

Topics

Continuous improvement

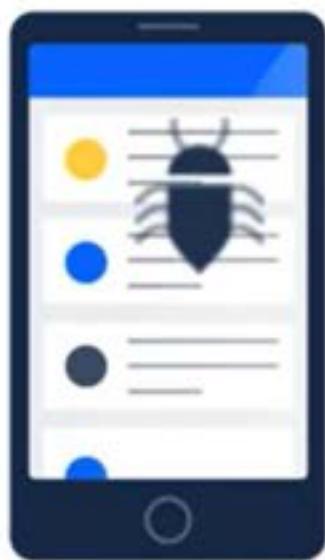
Managing project versions

Branches

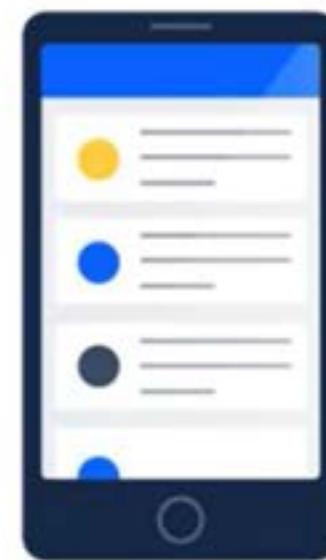
Pull requests

SMALL BATCH SIZE

Small batch size leads to continuous improvement



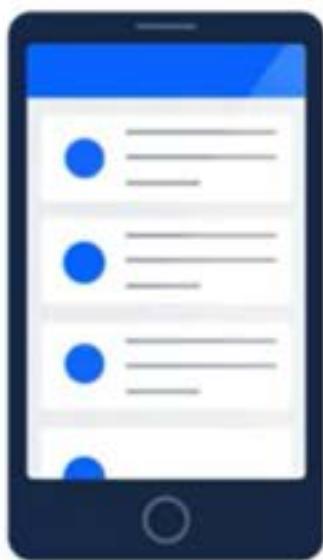
product with a bug



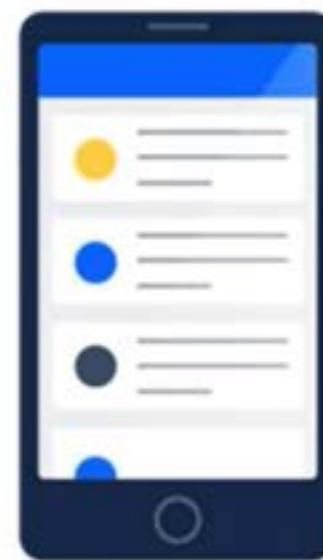
product without a bug

ADDING A FEATURE

Small batch size also applies to features



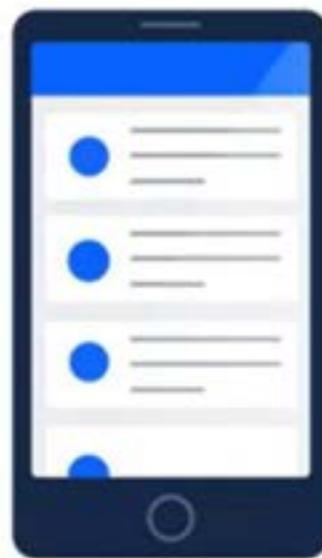
product with all blue icons



product with icon color

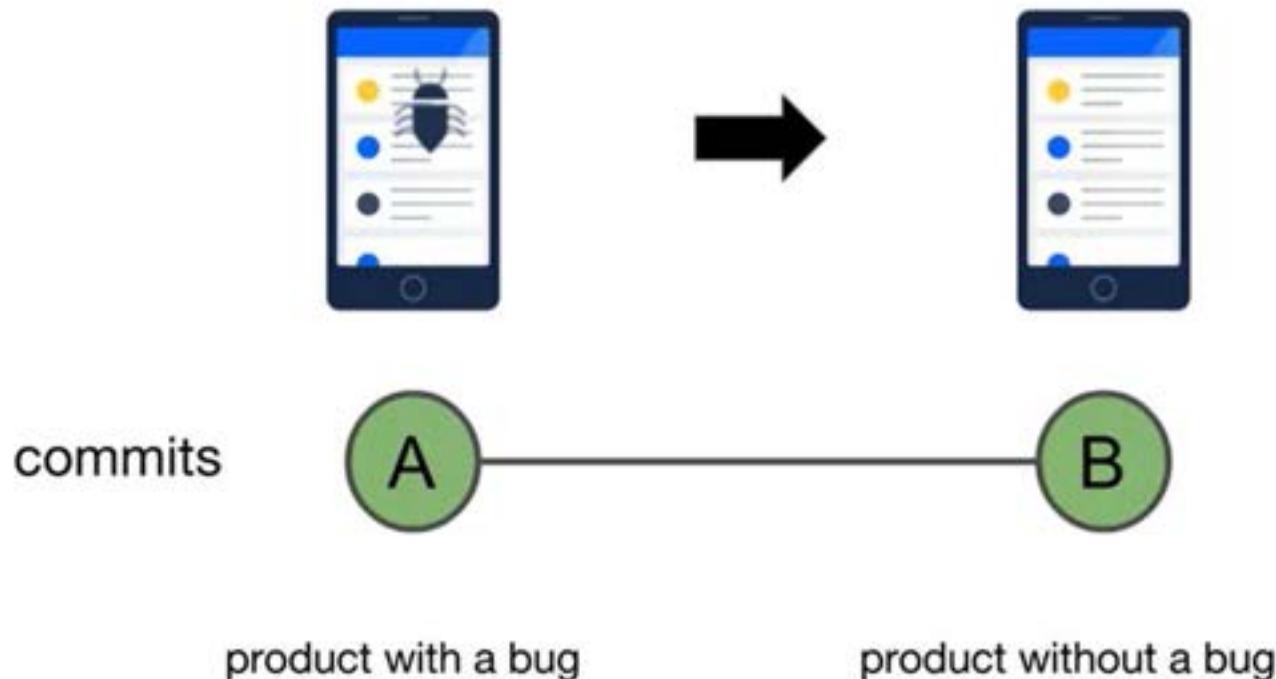
OUR PROJECT

- Product contains 50 files of code
 - This is our "project"
- We want to continuously improve it
- How does Git help?



GIT COMMITS

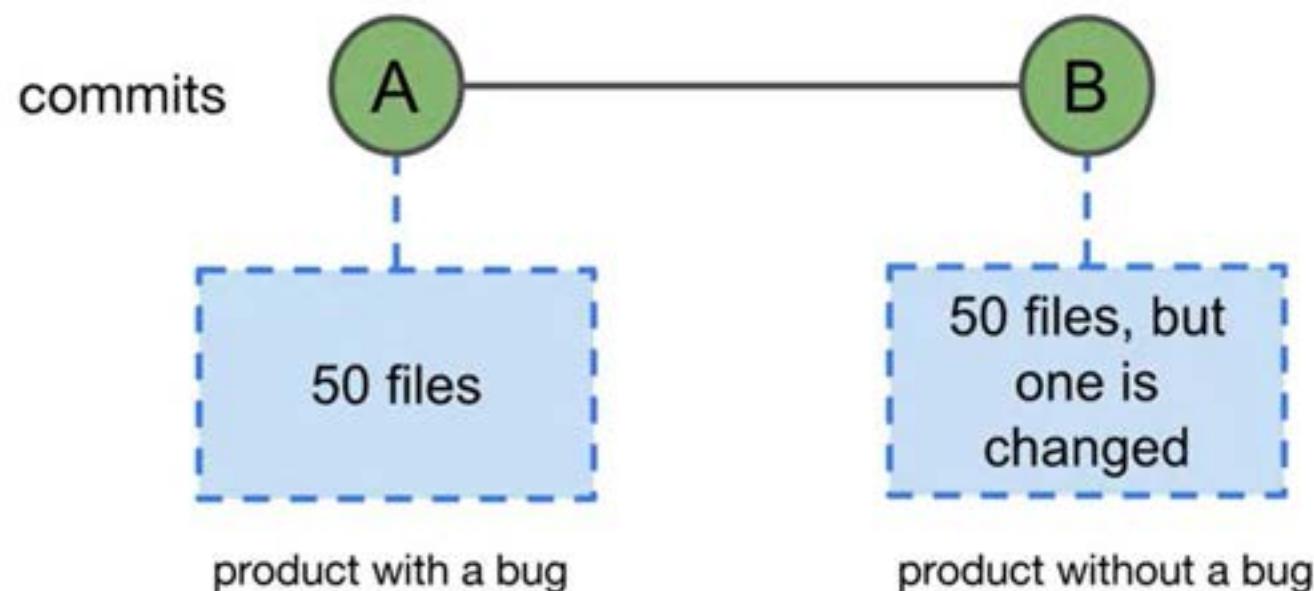
- Git manages versions of projects
- Each version of a project is called a *commit*



GIT COMMITS

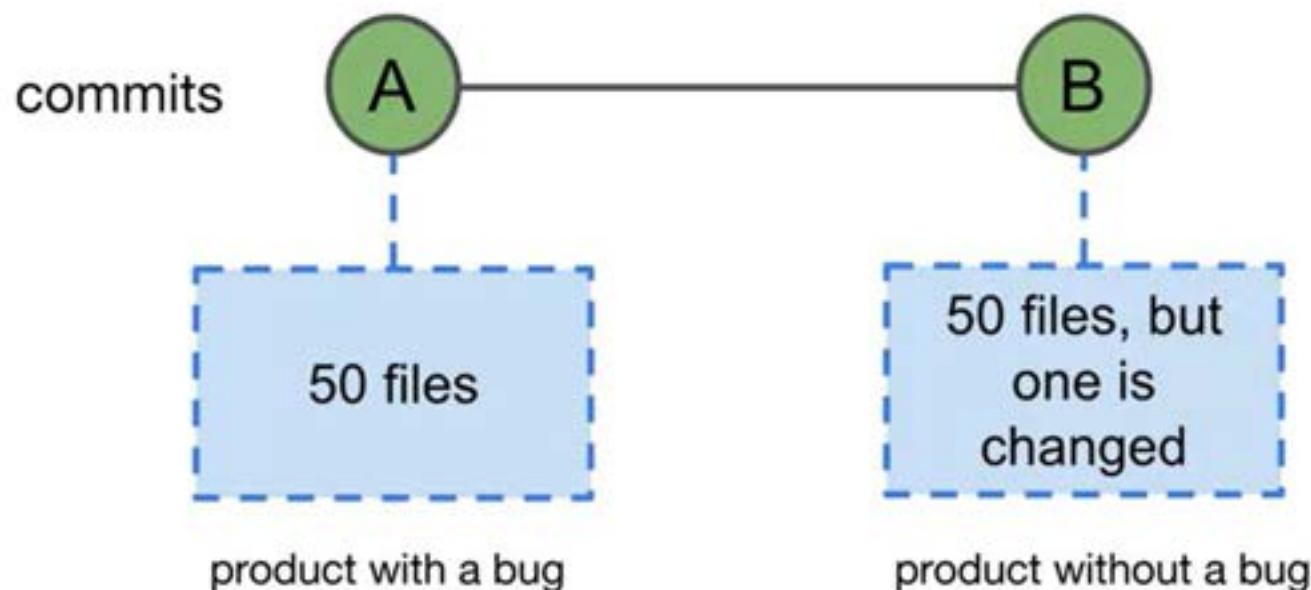
Commits A and B only differ by one file

- A small improvement in the product



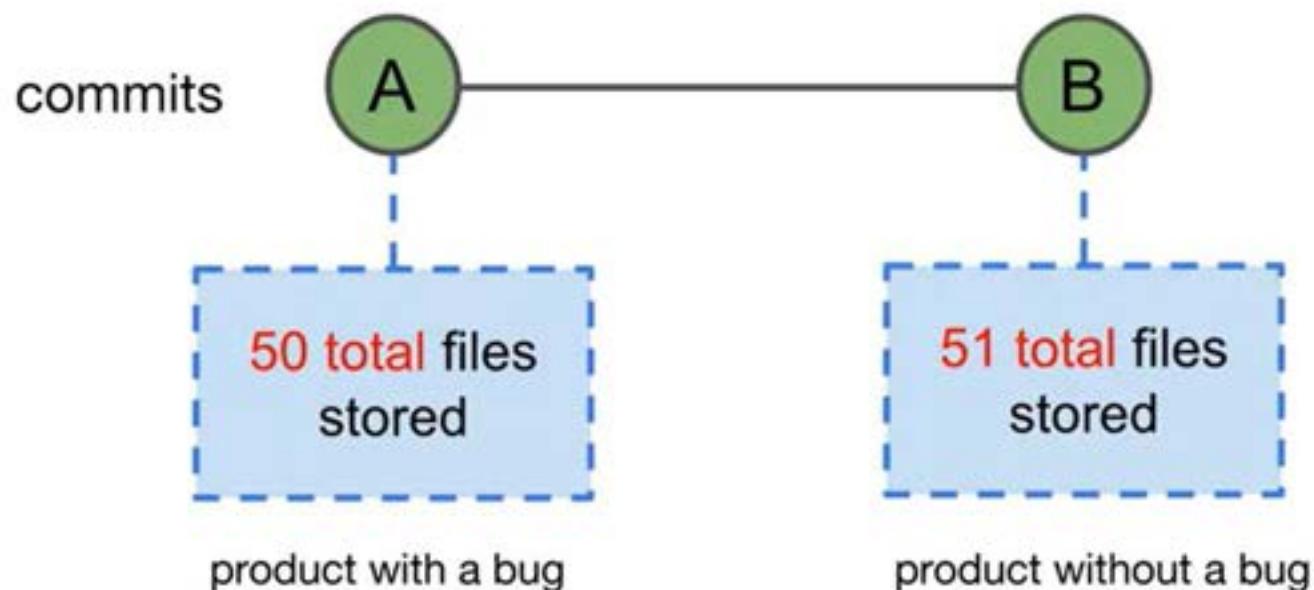
IS GIT EFFICIENT?

- Each commit is a snapshot of the entire project



IS GIT EFFICIENT?

