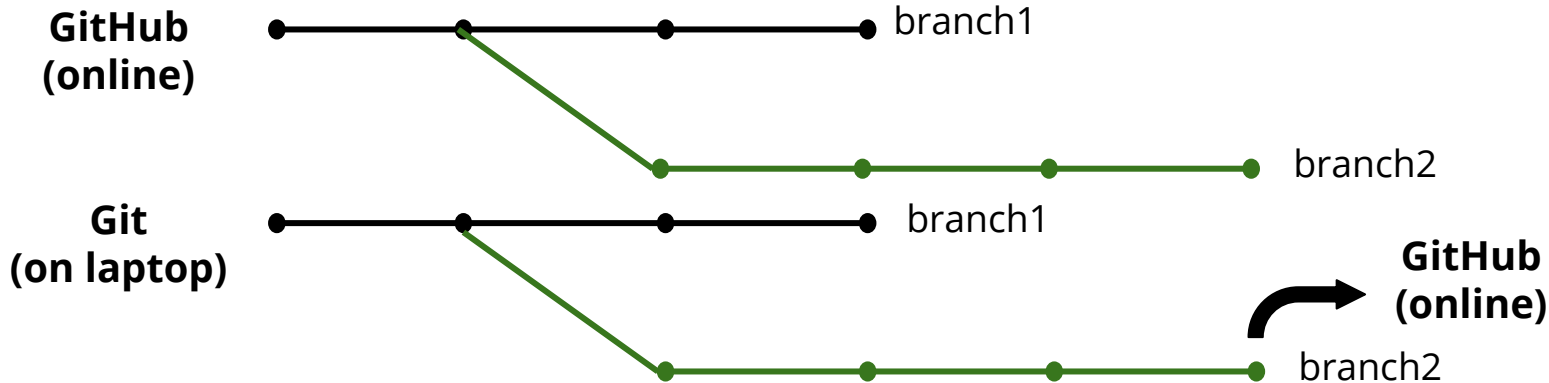
# Iterating on Different Versions

We can push **commits** per **branch**



# Iterating on Different Versions

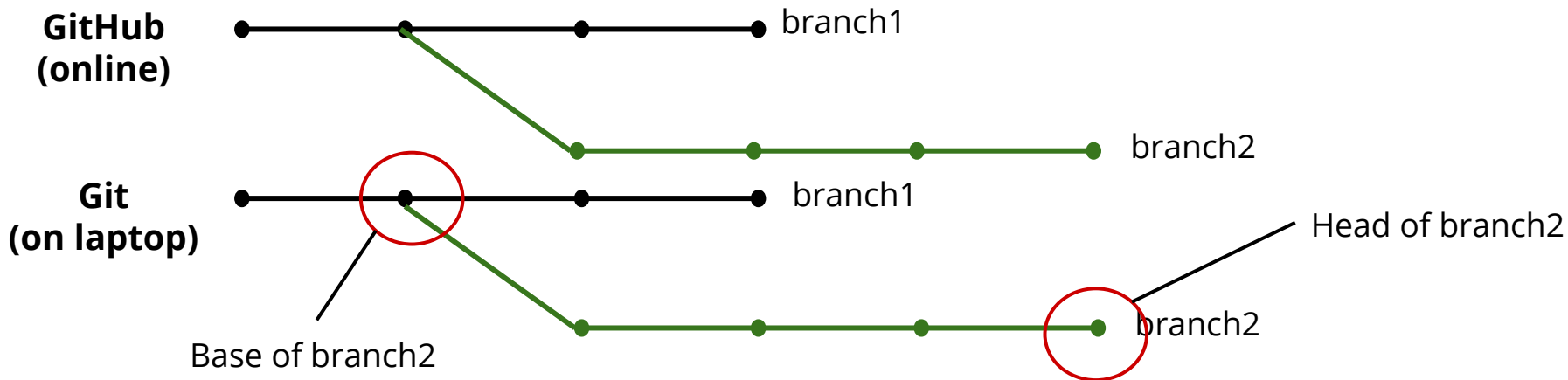
We can push **commits** per **branch**





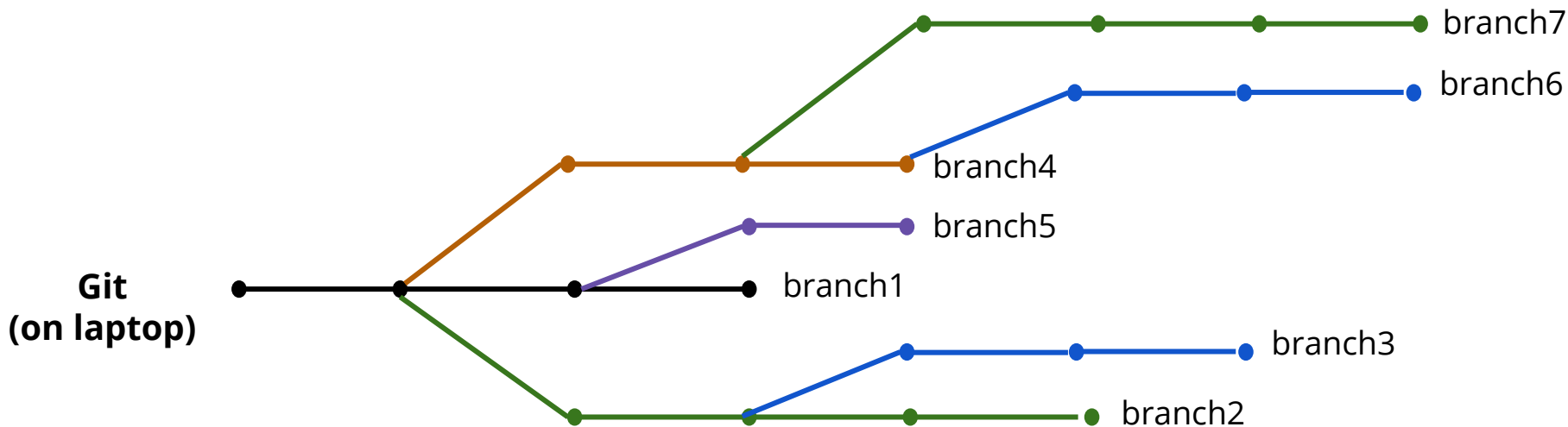
# Iterating on Different Versions

We can push **commits** per **branch**



# Iterating on Different Versions

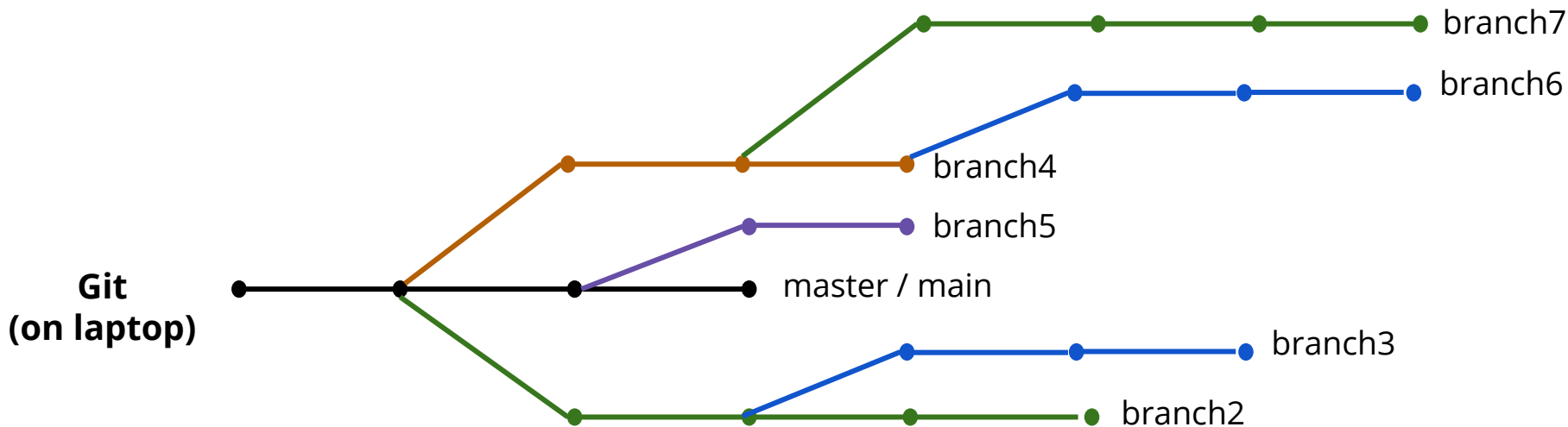
We can create lots of **branches**



# Iterating on Different Versions

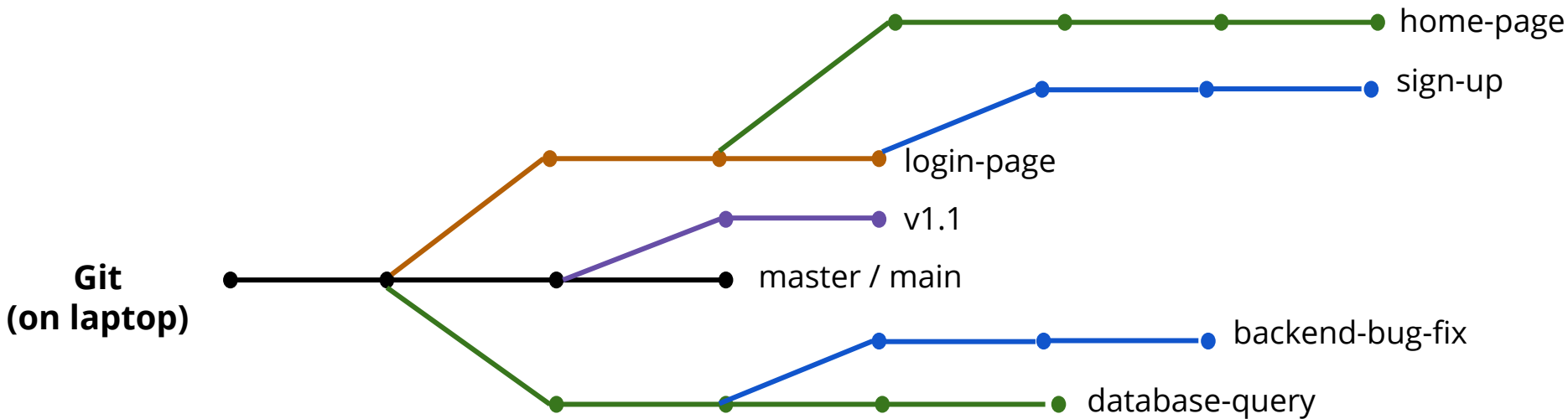
But one branch needs to be chosen as the primary, stable branch

This branch is typically called the “master” or “main” branch



# Iterating on Different Versions

Other branches are usually named after either the feature that is being developed on or the major or minor version of the software / product

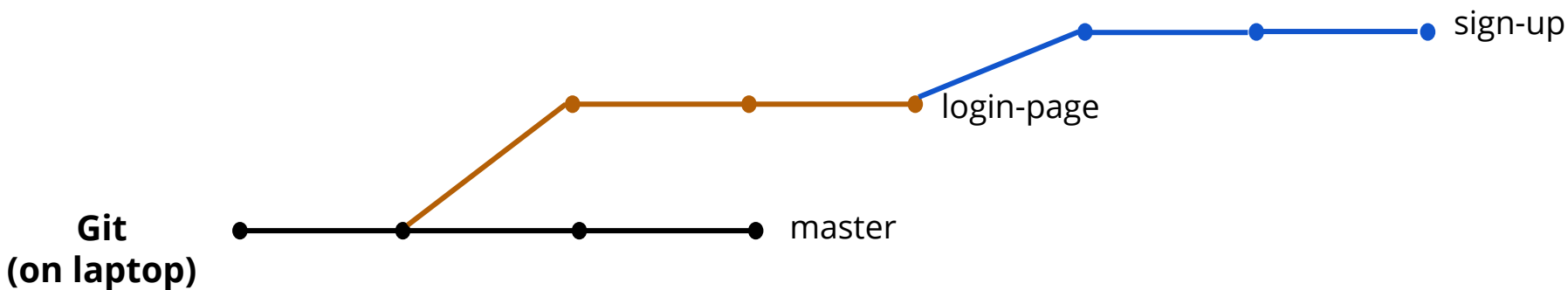


# Iterating on Different Versions

At some point we will want to clean up certain branches by **merging** them with the master / main branch or with each other.