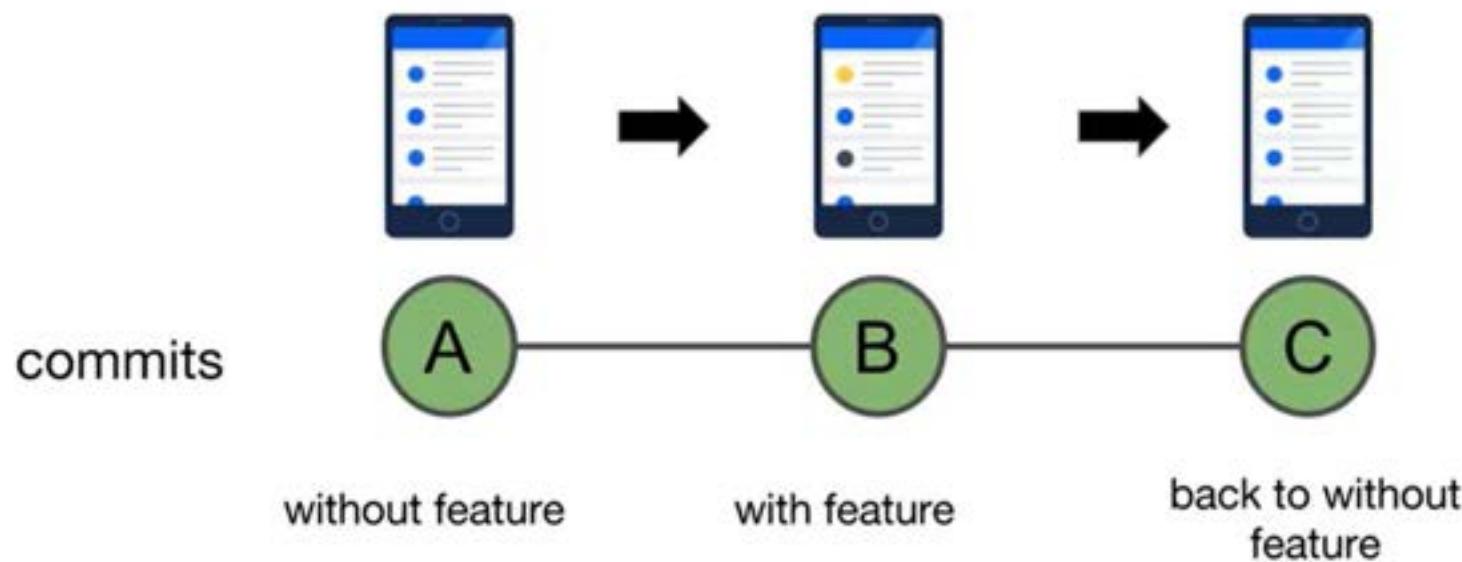
- Each commit is a snapshot of the entire project
- Behind the scenes, Git is very efficient at storing commits
 - Each unique file is stored only once



PROJECT HISTORY

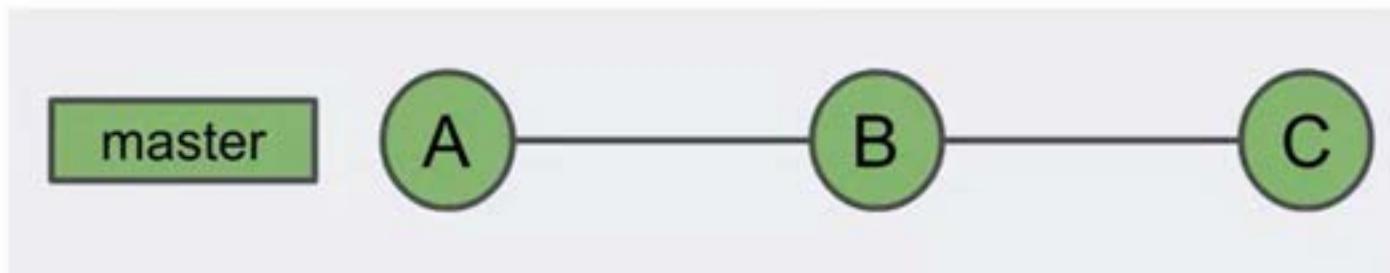
The collection of commits contain the history of the project

- You can review the history
- You can "undo" a change



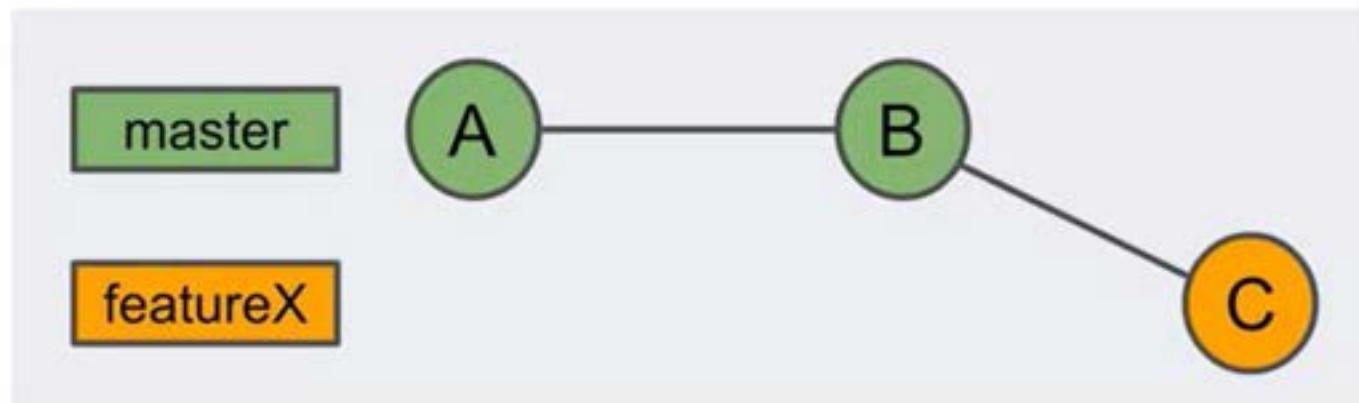
BRANCHES

- All commits belong to a *branch*
 - An independent line of development of the project
- By default, there is a single branch and it is called *master*



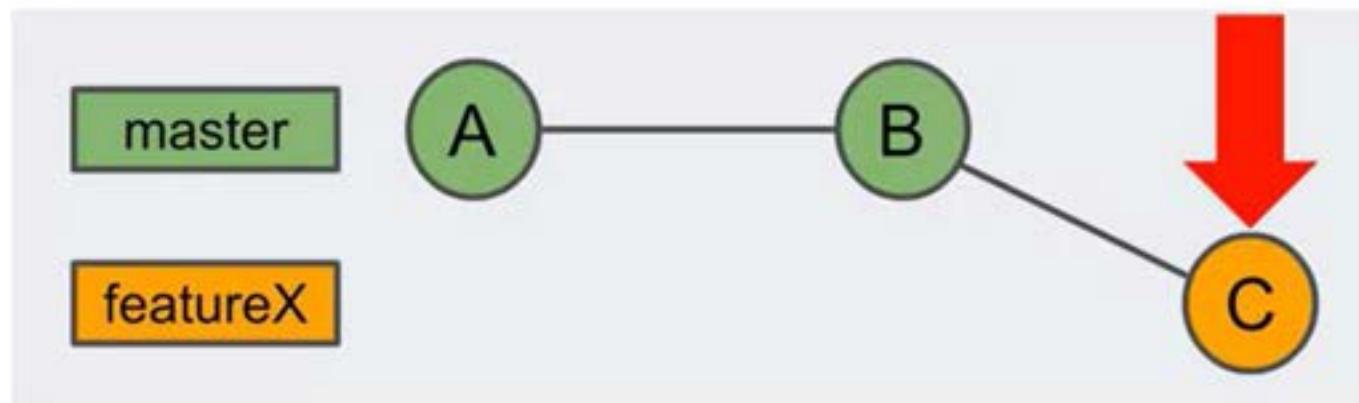
CREATING BRANCHES

- How do you maintain a stable project at the same time that you are working on it?
 - Create a separate branch

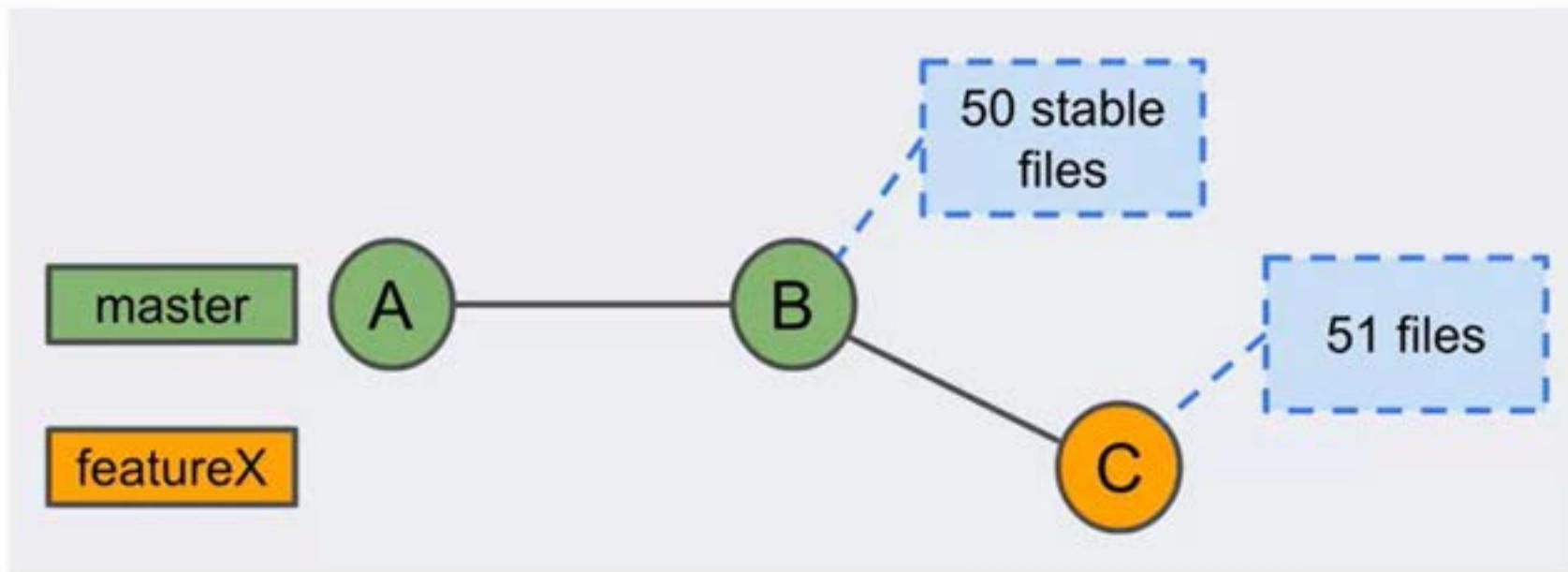


CREATING BRANCHES

- How do you maintain a stable project at the same time that you are working on it?
 - Create a separate branch
- The *master* branch does not know about the *featureX* branch

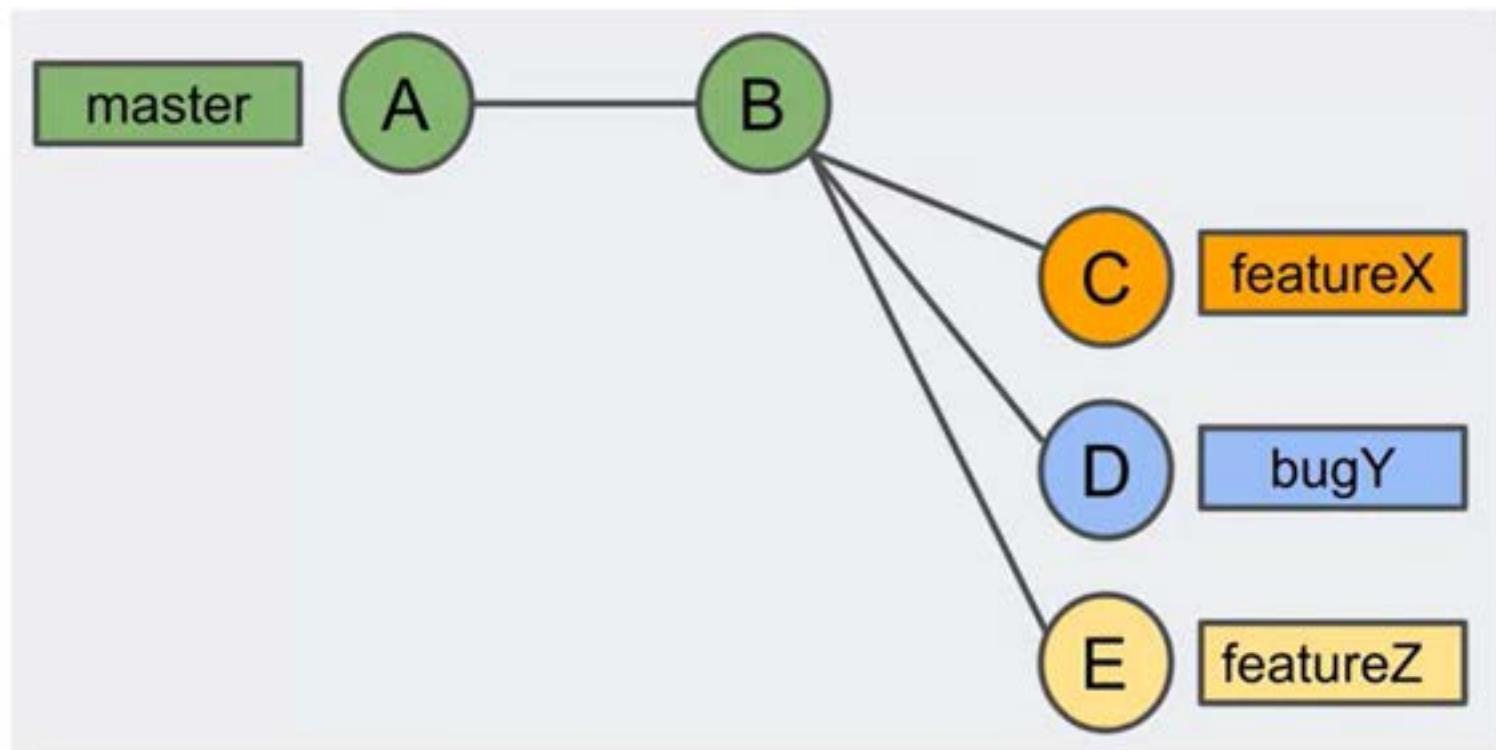


BRANCH INDEPENDENCE



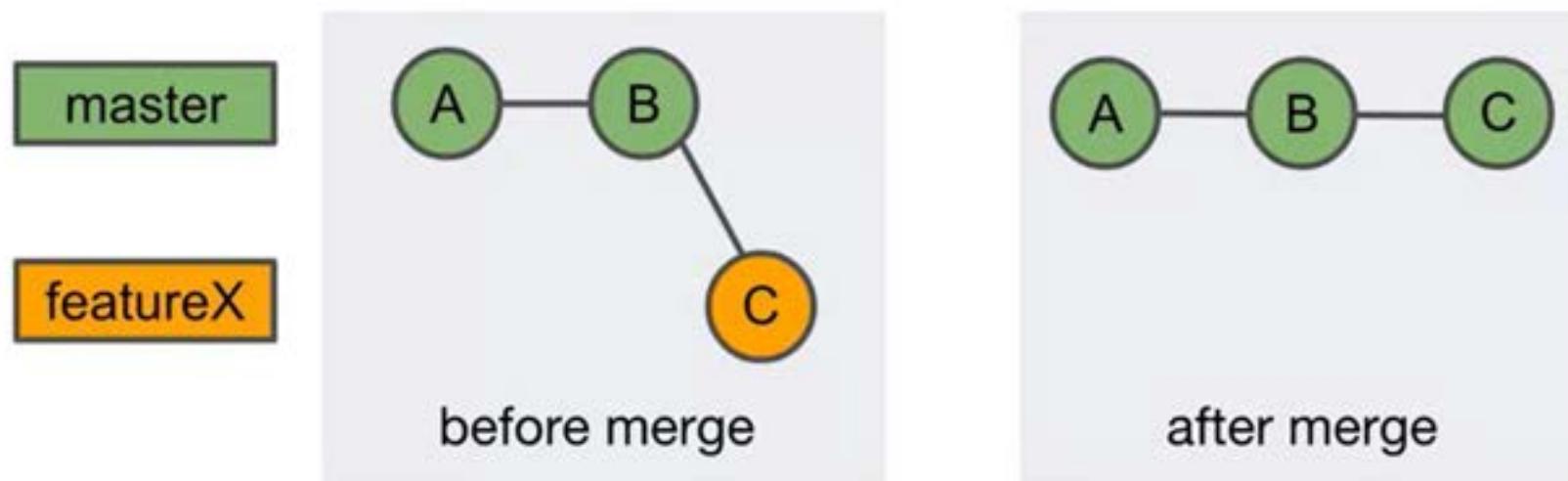
TEAMS

The independence of branches allows teams to scale their work



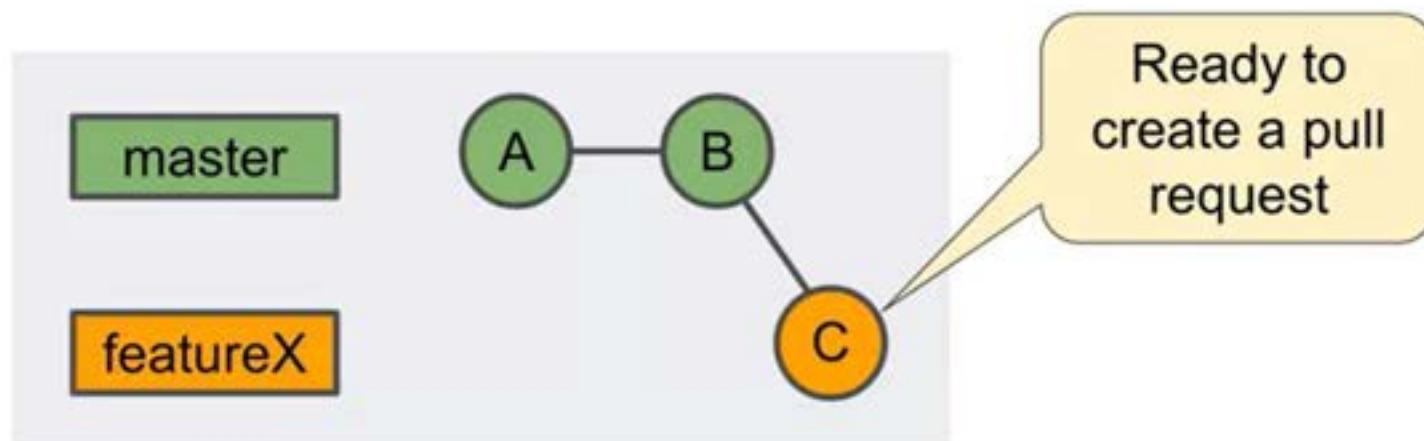
MERGING

After a *merge*, the *master* branch contains the new feature



PULL REQUEST

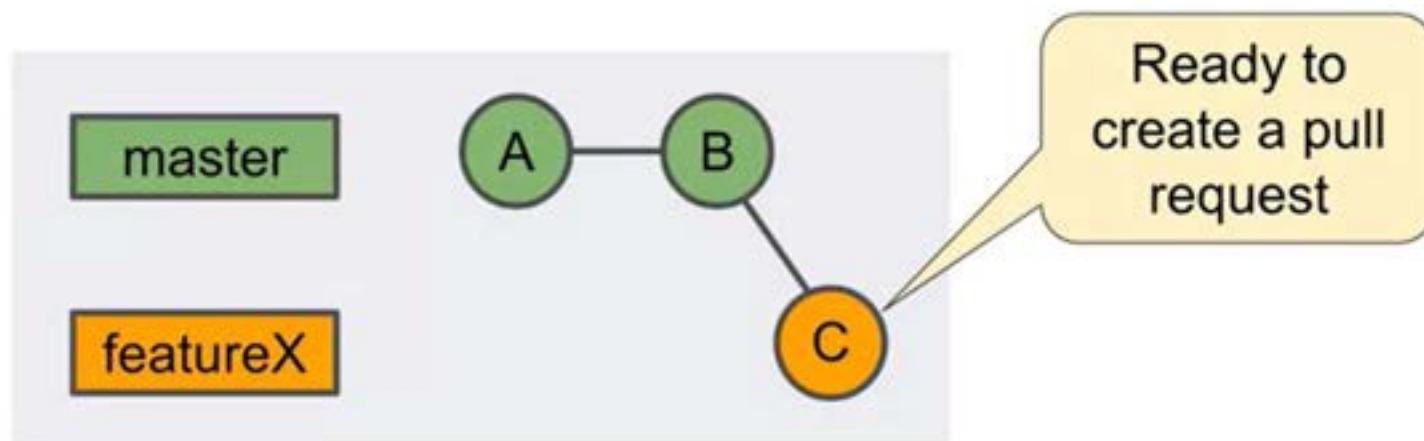
Request to merge your branch into another branch



PULL REQUEST

Request to merge your branch into another branch

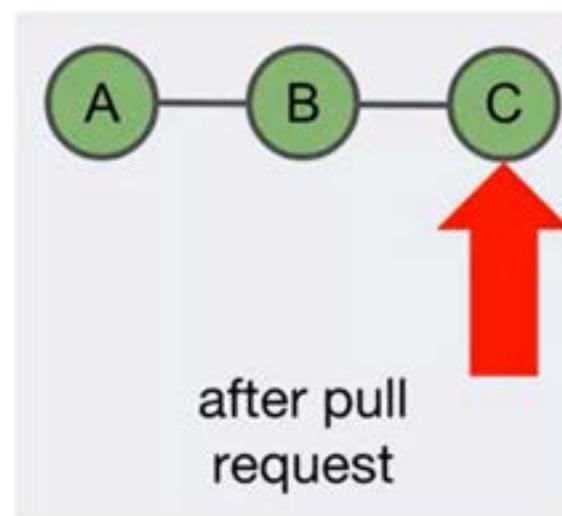
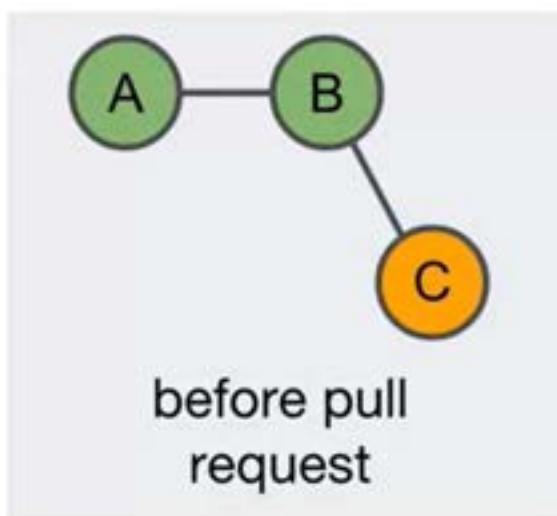
- Team members can discuss, review and approve your changes
- Can require passing automated tests



PULL REQUEST ACCEPTED

master

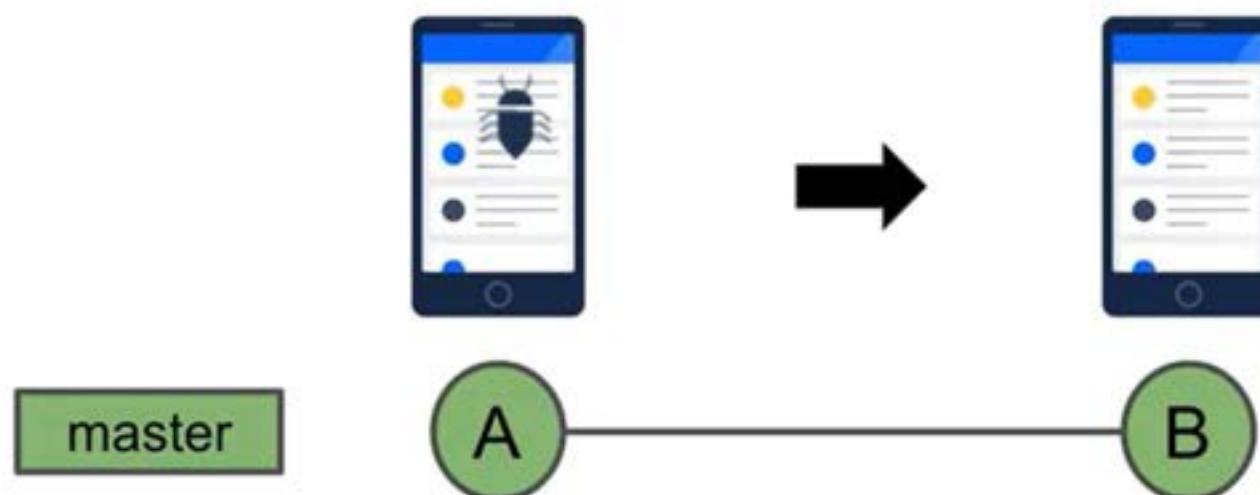
featureX



- ✓ reviewed
- ✓ tested

REVIEW

- Continuous improvement- **commits**
- Simultaneous stability and development- **branches**
- Improved quality- **pull requests**



Git Overview

Copyright © 2017 Atlassian

Topics

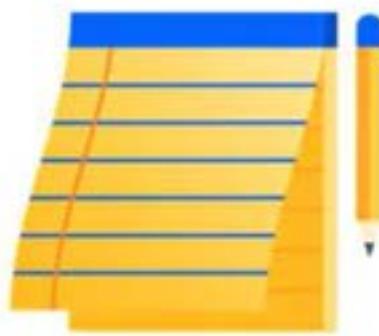
Version control overview

Git overview

Command line vs. user interface

WHAT IS VERSION CONTROL?

content



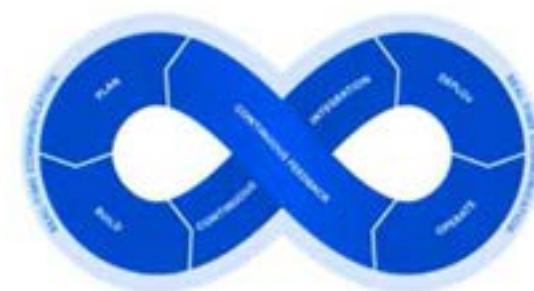
- Complete history tracked and available

teams



- Supports many workflows
- Collaboration
- Quality through team communication and reviews

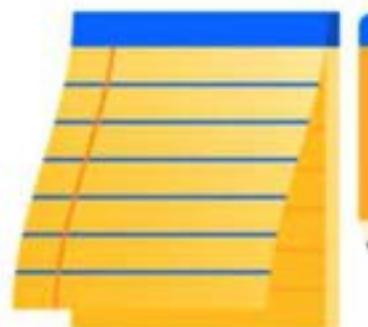
agility



- Manages small changes
- Easily test, fix or undo ideas and changes

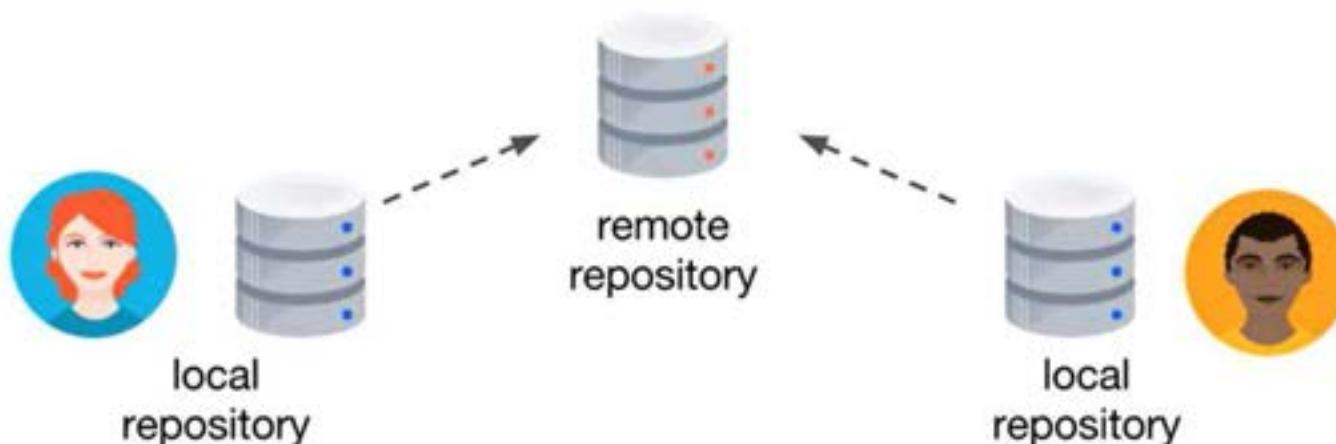
WHAT TYPE OF CONTENT?

- Source code
- Automated tests
- Server configuration
- Documentation
- A book
- Web site content



DISTRIBUTED VERSION CONTROL SYSTEM (DVCS)

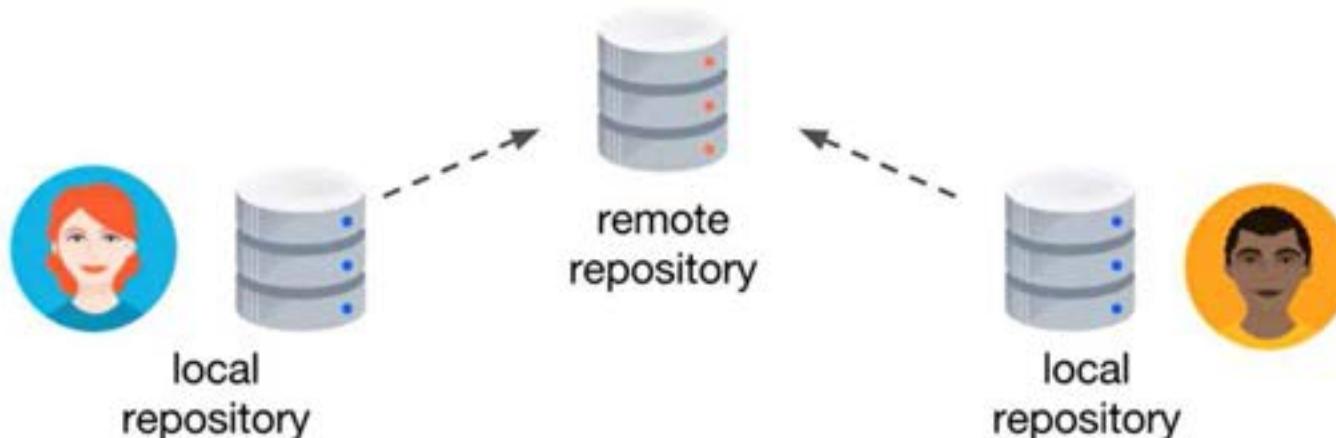
A DVCS usually has these characteristics:



DISTRIBUTED VERSION CONTROL SYSTEM (DVCS)

A DVCS usually has these characteristics:

- Each user has a local project history (repository)
- Users can work offline
- Can easily synchronize repositories



WHAT IS GIT?



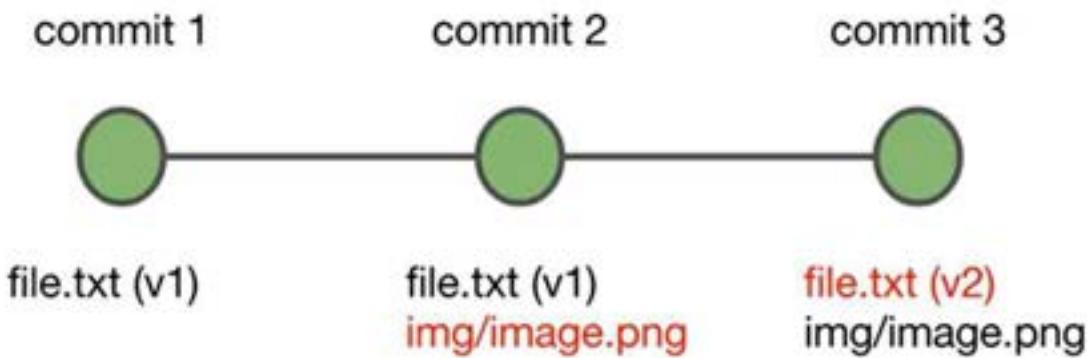
- Git is a distributed version control system
- Open source software (OSS)
 - Has a vibrant community and ecosystem
- Adapts to many types of projects and workflows
 - Works well for large or small projects

<https://git-scm.com/>

WHAT IS A GIT REPOSITORY?