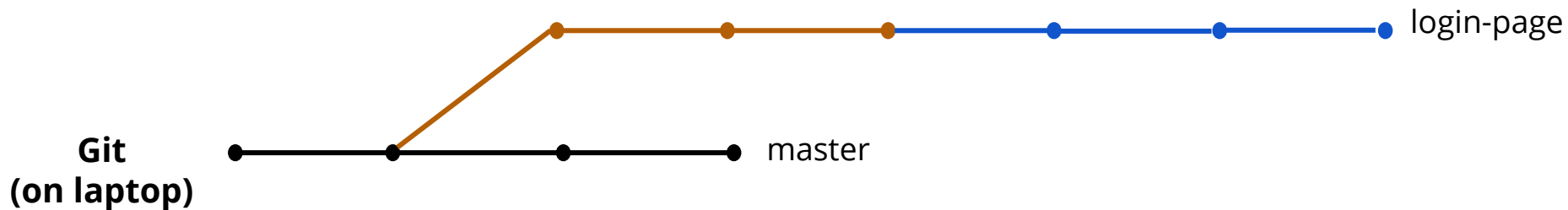
# Iterating on Different Versions

At some point we will want to clean up certain branches by **merging** them with the master / main branch or with each other.



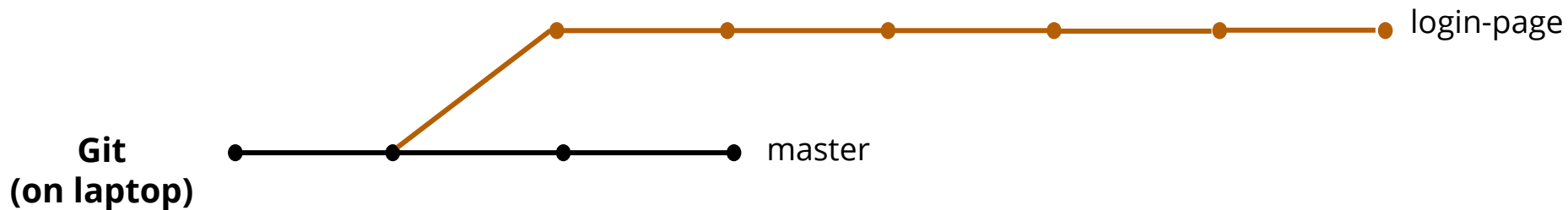
# Iterating on Different Versions

Merging is trivial if the **base** of one branch is the **head** of the other - the changes are “simply” appended.



# Iterating on Different Versions

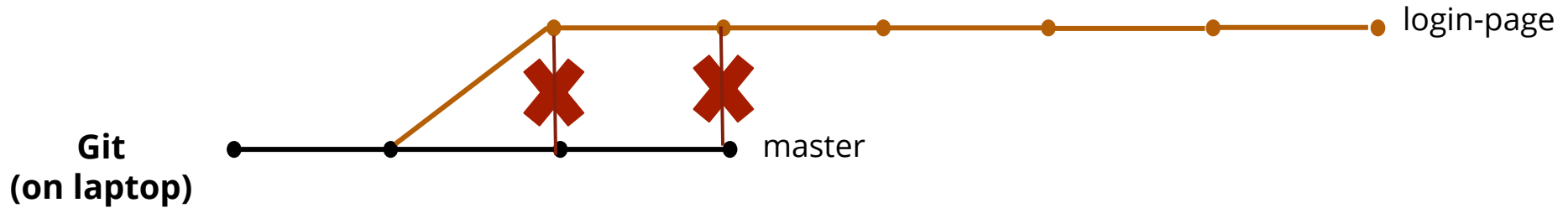
When this is not the case, commits can conflict with each other





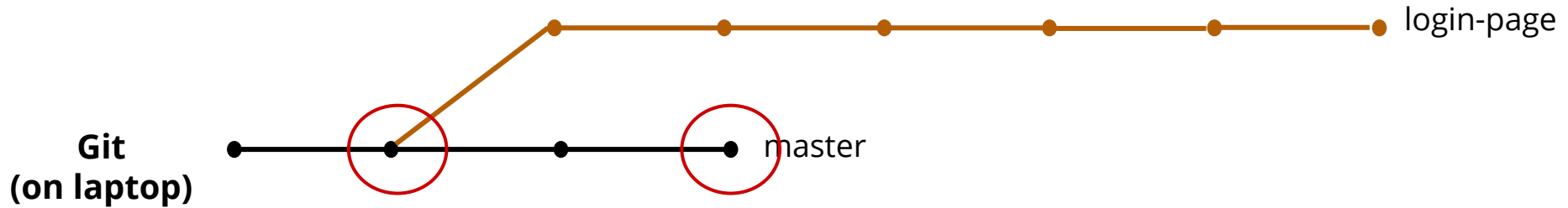
# Iterating on Different Versions

When this is not the case, commits can conflict with each other



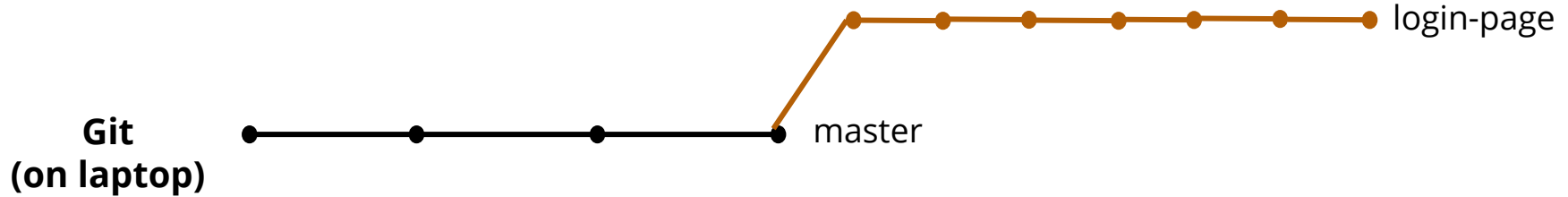
# Iterating on Different Versions

We need to change the **base** of the login-page branch (**rebase**) to be at the **head** of the master branch



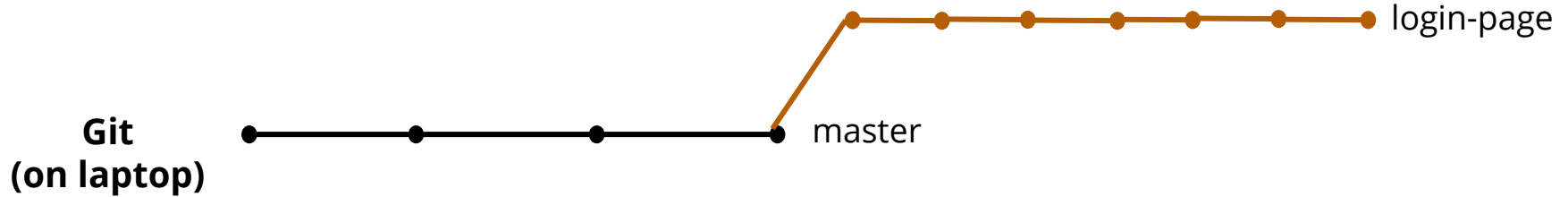
# Iterating on Different Versions

We need to change the base of the login-page branch (**rebase**) to be at the head of the master branch



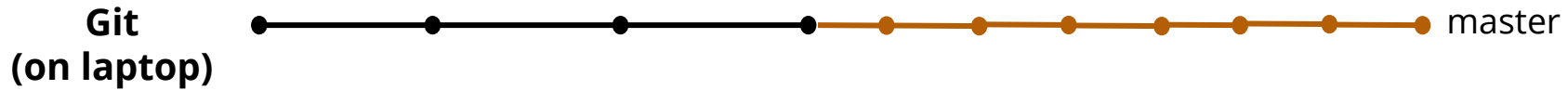
# Iterating on Different Versions

This is not a simple operation! It will often require **manual intervention** to resolve the conflicts.



# Iterating on Different Versions

This is not a simple operation! It will often require **manual intervention** to resolve the conflicts.



**Demo**

# Collaboration

Possible to collaborate on a single repository by having each collaborator create new branches for development and having a process for merging their branch into master.

This request to merge code is done by a **Pull Request** (PR).

# Collaboration

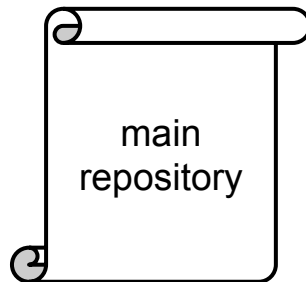
Typically (in particular with open source repositories), repository owners prefer not having to manage collaborator permissions on the repository.