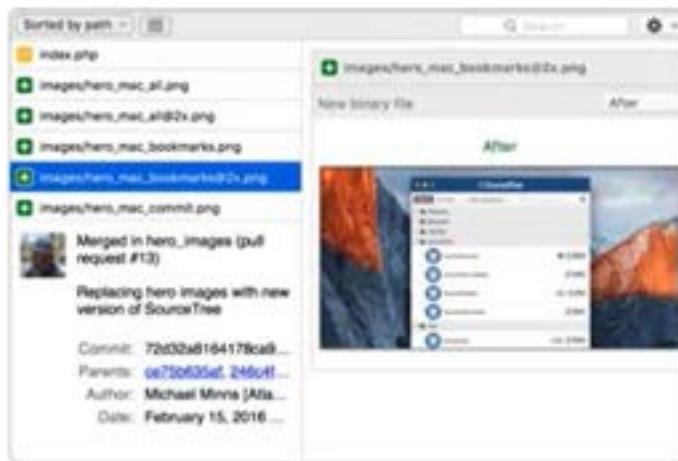
A series of *snapshots*, or *commits*



COMMAND LINE VS. SOURCETREE

```
$ git --version  
git version 2.14.1
```

command line



Sourcetree

SHOULD YOU USE THE COMMAND LINE?

```
$ git --version  
git version 2.14.1
```

command line

SHOULD YOU USE THE COMMAND LINE?

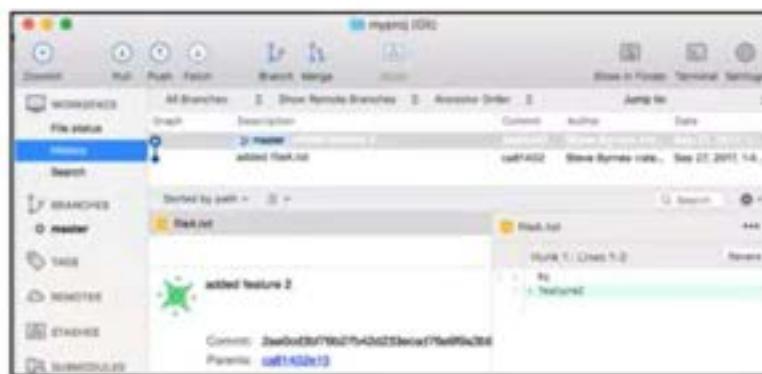
- Command line skills are assumed by the industry
- Command line = automatable
- Fast and easy

```
$ git --version  
git version 2.14.1
```

command line

SHOULD YOU USE SOURCETREE?

- You do not currently have command line knowledge
- Some tasks may be easier with a user interface
- You will not use Git often



Sourcetree

REVIEW

- Version control enables teams to manage a collection of files in an agile way
- Git is a distributed version control system
 - Each user has a local copy of a Git repository
- A repository contains the project history as commits
 - A commit is a snapshot of the entire project
- You have the choice of working with Git using a command line and/or a graphical interface



Topics

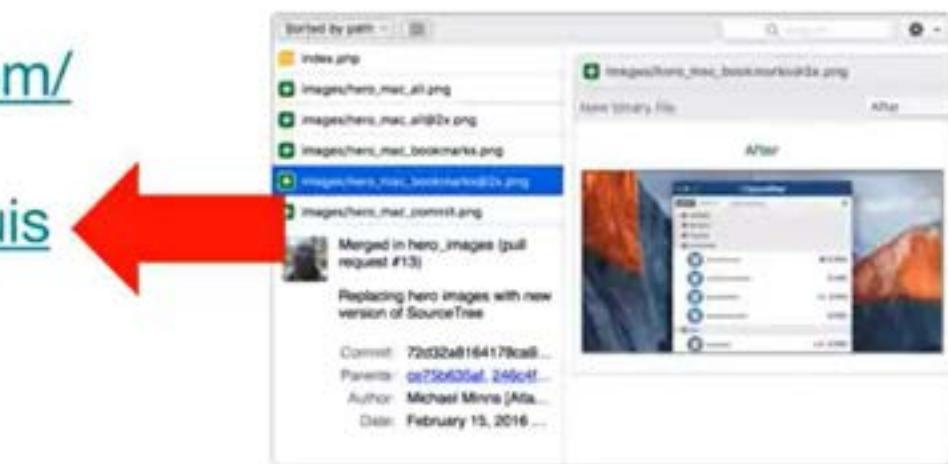
Installing Sourcetree

Configuring user information

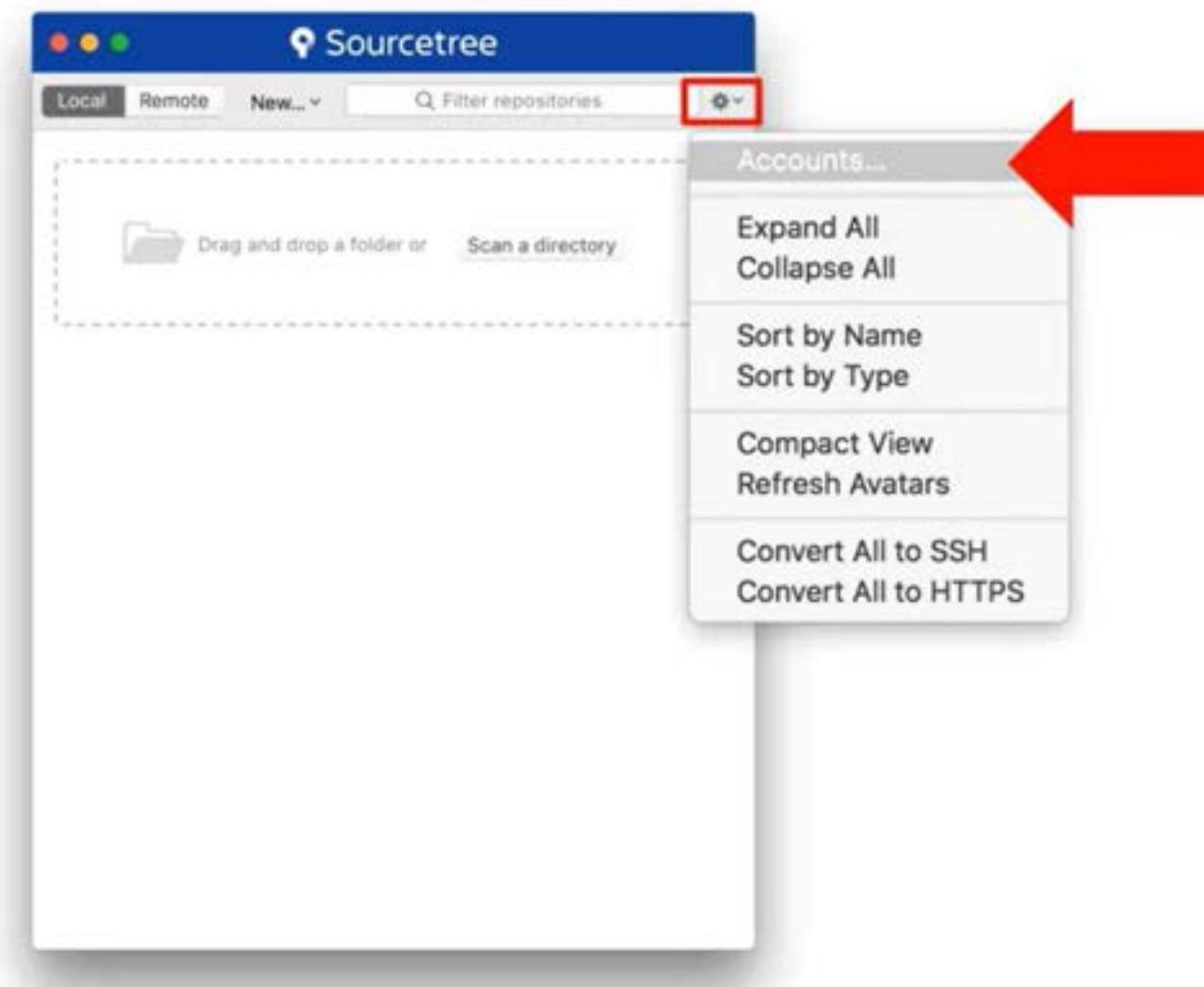
Create a folder for local
repositories

INSTALLING SOURCETREE

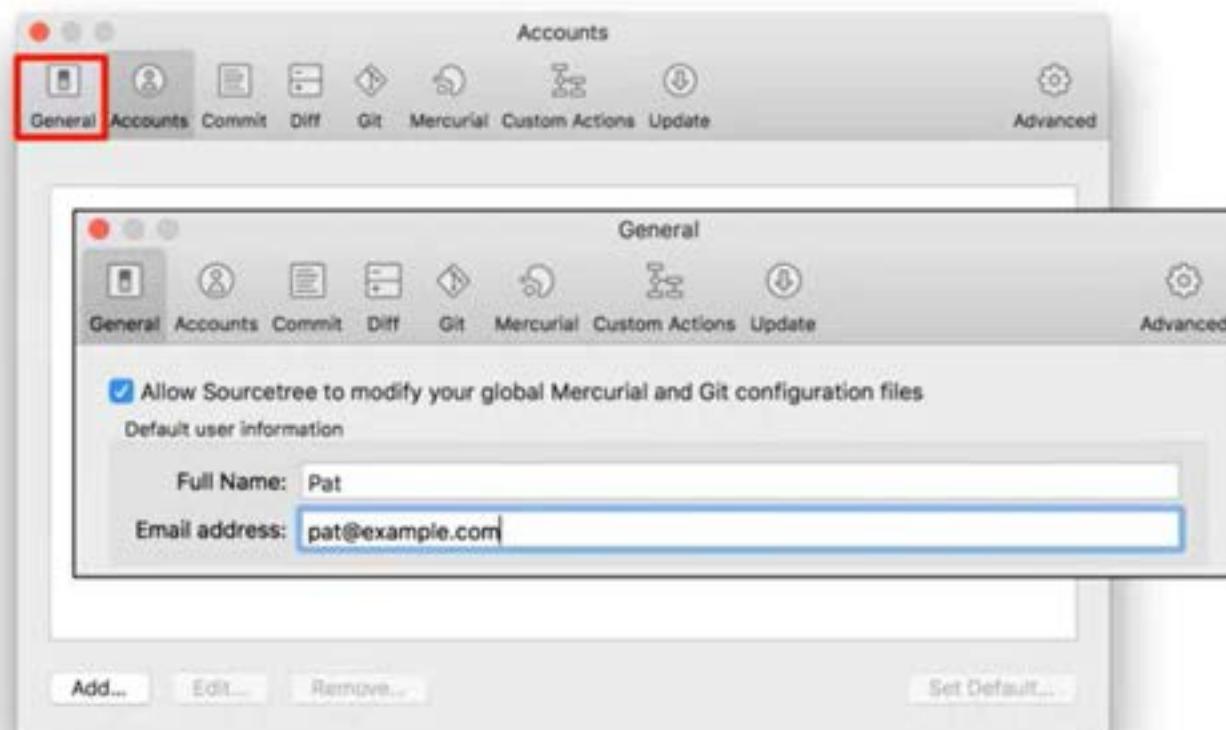
- Sourcetree can be installed on Windows or Mac OS
 - <https://www.sourcetreeapp.com/>
- Other Git clients can be found at
<https://git-scm.com/downloads/guis>



CONFIGURING SOURCETREE



CONFIGURING SOURCETREE- USER INFORMATION



Topics

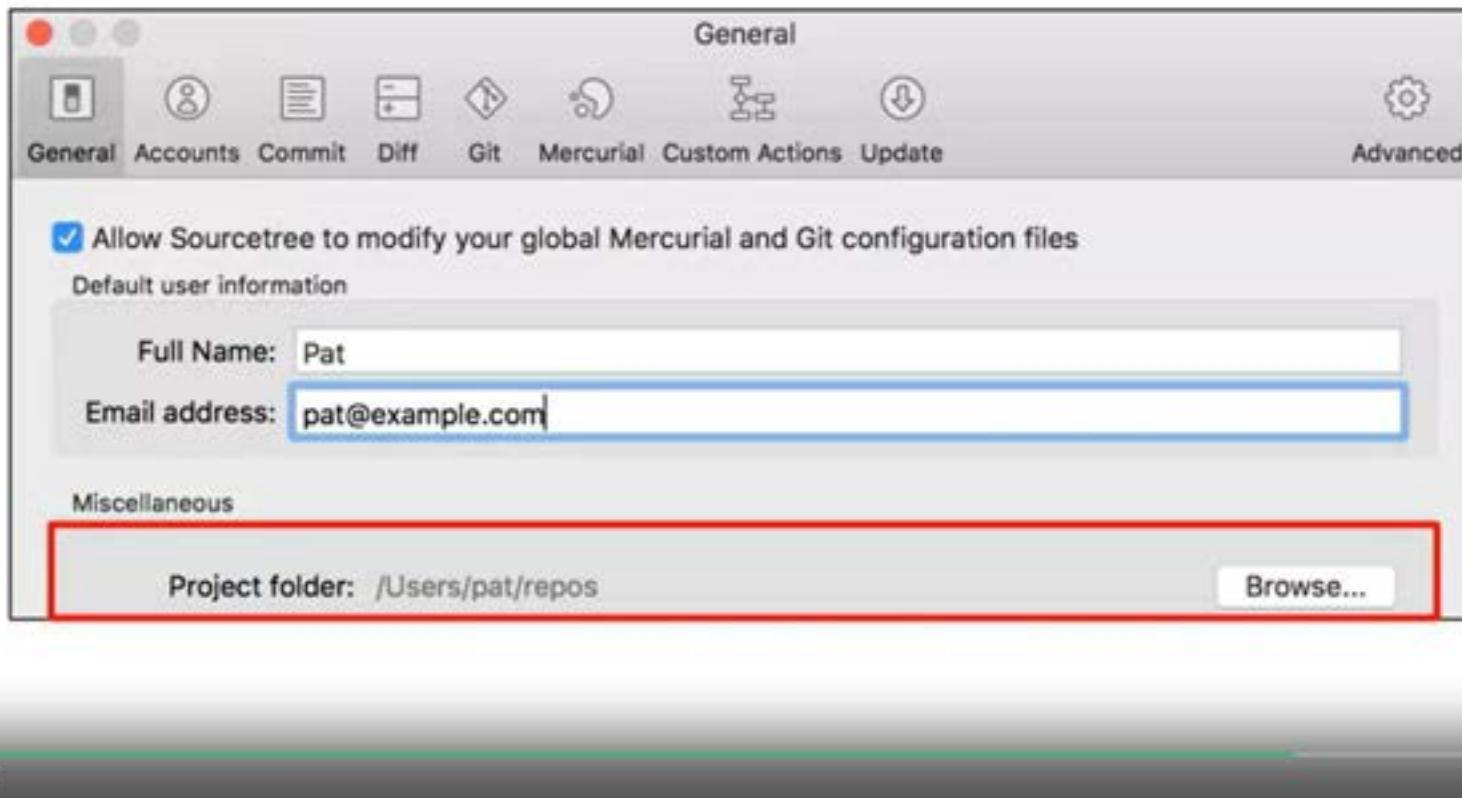
Installing Sourcetree

Configuring user information

Create a folder for local
repositories

CONFIGURING SOURCETREE- LOCAL REPOSITORY LOCATION

Suggestion: create a /repos folder in your user folder



HANDS ON

1. Install Sourcetree (if necessary)
2. Configure your name and email address
3. Create a folder for your local repositories