In order to contribute code, collaborators must:

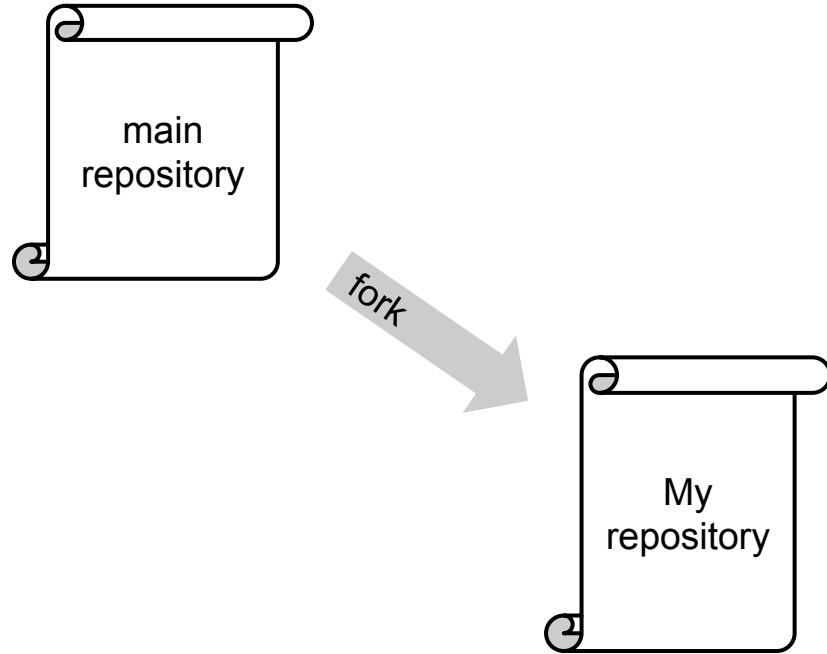
1. Make a copy of (**fork**) the main repository
2. Make all the changes they want to this copy
3. Request that part of their copy be merged into the main repository via a **Pull Request** (PR)



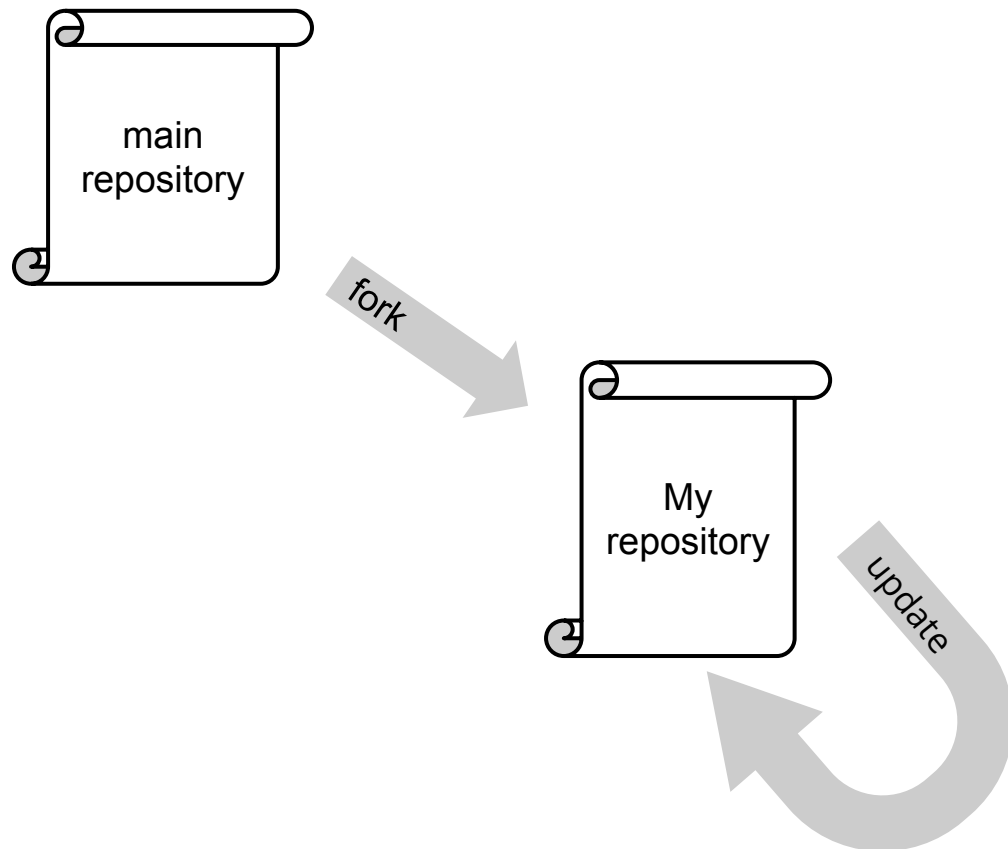
# Collaboration



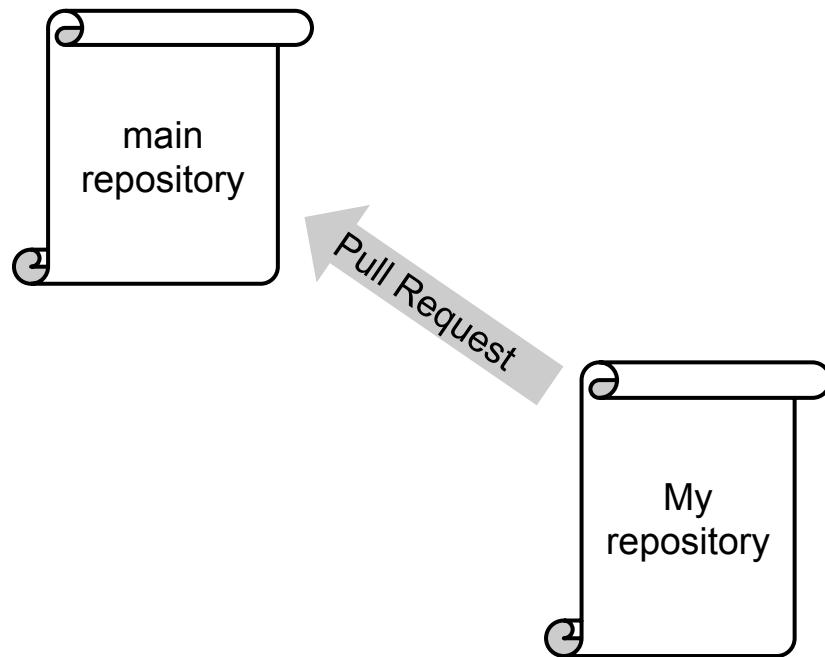
# Collaboration



# Collaboration



# Collaboration



# Collaboration

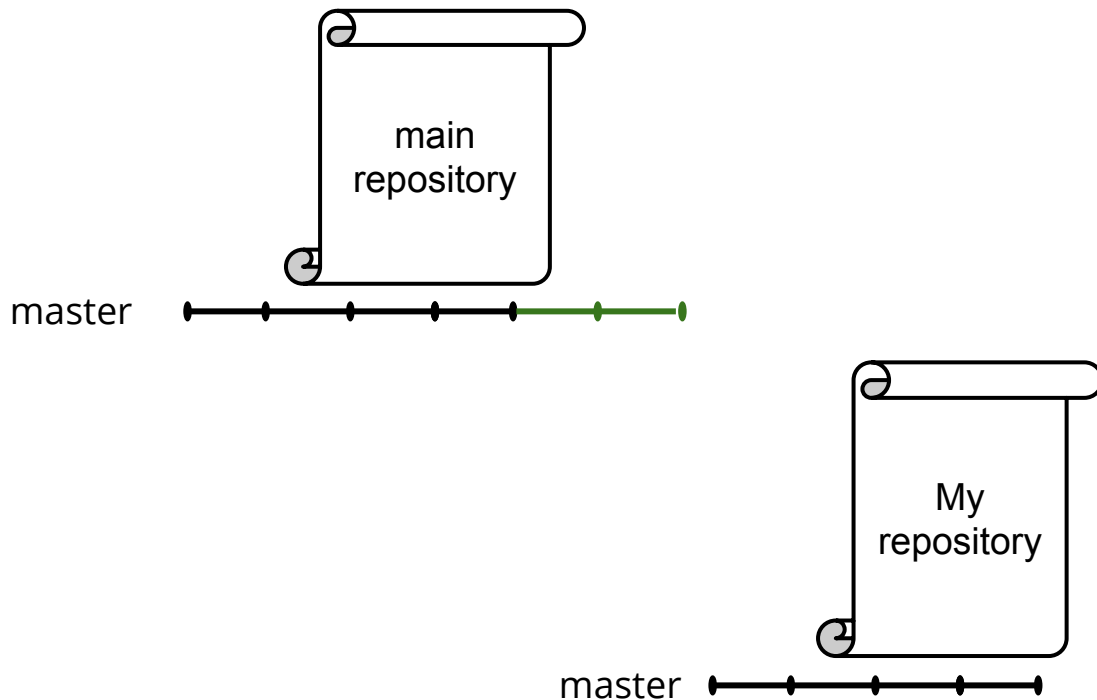
In the time it took you to implement your new feature, the main repository has changed since the time you **forked** the repository.

How do you keep your copy up to date?

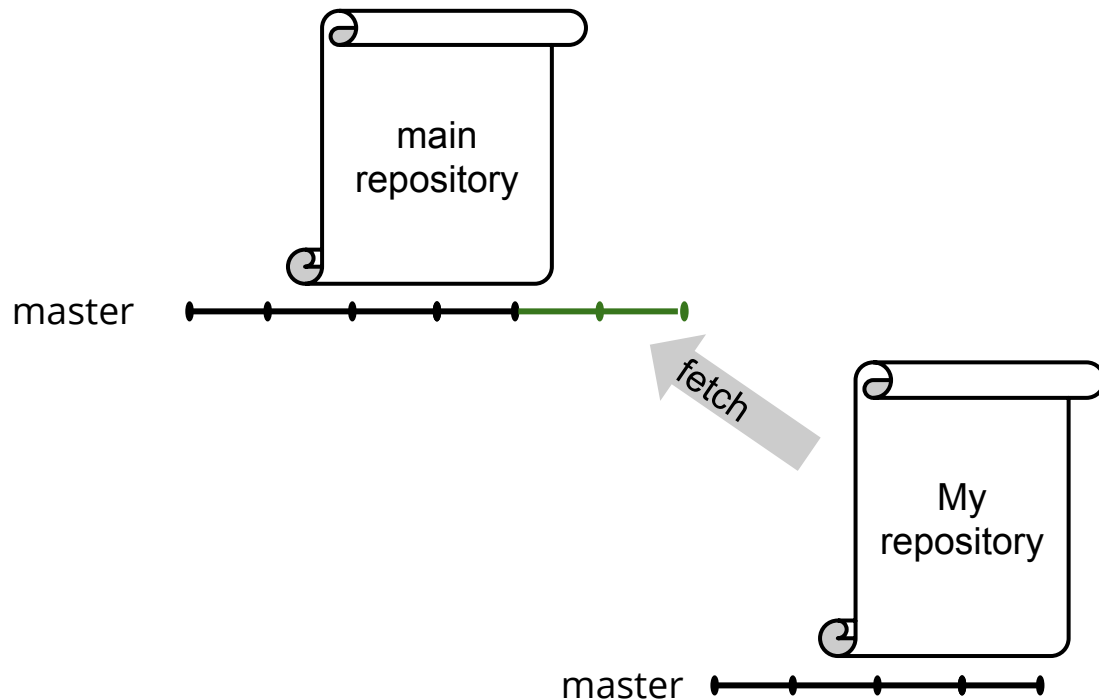
You could delete your copy and re-copy the latest... But you would lose all the work you committed to your copy!

Luckily, we're only interested in keeping the master branch in sync! Why?