Installation and Getting Started

Using a command line

Copyright © 2017 Atlassian

Topics

Installing Git

Git syntax

Getting help

Configuring user information and
the default editor

BASIC GIT SYNTAX

git [command] [--flags] [arguments]



```
$ git status  
On branch master  
nothing to commit, working tree clean  
$ git status -short  
$  
$ git add file.txt
```



READING HELP

- `-f` or `--flag` Change the command's behavior
- `|` Or
- [optional]
- `<placeholder>`
- [`<optional placeholder>`]



```
git fakecommand (-p|--patch) [<id>] [--] [<paths>...]
```

<https://github.com/git/git/blob/master/Documentation/CodingGuidelines>

READING HELP

- `-f` or `--flag` Change the command's behavior
- `|` Or
- [optional]
- `<placeholder>`
- [`<optional placeholder>`]
- `()` Grouping
- `--` Disambiguates the command
- ... multiple occurrences possible



```
git fakecommand (-p|--patch) [<id>] [--] [<paths>...]
```

<https://github.com/git/git/blob/master/Documentation/CodingGuidelines>

SETTING YOUR USER NAME AND EMAIL

```
git config [--local|--global|--system] <key> [<value>]
```

- The --system flag applies to every repository for all users on your computer
- The --global flag applies to every repository that you use on your computer
- No flag or --local applies only to the current repository (highest precedence)

```
# set user name and email
$ git config --global user.name "Pat"
$ git config --global user.email "pat@example.com"
```

SETTING GIT'S DEFAULT EDITOR

Specify an editor that you like to use by
configuring `core.editor`

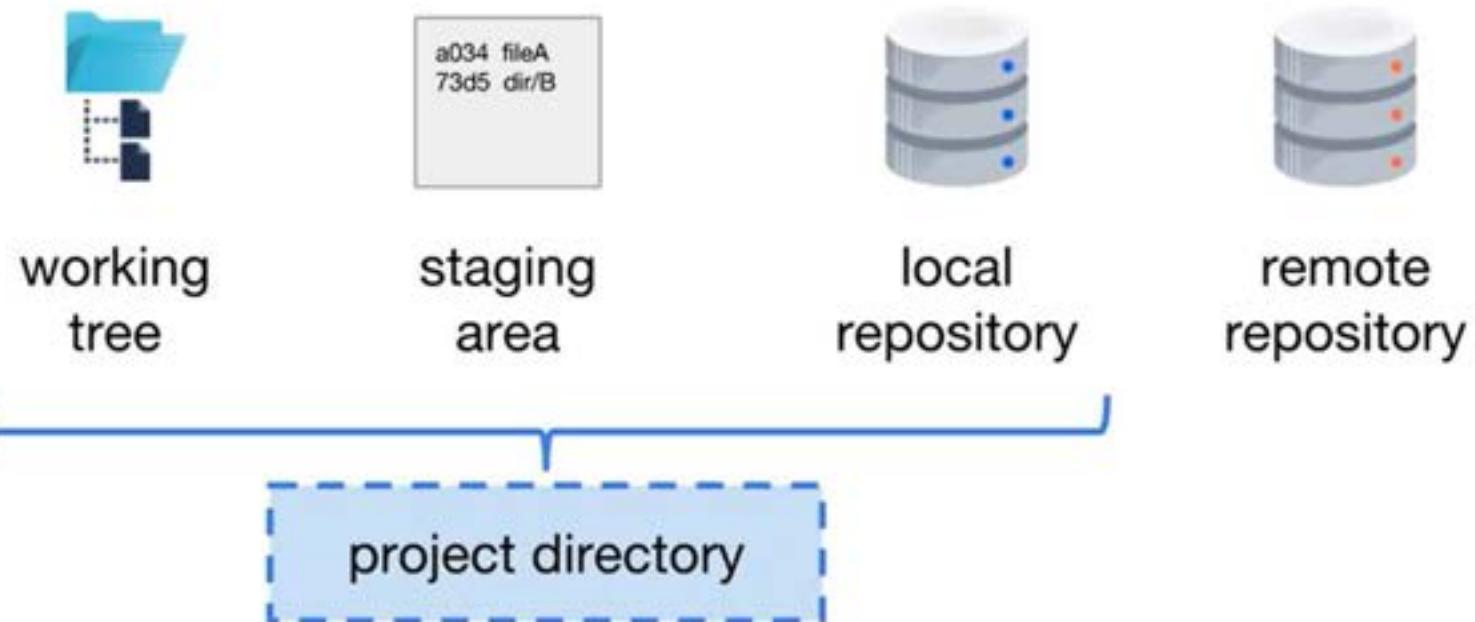
```
$ git config --global core.editor nano
```



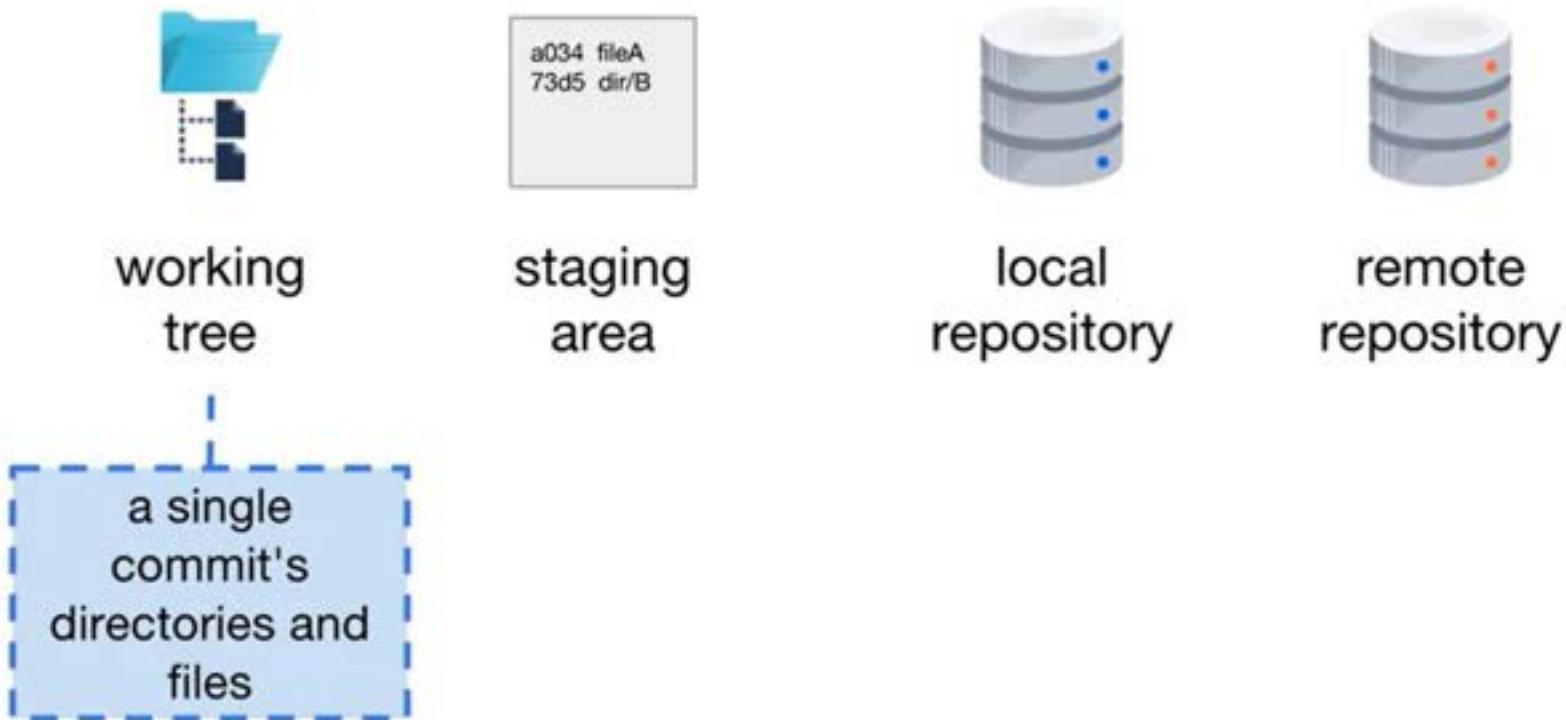
HANDS ON

1. Install the Git command line interface (if necessary)
2. Verify your Git version
3. Explore Git help
4. Configure your user name, email address and default editor