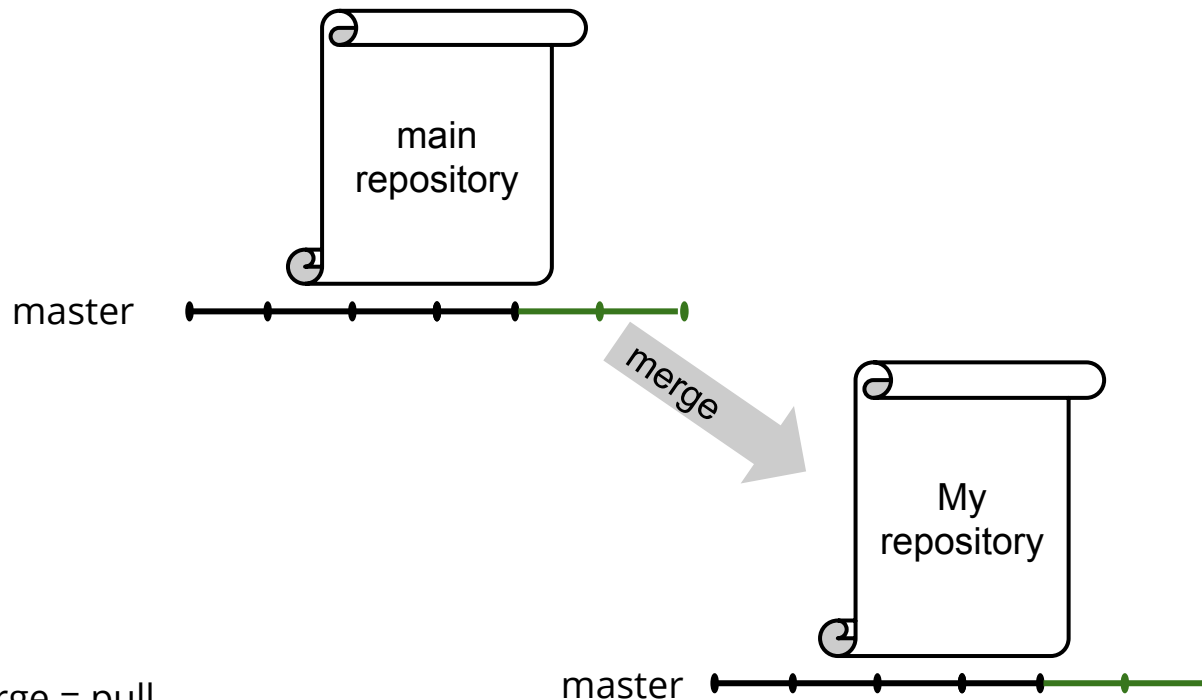
# Collaboration



# Collaboration



# Collaboration



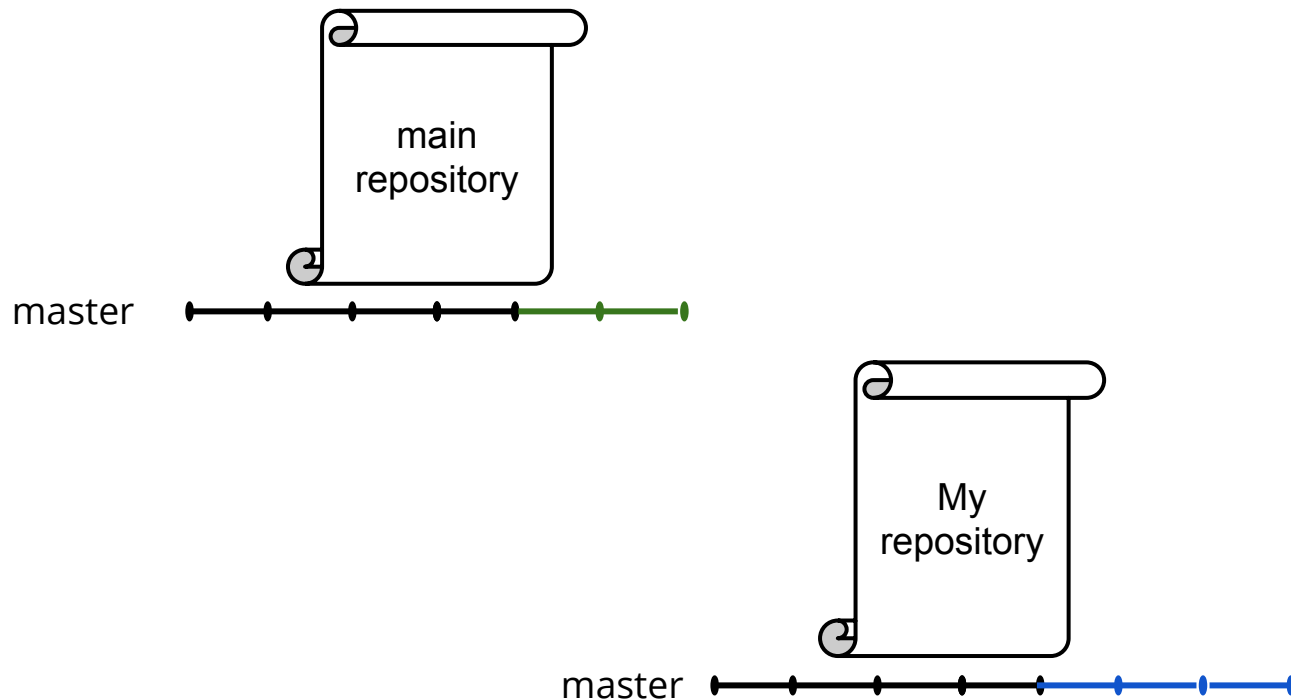
Note: fetch + merge = pull



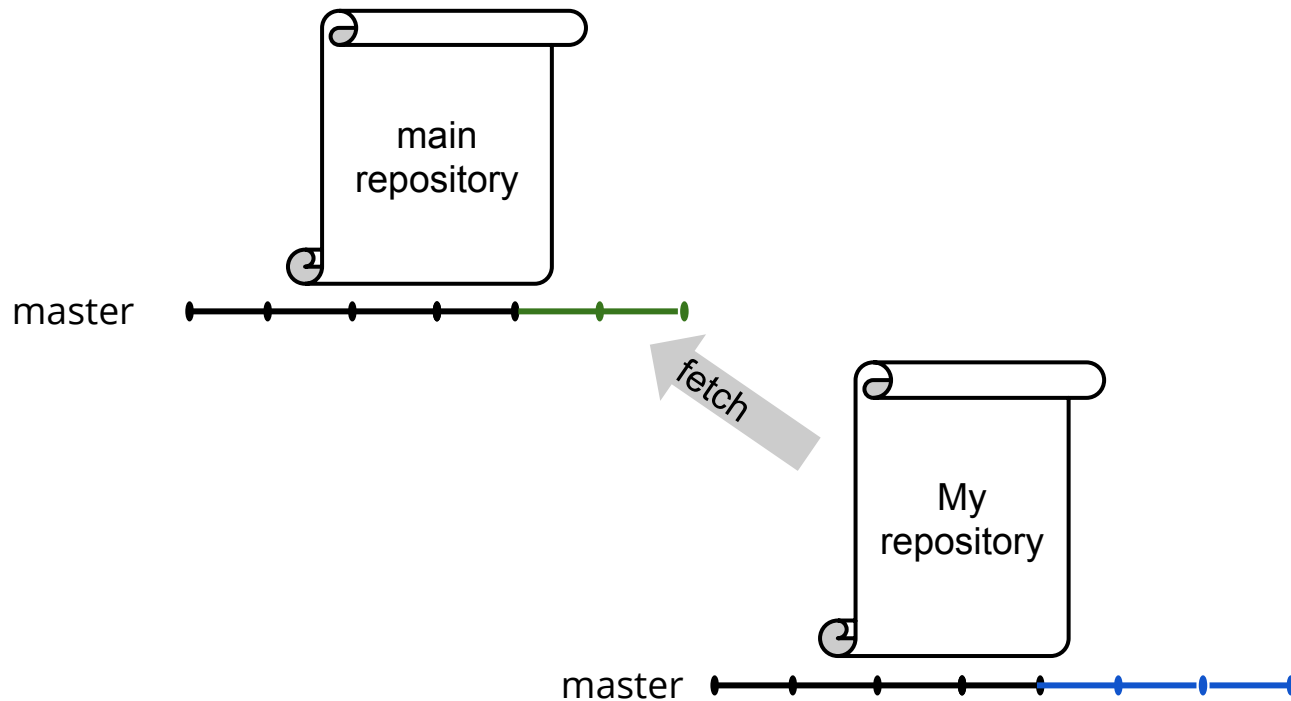
# Collaboration

This is trivial when the **base** of one branch matches the **head** of the other.

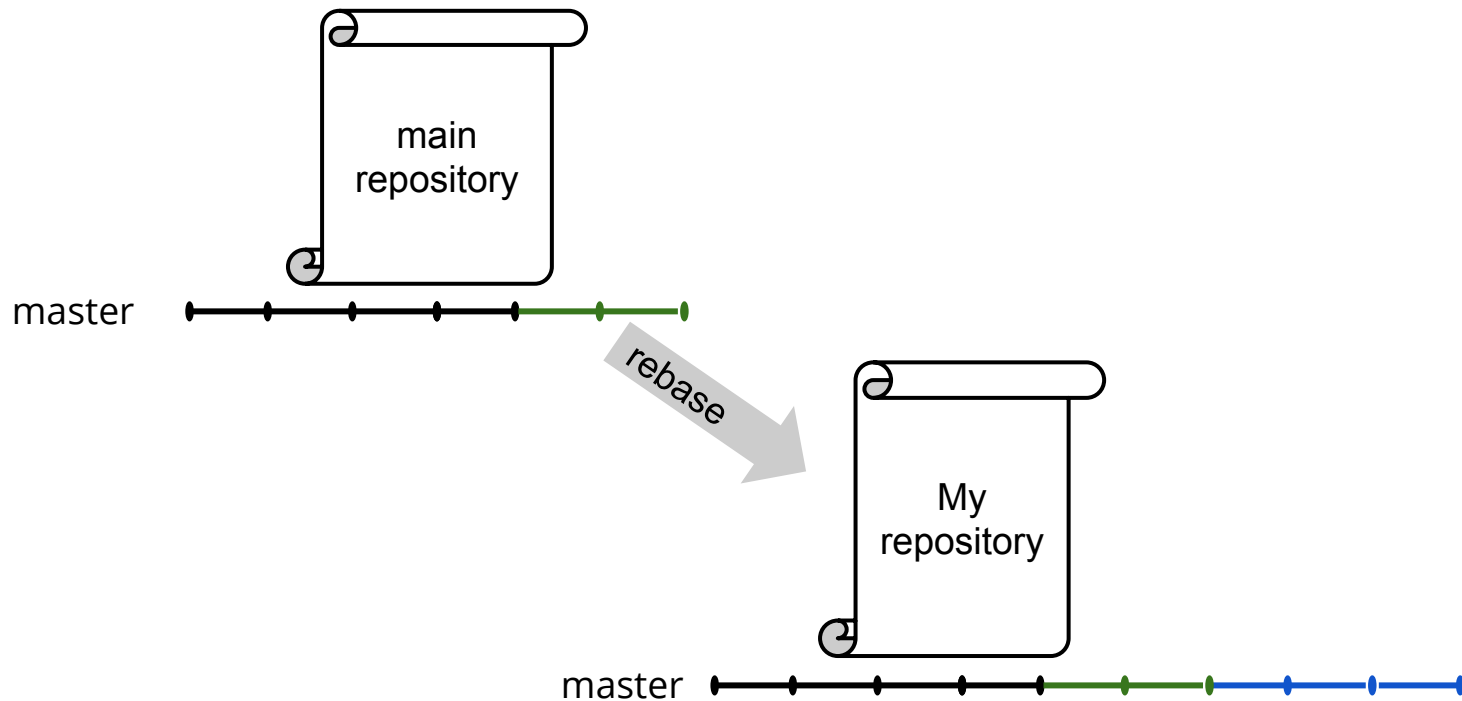
# Collaboration



# Collaboration



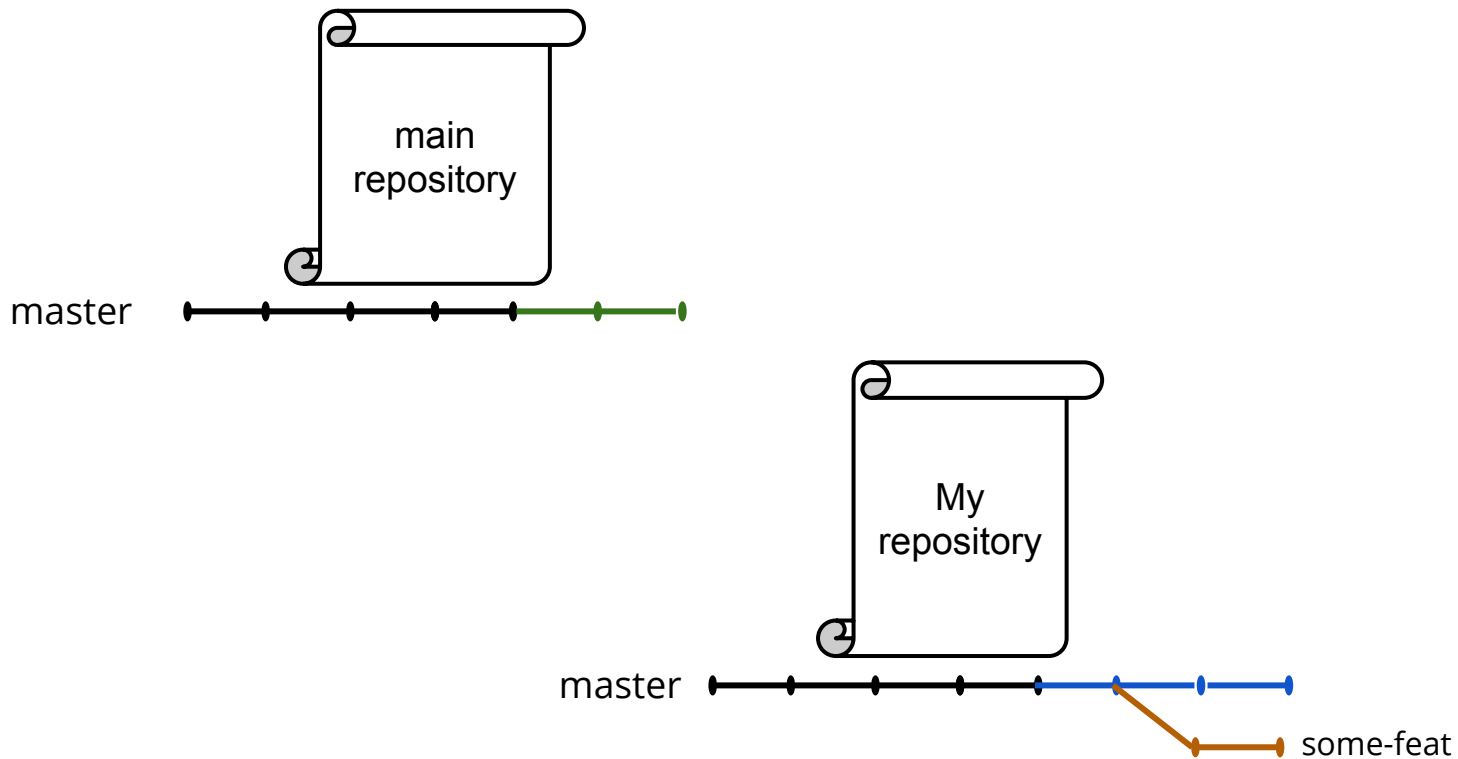
# Collaboration



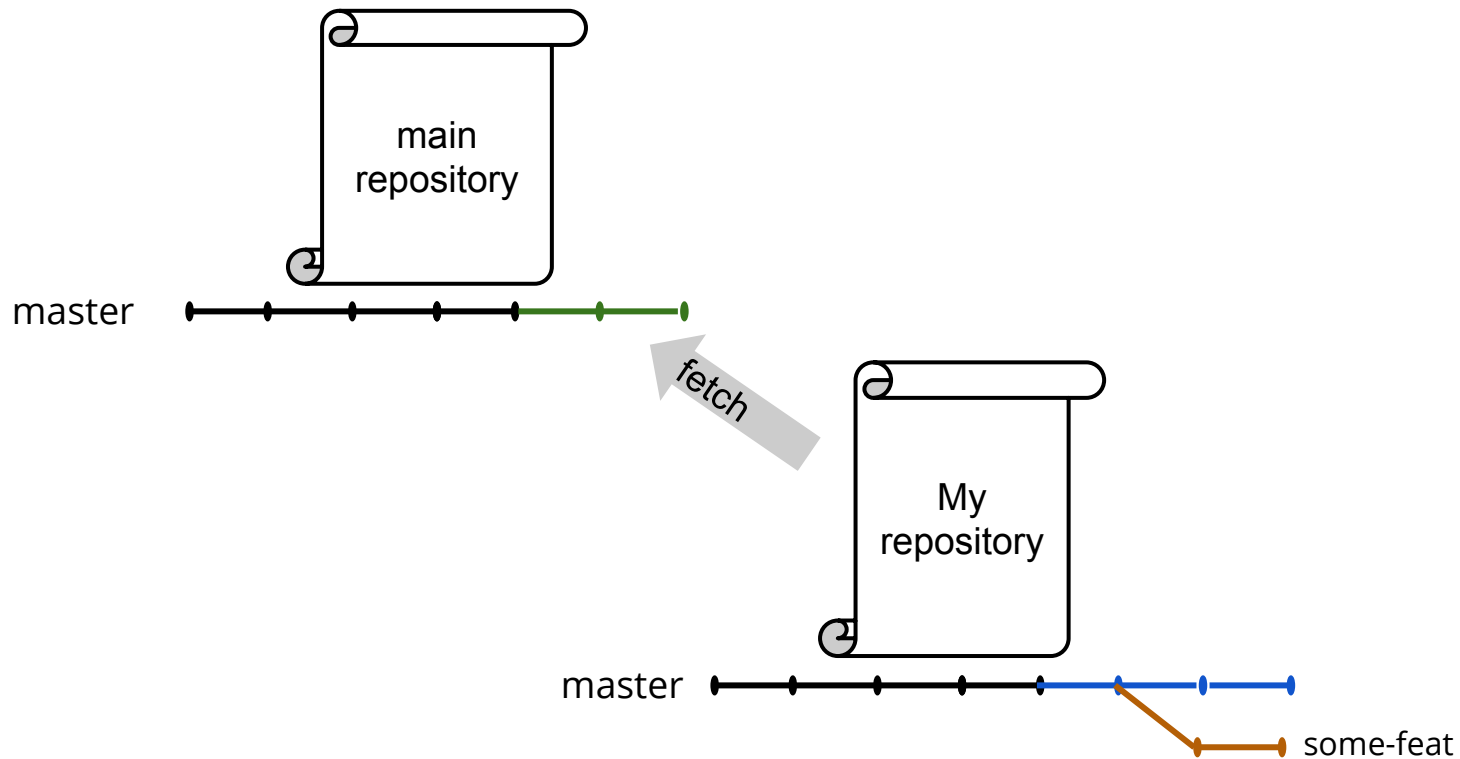
# Collaboration

What happens to other branches?