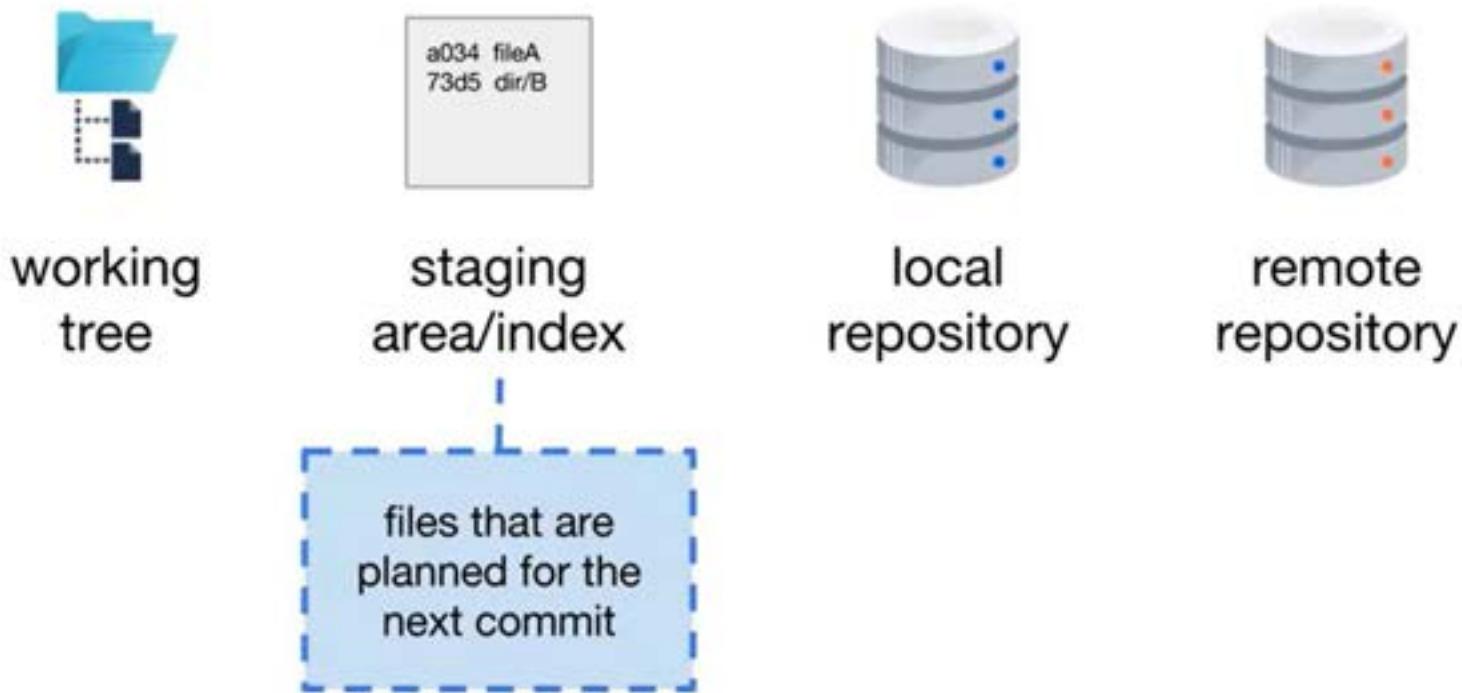
LOCATIONS OF GIT



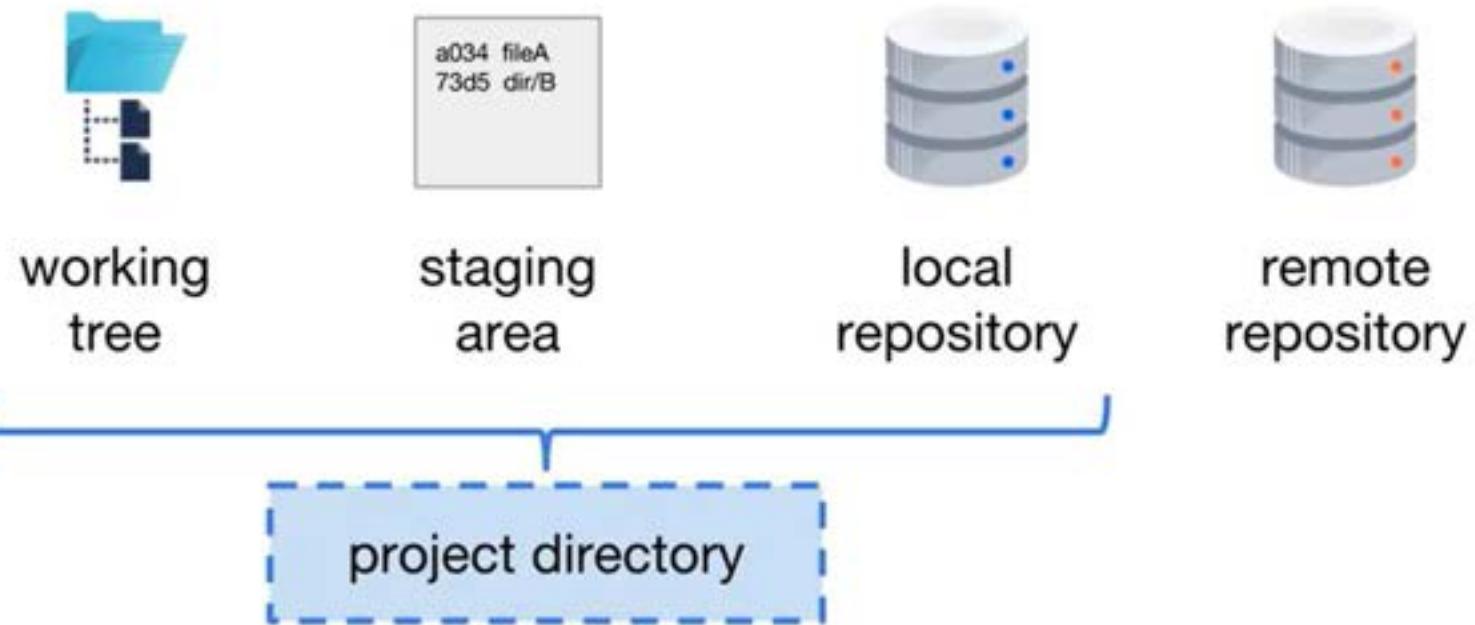
WORKING TREE



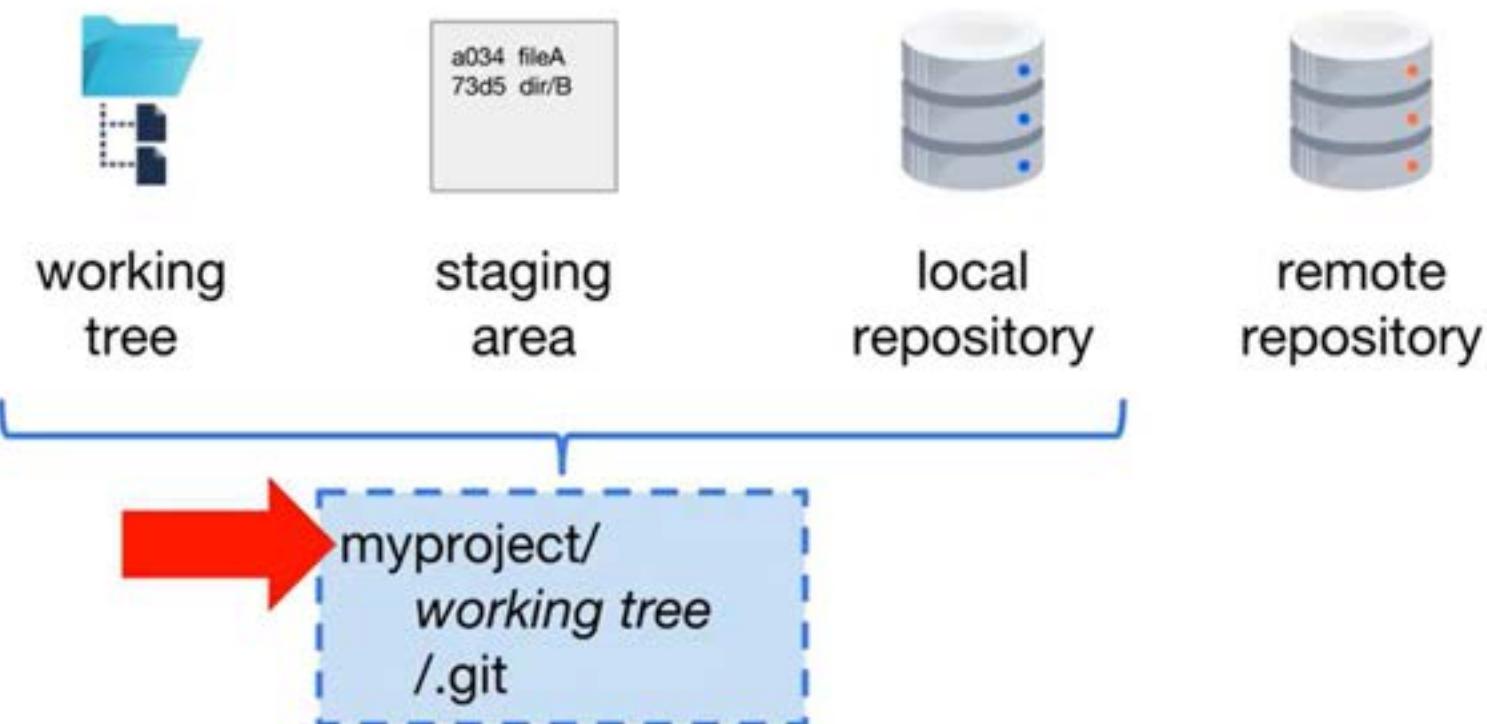
STAGING AREA/INDEX



PROJECT DIRECTORY



PROJECT DIRECTORY



REMOTE REPOSITORY



GIT LOCATIONS



working
tree



staging
area



local
repository

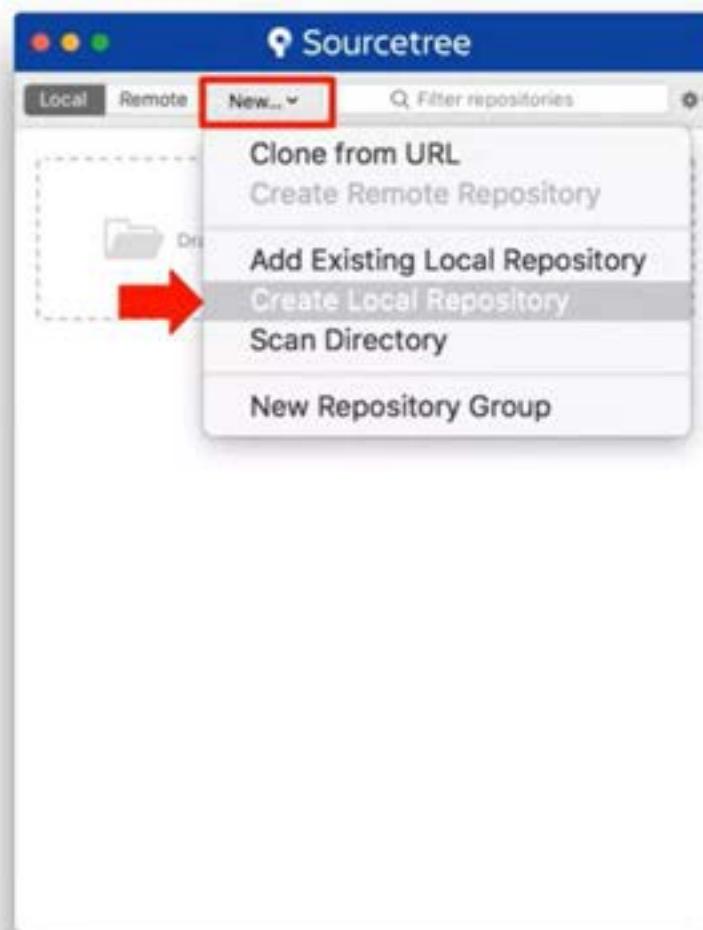


remote
repository

myproj/
working tree (your project files)
.git

CREATE A LOCAL REPOSITORY

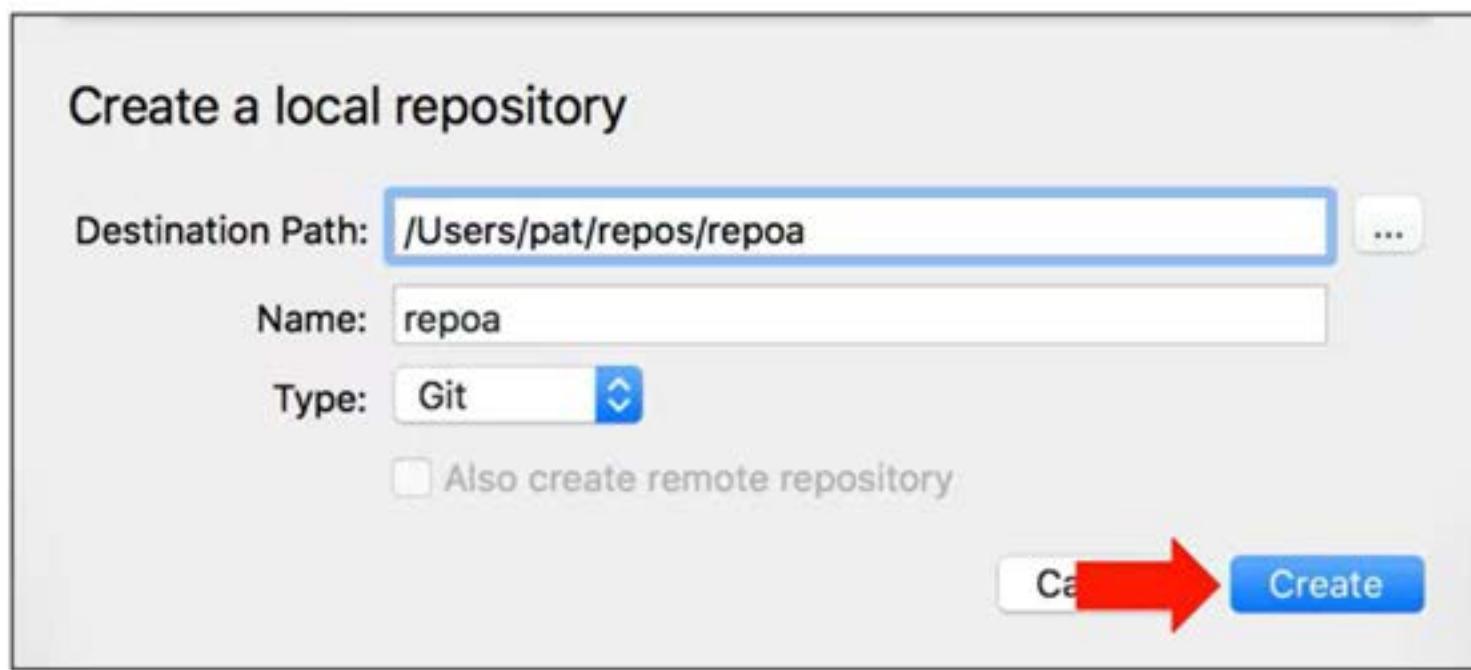
New... > Create Local Repository



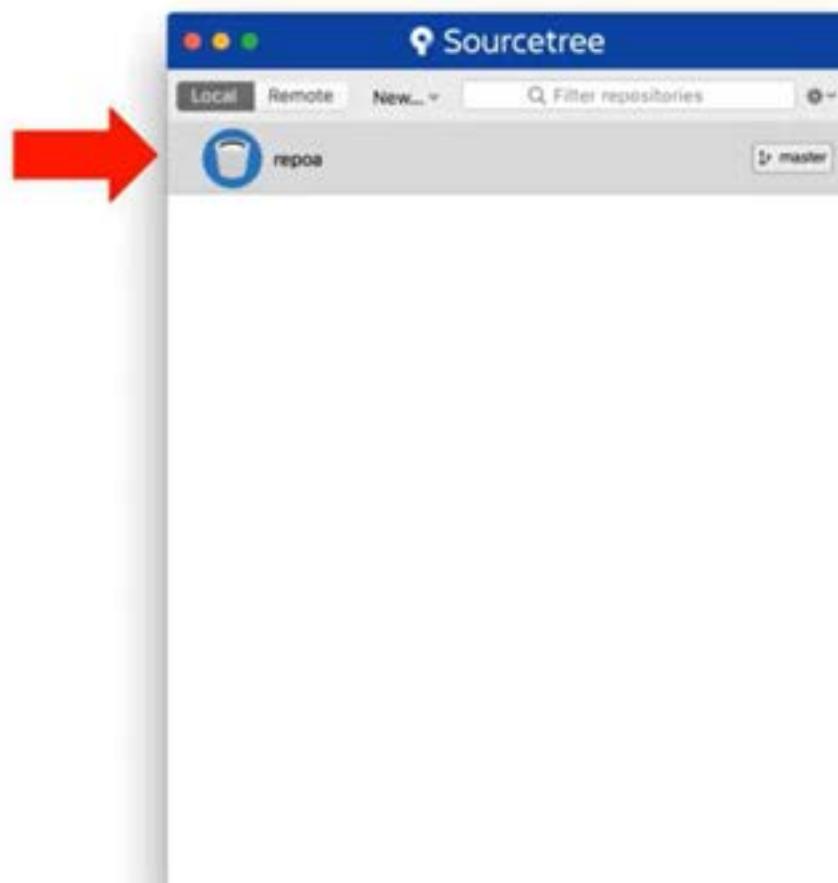
Note: This is equivalent to executing `git init` on the command line

DESTINATION PATH AND REPOSITORY NAME

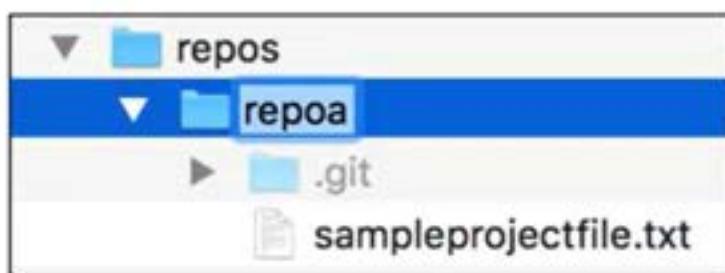
Specify the destination path (project directory) and repository name



LOCAL REPOSITORY CREATED



THE RESULT OF CREATING A LOCAL REPOSITORY



working
tree



staging
area



local
repository

project directory

repoa/
/.git
working tree (your project files)

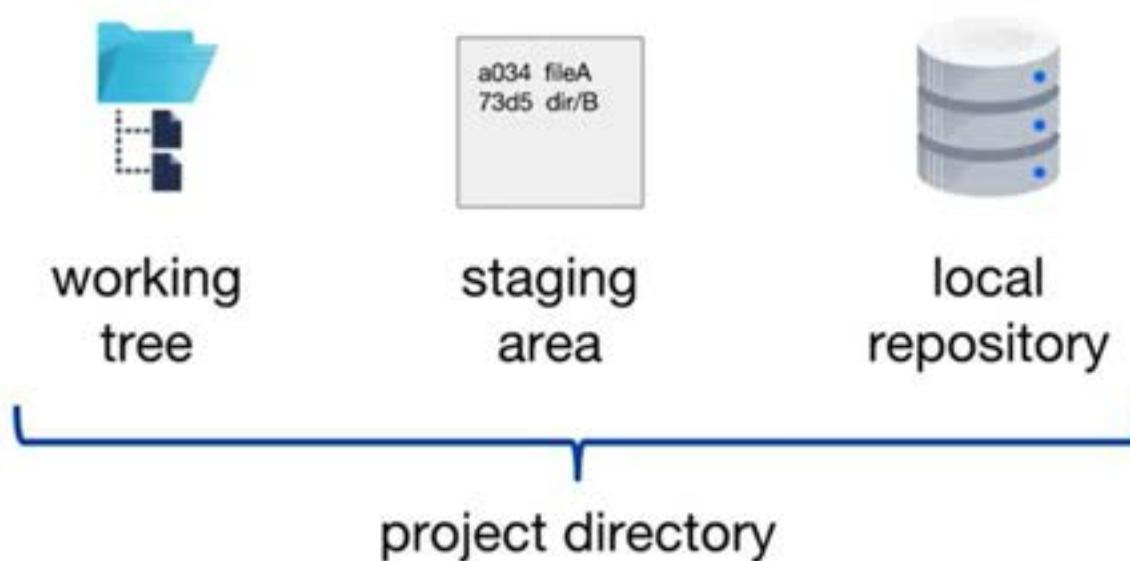
CREATING THE FIRST THREE GIT LOCATIONS

git init - initialize (create) a repository

```
~$ mkdir repos
~$ cd repos
repos$ mkdir myproj
repos$ cd myproj
myproj $ git init
Initialized empty Git repository in myproj/.git/
myproj $ ls -a
.  ..  .git
```

THE RESULT OF `git init`

```
myproj $ git init
Initialized empty Git repository in myproj/.git/
```



Topics

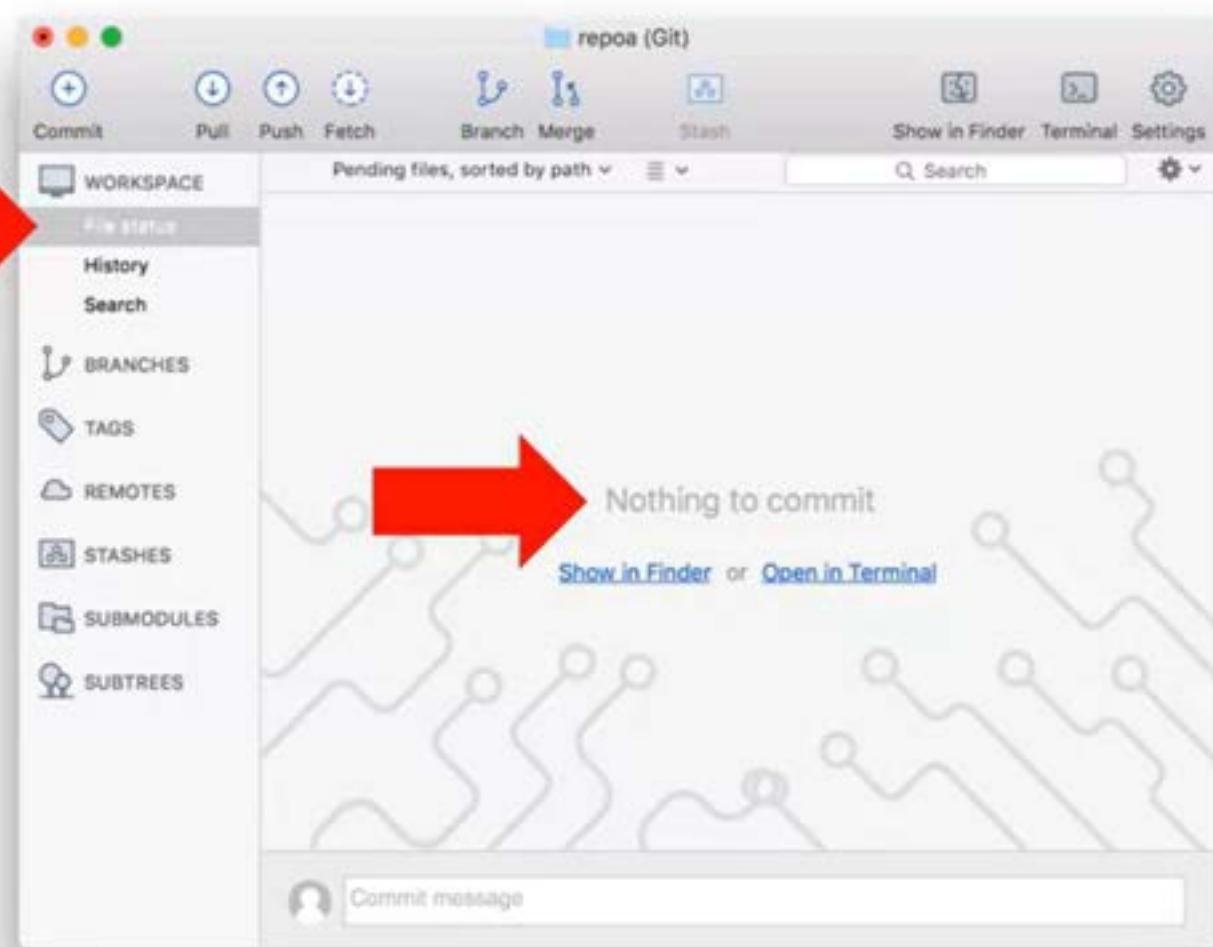
[View file status](#)

[Stage content](#)

[Commit content to the repository](#)

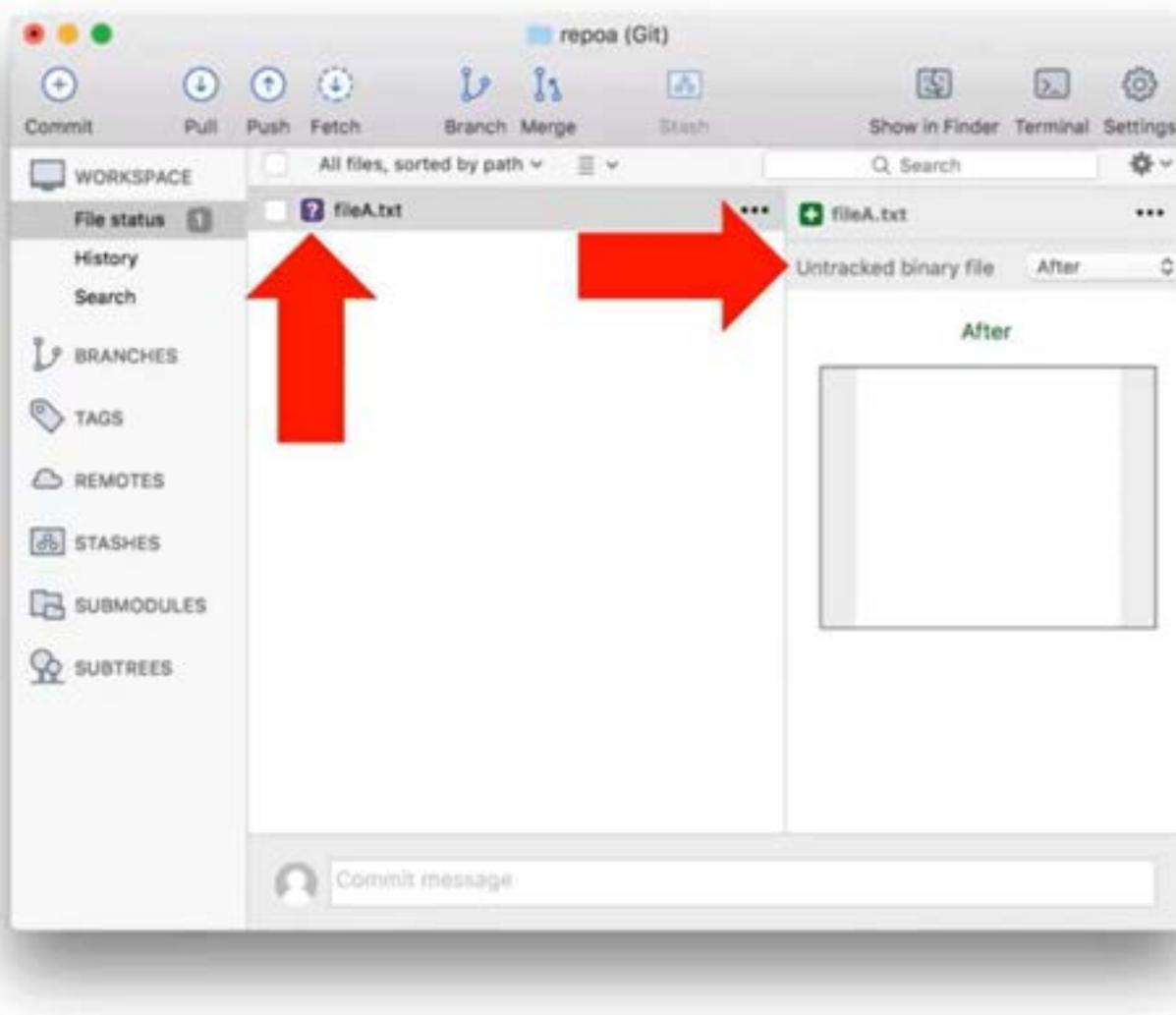
[View the commit history](#)

FILE STATUS



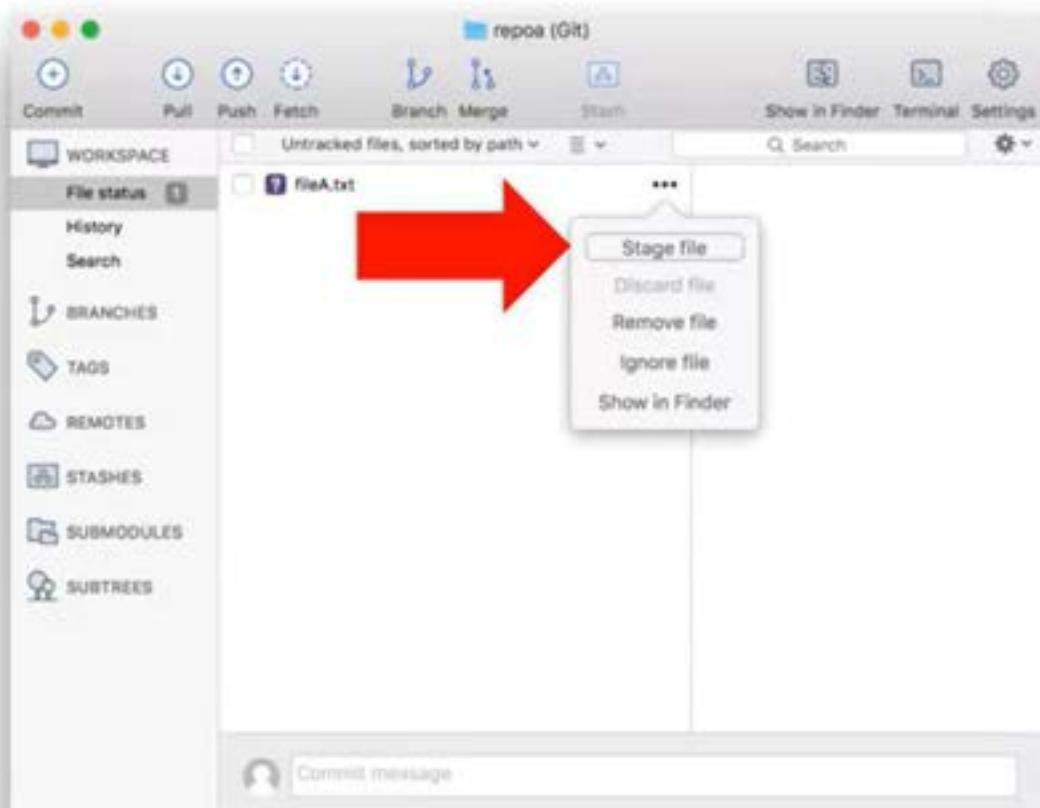
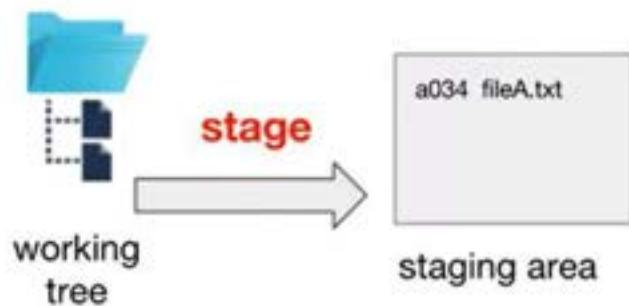
git status is used to obtain this information at the command line

UNTRACKED



STAGING A FILE

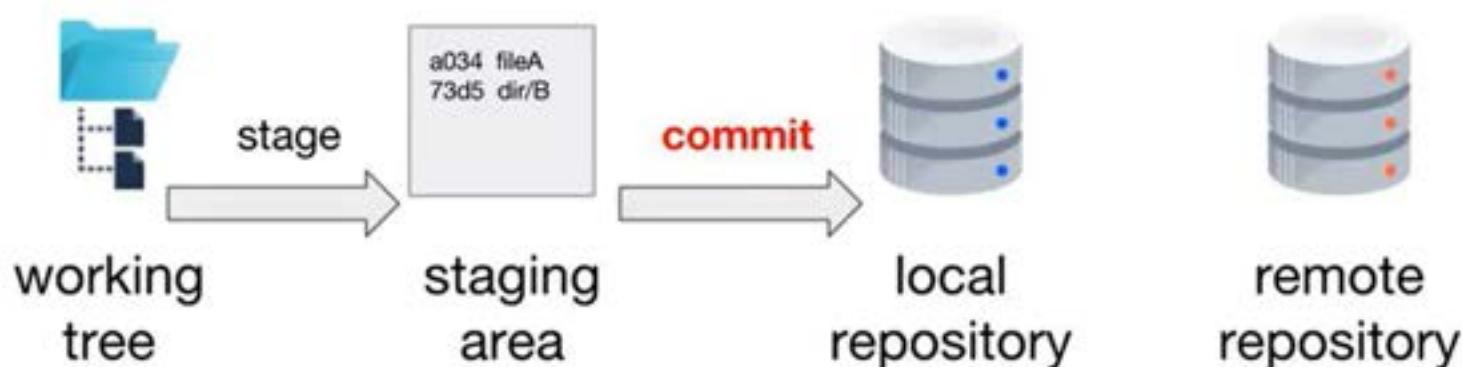
Staged content is part of the next commit



git add is used to stage a file at the command line

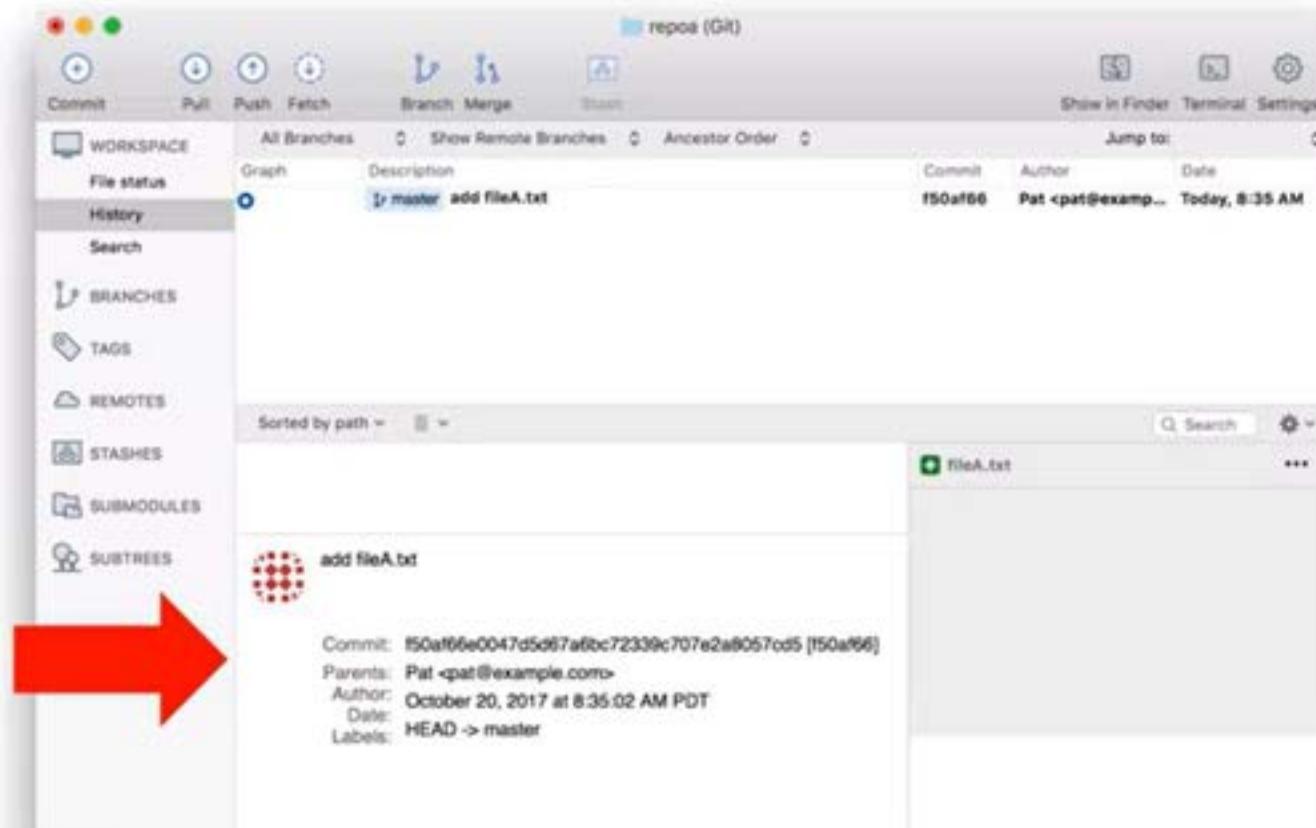
COMMITTING TO THE LOCAL REPOSITORY

A commit adds content to the **local** repository



HISTORY

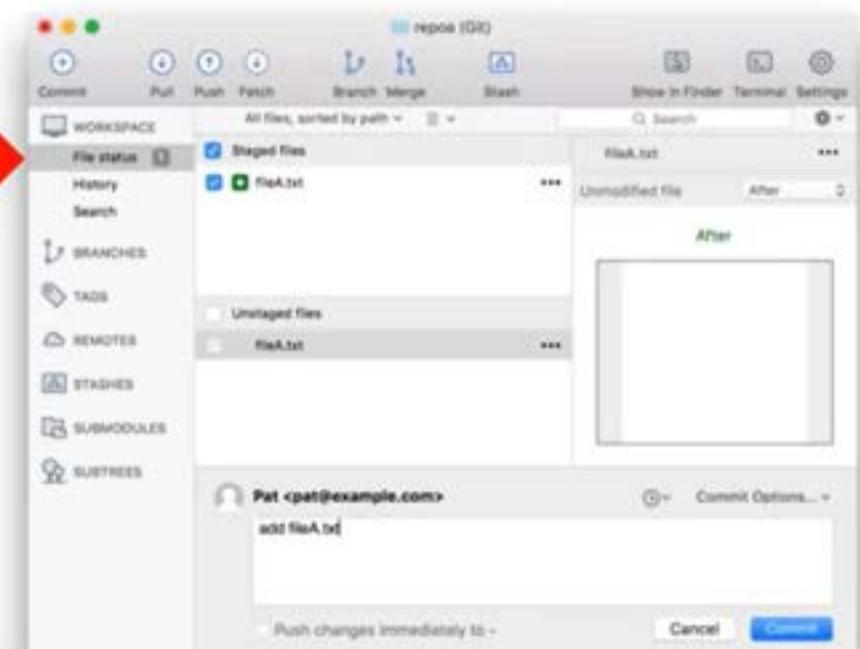
The **History** tab shows the commit history