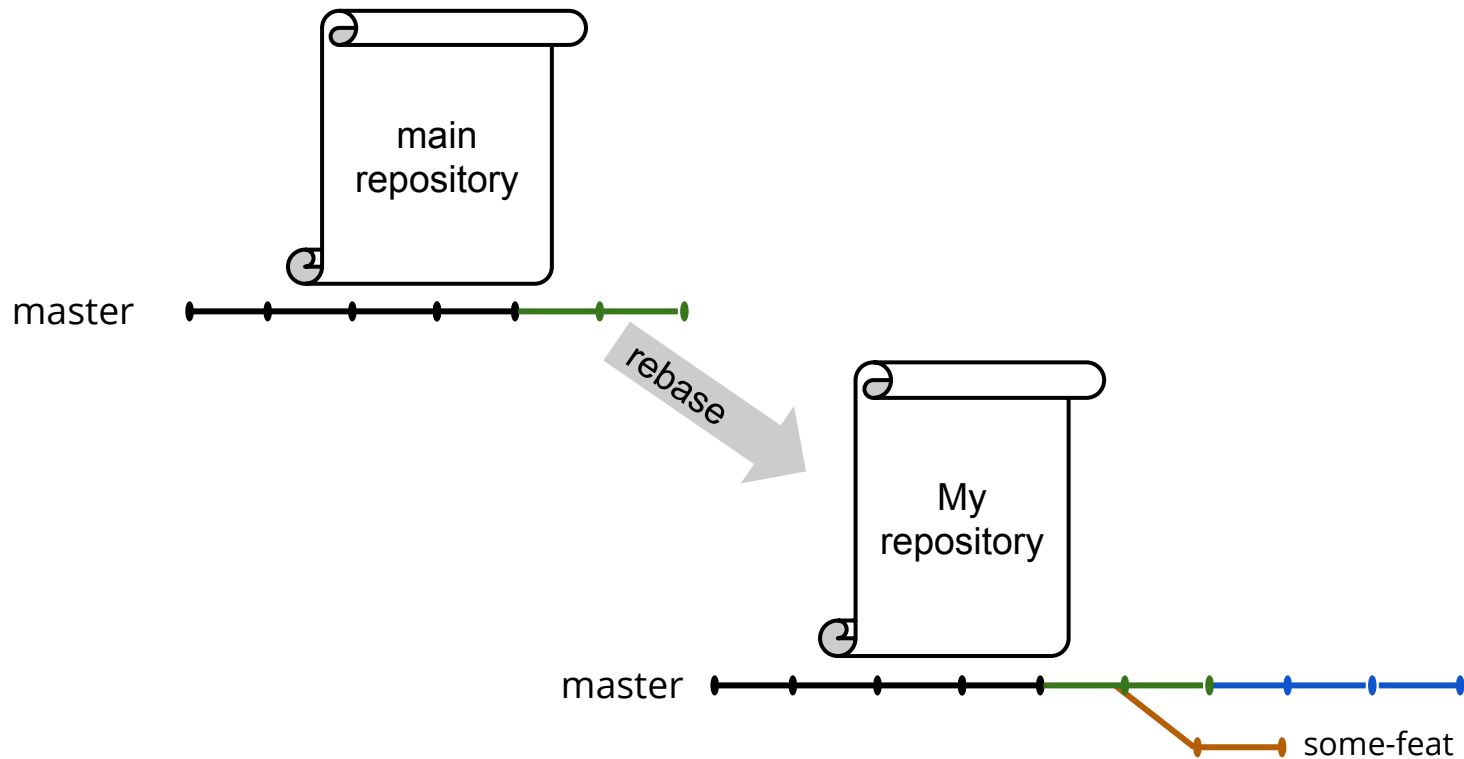
# Collaboration



# Collaboration

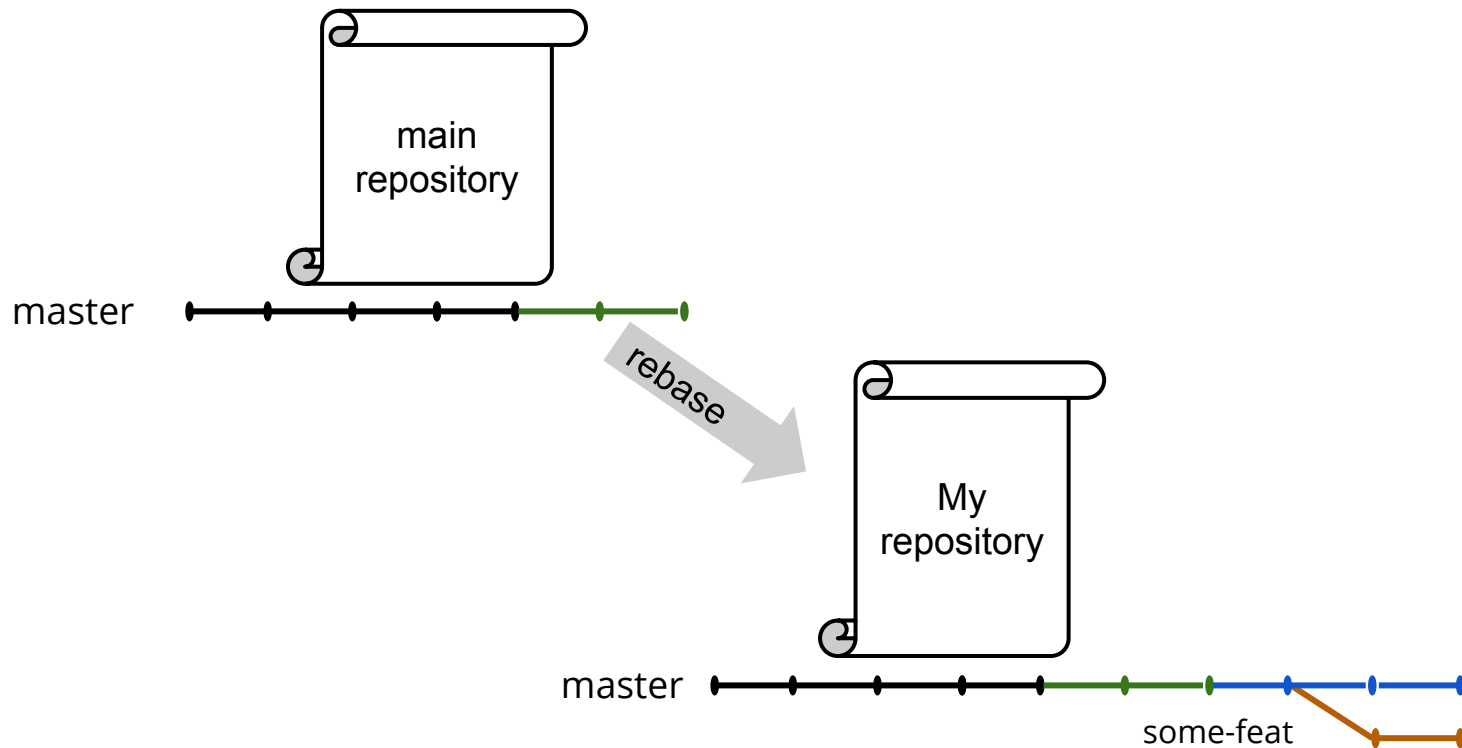


# Collaboration

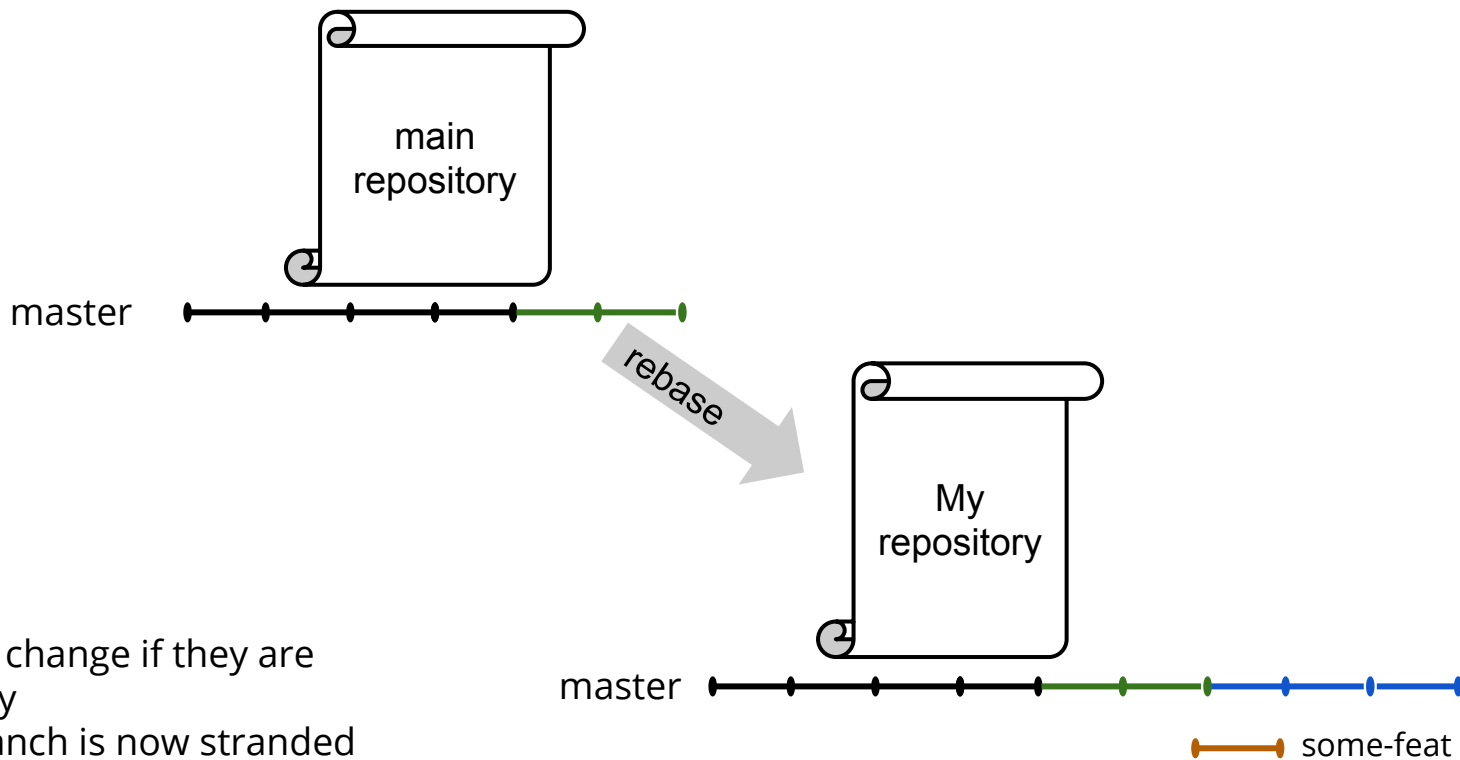




# Collaboration



# Collaboration

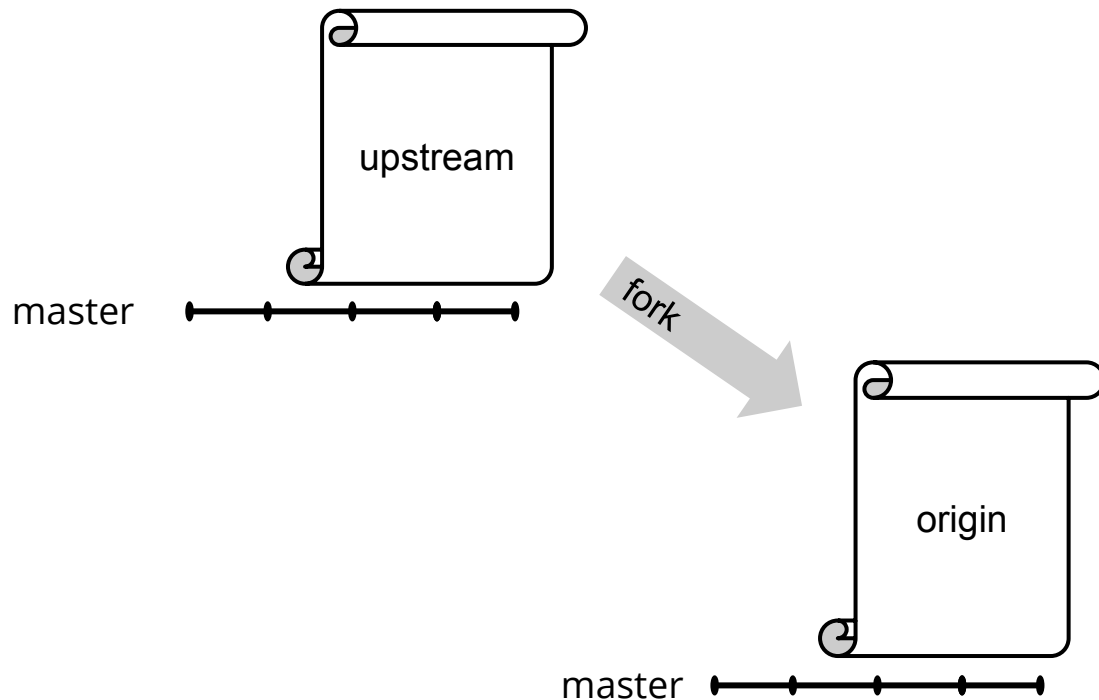


# Collaboration

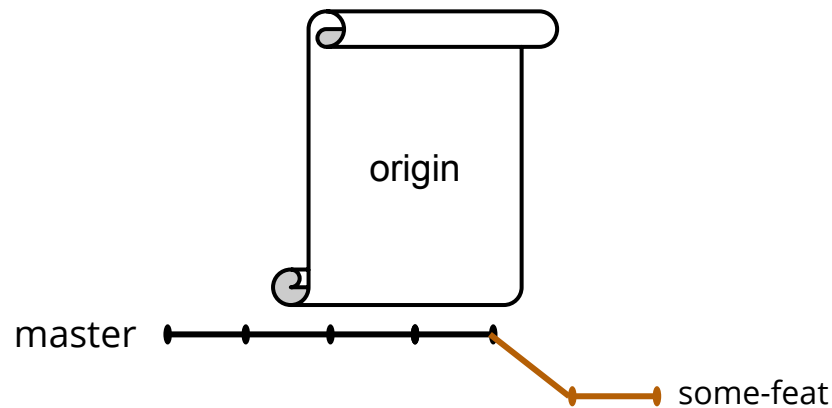
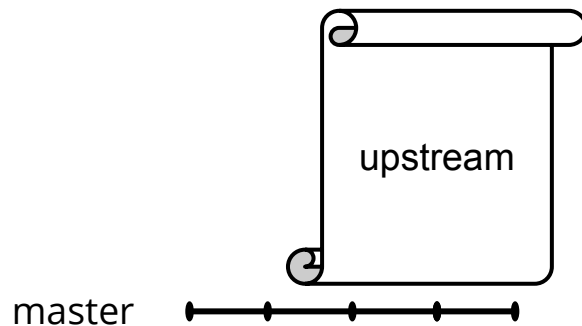
If you never commit anything to your master branch, keeping your master branch in sync with the main repository's is easy! And keeping all branches attached comes for free!

As a rule, always create a new branch when developing - **never commit directly to the master branch**

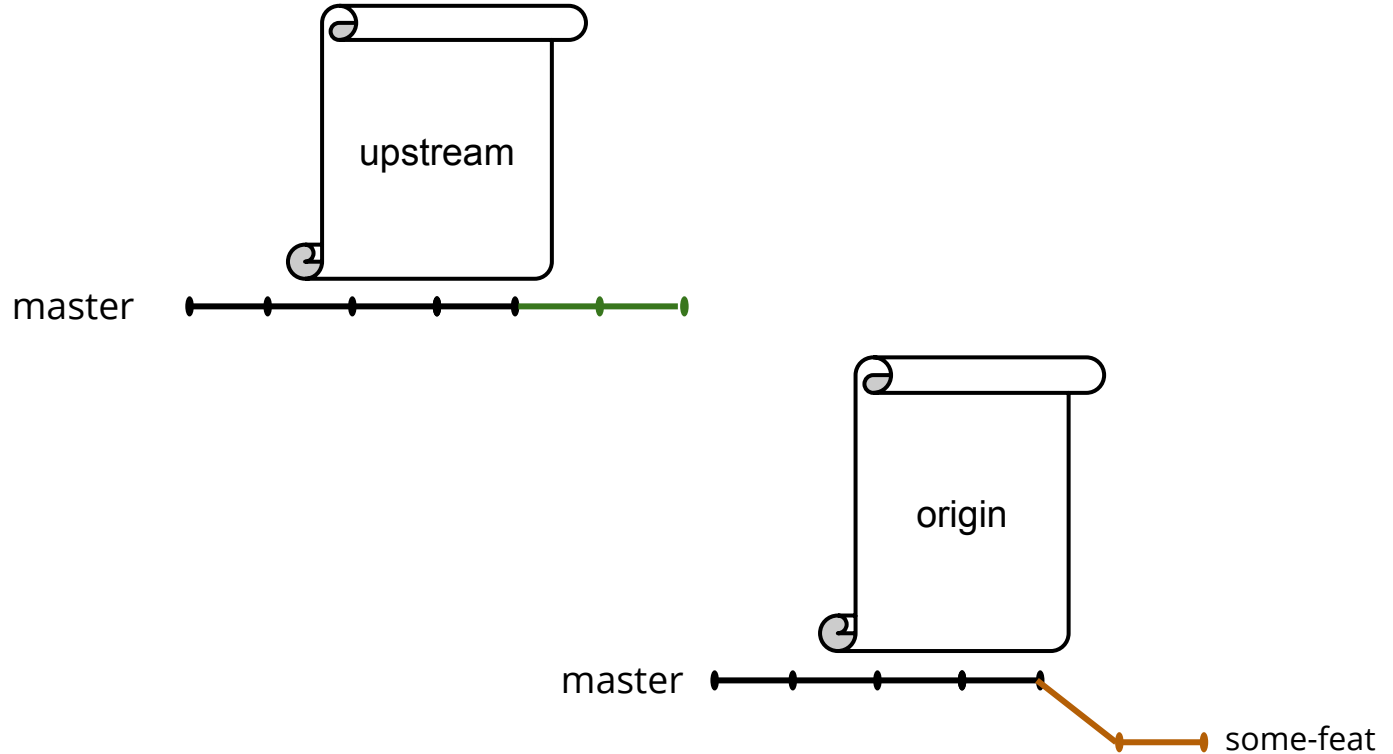
# Collaboration



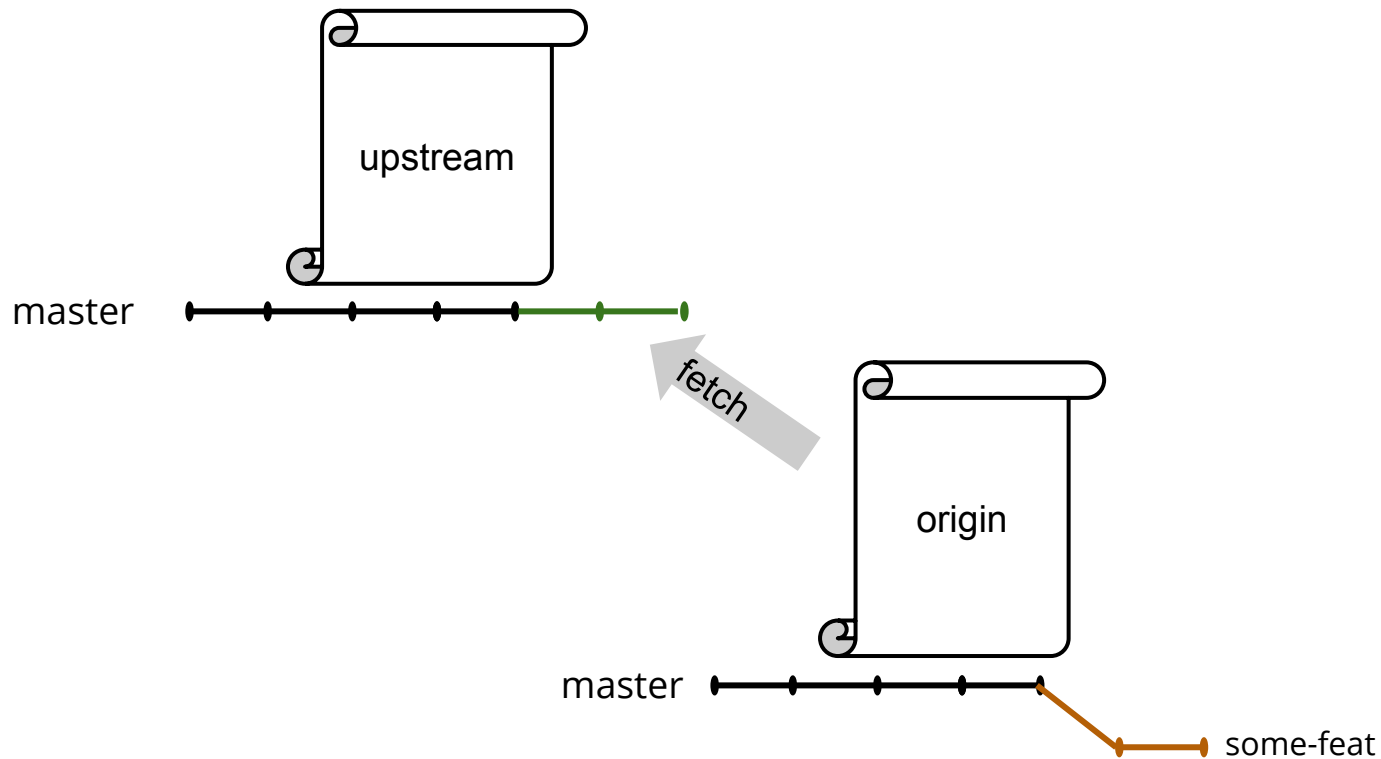
# Collaboration



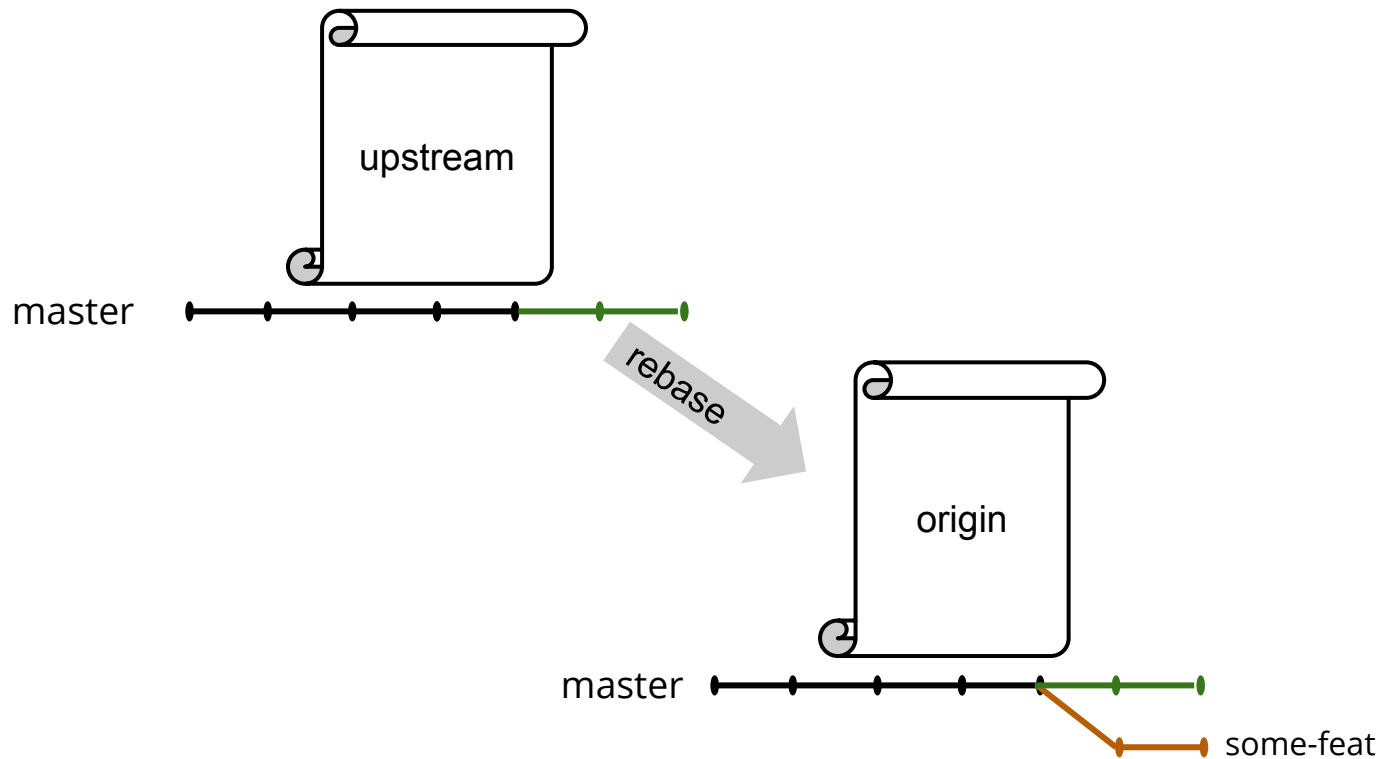
# Collaboration



# Collaboration

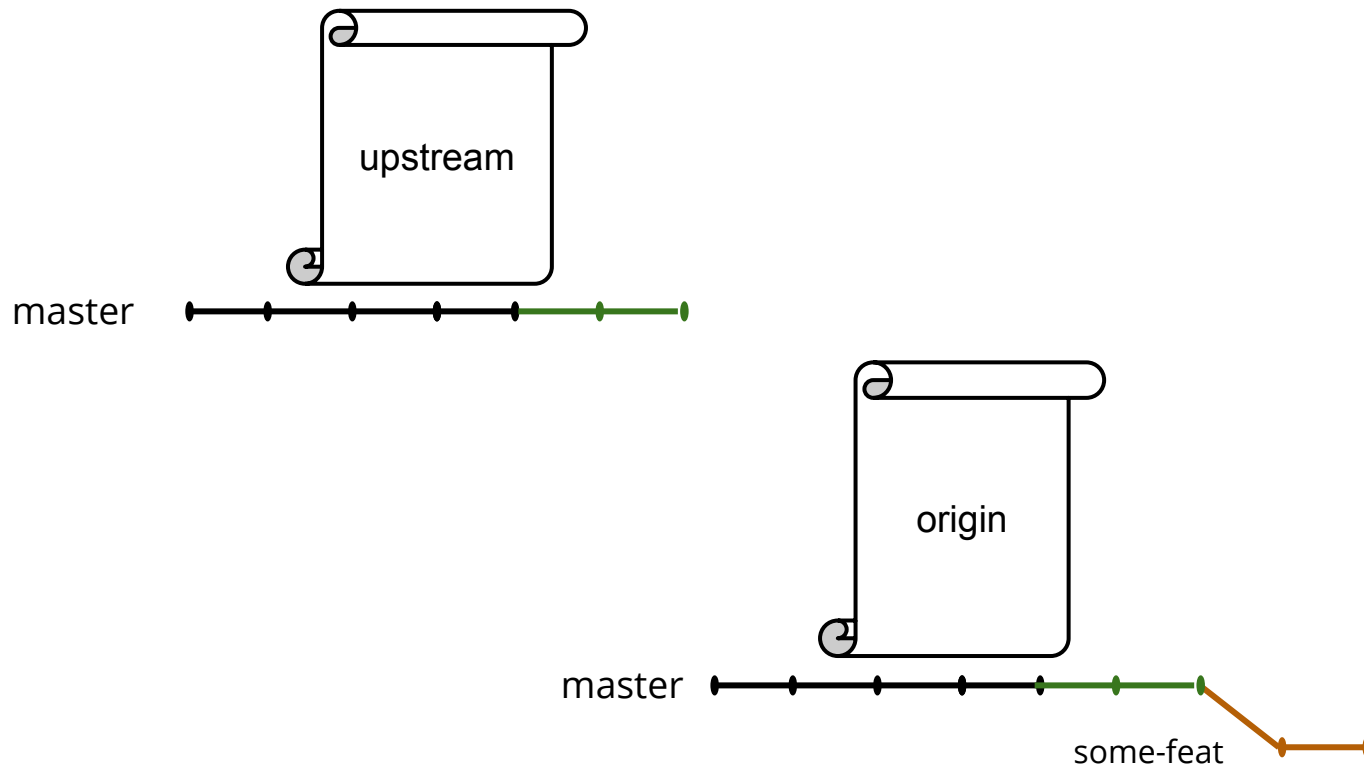


# Collaboration

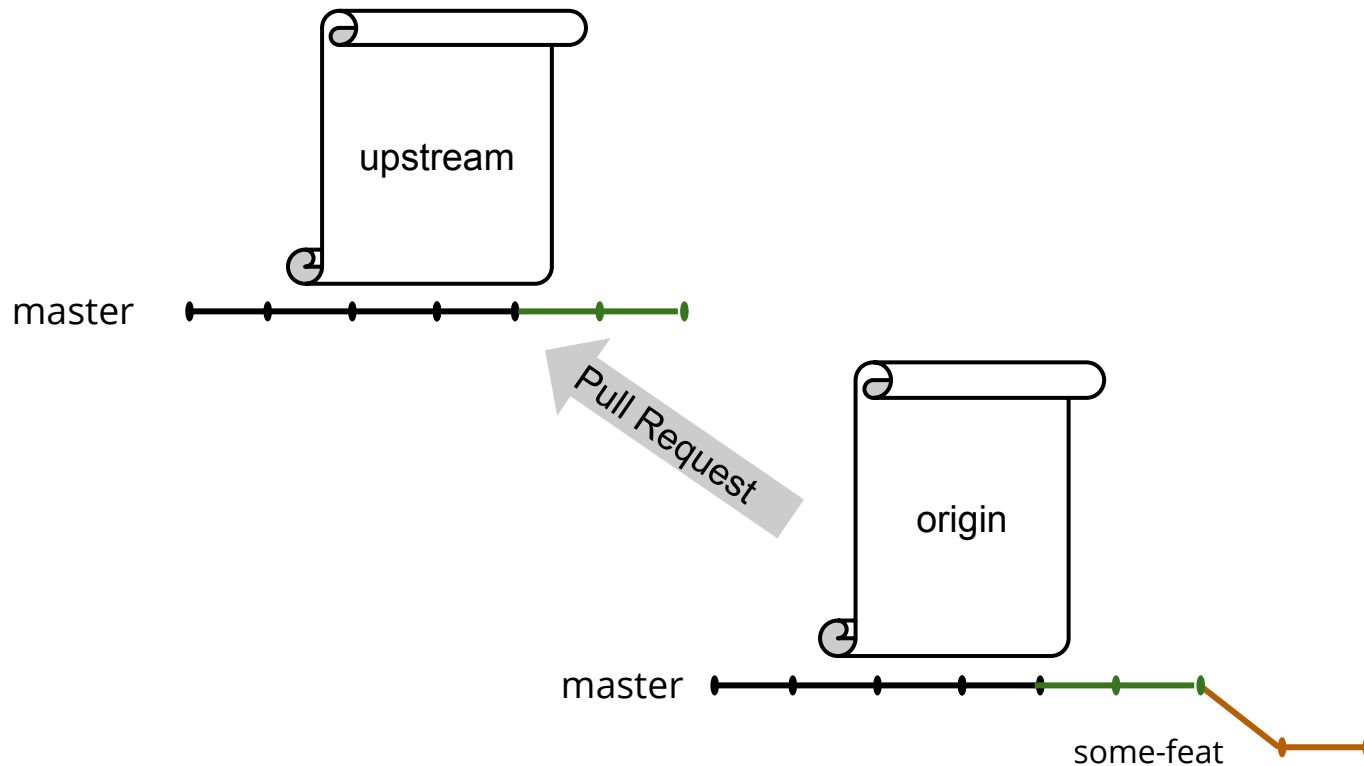




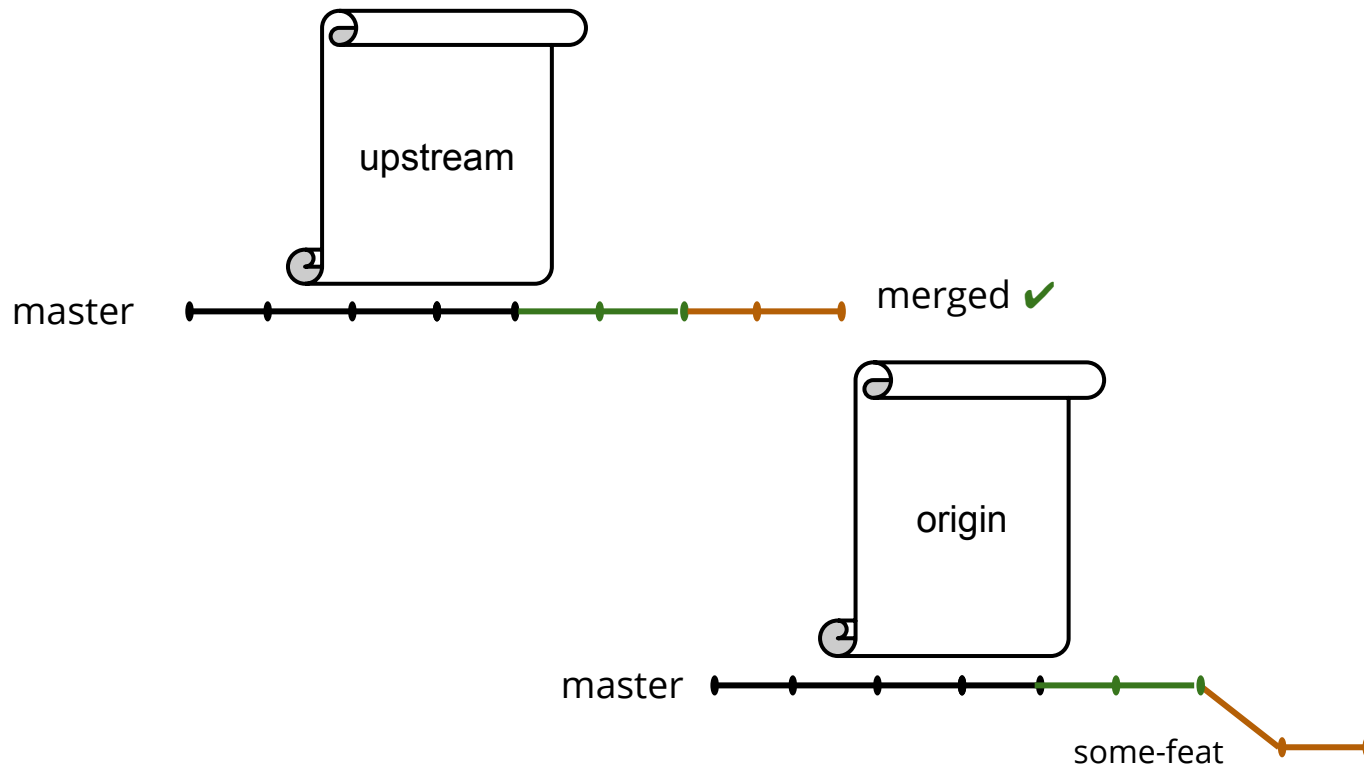
# Collaboration



# Collaboration



# Collaboration



**Demo**