git log shows the commit history at the command line

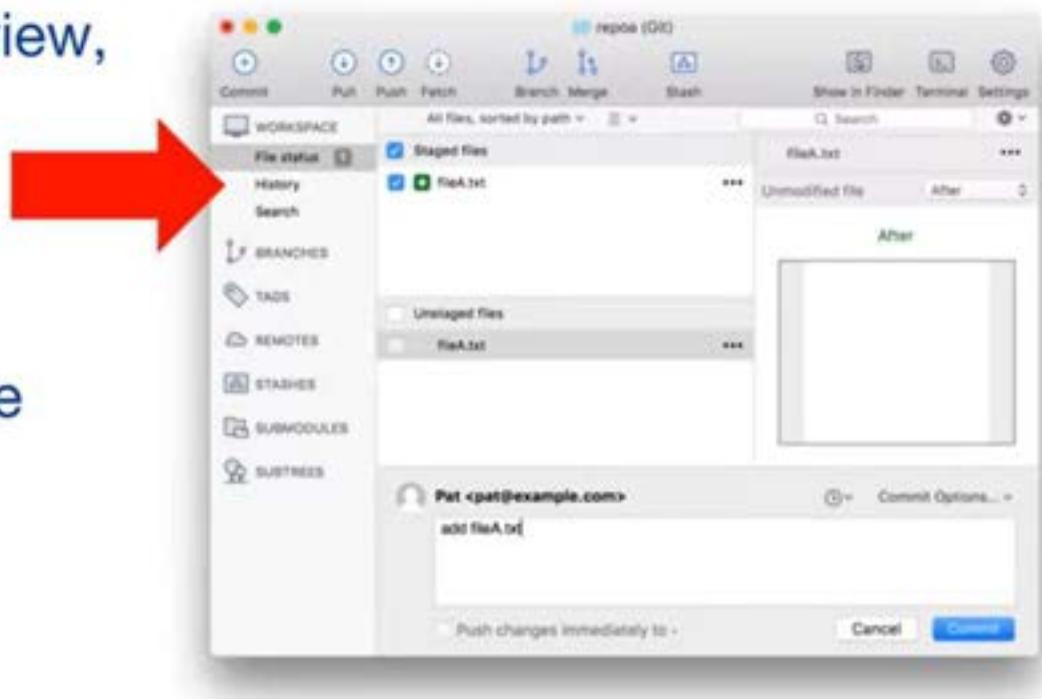
REVIEW

- The File status tab is used to view, stage and commit files or directories



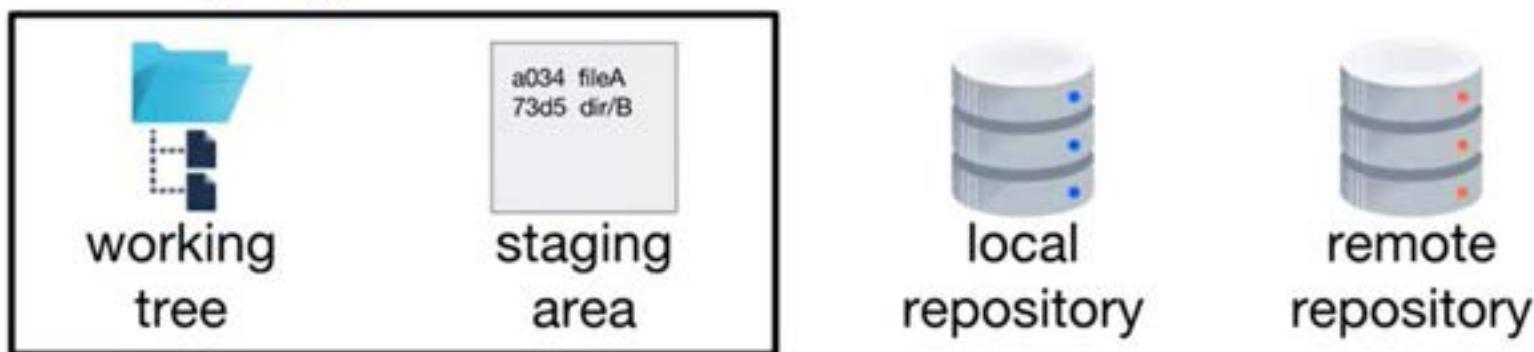
REVIEW

- The File status tab is used to view, stage and commit files or directories
- Specify a meaningful commit message
- Use the **History** tab to view the project's commits



git status

Use `git status` to view the status of files in the working tree and staging area



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

git status WITH AN UNTRACKED FILE

```
myproj$ touch fileA.txt
myproj$ git status
On branch master
No commits yet
Untracked files:
  (use "git add <file>..." to include in what will be
committed)
    fileA.txt
nothing added to commit but untracked files present (use "git
add" to track)
```



EXISTING CONTENT SHOWS AS UNTRACKED

```
myproj$ git status
On branch master
No commits yet
Untracked files:
  (use "git add <file>..." to include in what will be
committed)
    fileA.txt
nothing added to commit but untracked files present (use "git
add" to track)
```

git add

git add <file-or-directory>



git add

git add <file-or-directory>



```
myproj$ git add fileA.txt
On branch master
No commits yet
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

    new file:   fileA.txt
```

git add- DIRECTORIES

Add directories with git add <directory>

```
(create dirA and dirA/fileA.txt, dirA/fileB.txt)
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    dirA/
nothing added to commit but untracked files present (use "git add" to
track)
$ git add dirA
$ git status
On branch master
No commits yet
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   dirA/fileA.txt
    new file:   dirA/fileB.txt
```

git add .

Add all untracked or modified files using git add .

```
$ touch fileA.txt
$ touch fileB.txt
$ git add .
$ git status
On branch master
No commits yet
Changes to be committed:
(use "git rm --cached <file>..." to unstage)
  new file:   fileA.txt
  new file:   fileB.txt
```

MODIFIED FILE

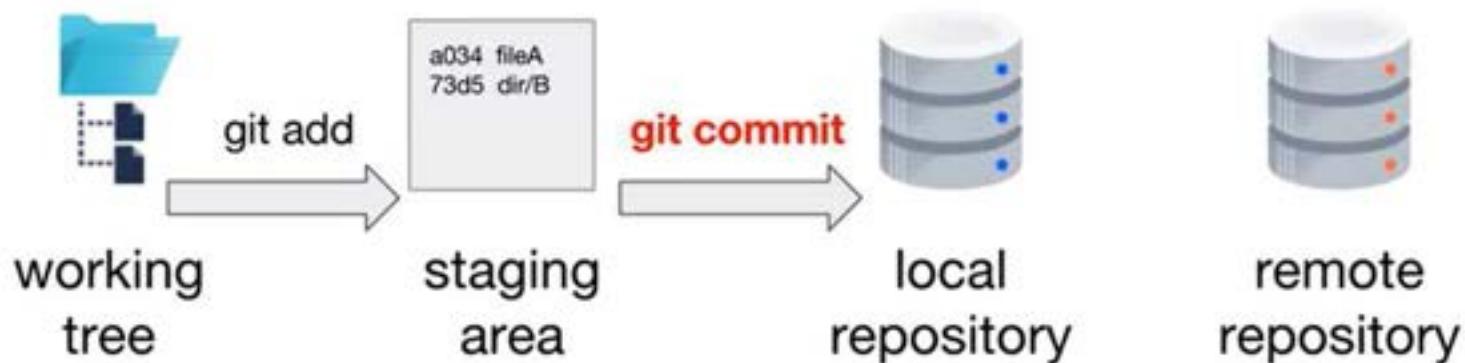
Has been added to the stage and/or committed but then changed in the working tree

```
$ touch fileA.txt # create fileA.txt
$ git status -s # -s means short status
?? fileA.txt # ?? means untracked
$ git add fileA.txt
$ git status -s
A fileA.txt # A means added (staged)
$ echo "feature 1" > fileA.txt # modify fileA.txt
$ git status -s
AM fileA.txt # AM means added and modified
$ git add fileA.txt
$ git status -s
A fileA.txt
```

git commit

Adds staged content to the local repository as a commit

- Previously committed files are also included
- Creates a snapshot of the entire project



git commit

```
$ git commit -m "initial commit"
[master (root-commit) 725c95a] initial commit
 1 file changed, 1 insertion(+)
  create mode 100644 fileA
$ git status
On branch master
nothing to commit, working tree clean
```

VIEWING THE COMMIT HISTORY WITH `git log`

```
$ git log
commit 725c95ae156e5f10d4f99735b2fa8698a9860128 (HEAD -> master)
Author: Pat Smith <pat@example.com>
Date:   Sat Sep 9 14:15:04 2017 -0700

    initial commit
```

<https://git-scm.com/docs/git-log>

LIMITING THE SIZE OF git log

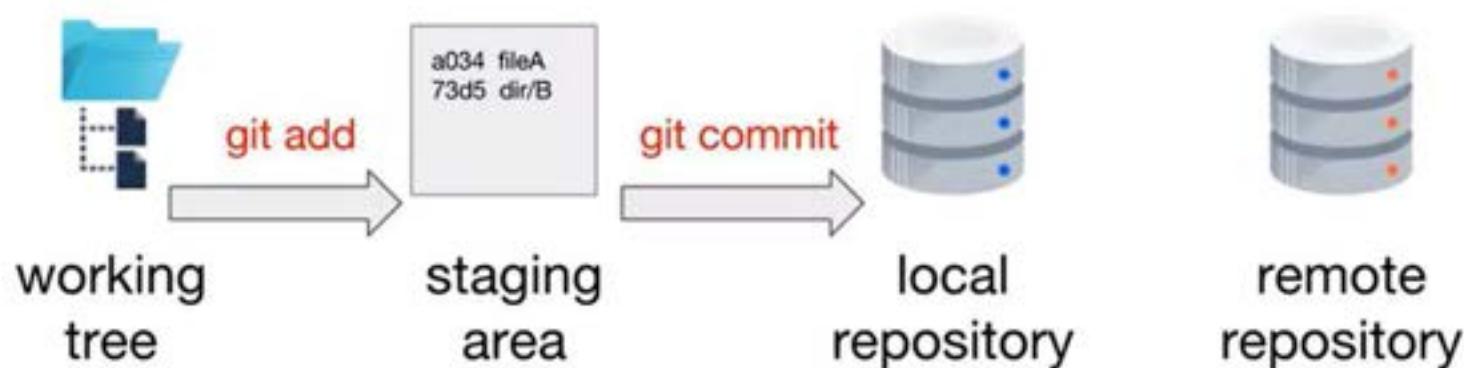
- `--oneline` condensed version of the log
- `-#` limit the log to the most recent # commits

```
$ git log --oneline
lef16ac (HEAD -> master) changed fileA.txt to version 2
e8d41c0 updated fileA.txt feature 1
14b97e6 added fileA.txt

$ git log --oneline -2
lef16ac (HEAD -> master) changed fileA.txt to version 2
e8d41c0 updated fileA.txt feature 1
```

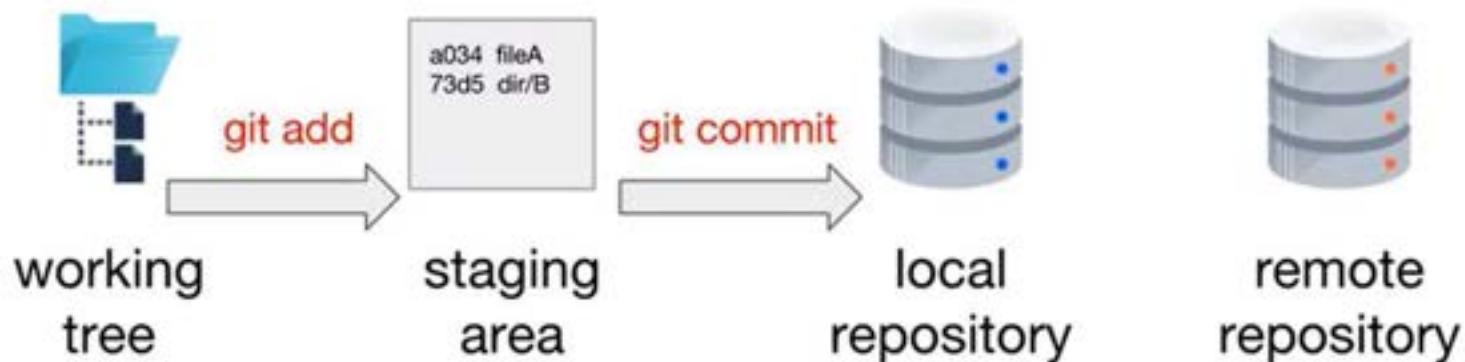
REVIEW

- `git status` - view the status of files in the working tree and staging area



REVIEW

- `git status` - view the status of files in the working tree and staging area
- `git add` - adds untracked or modified files to the staging area
- `git commit` - creates a snapshot of the current project
- `git log` - view the commit history



Create a Remote Repository

Copyright © 2017 Atlassian

REMOTE REPOSITORY

- Professionally managed
- Source of truth
- Integrates with other systems



working
tree



staging
area



local
repository



remote
repository



REMOTE GIT REPOSITORY OPTIONS

- Hosted options
 - Bitbucket
 - GitHub
- On-premise options
 - Bitbucket Server
 - GitHub Enterprise
 - open source software

REMOTE REPOSITORY

- A remote repository is often a "bare" repository
- By convention, remote repository names end with ".git"



<https://bitbucket.org/<team-name>/repoa.git>

behind the scenes, `git init --bare` is called to create the remote repository

REMOTE REPOSITORY

- A remote repository is often a "bare" repository
- By convention, remote repository names end with ".git"

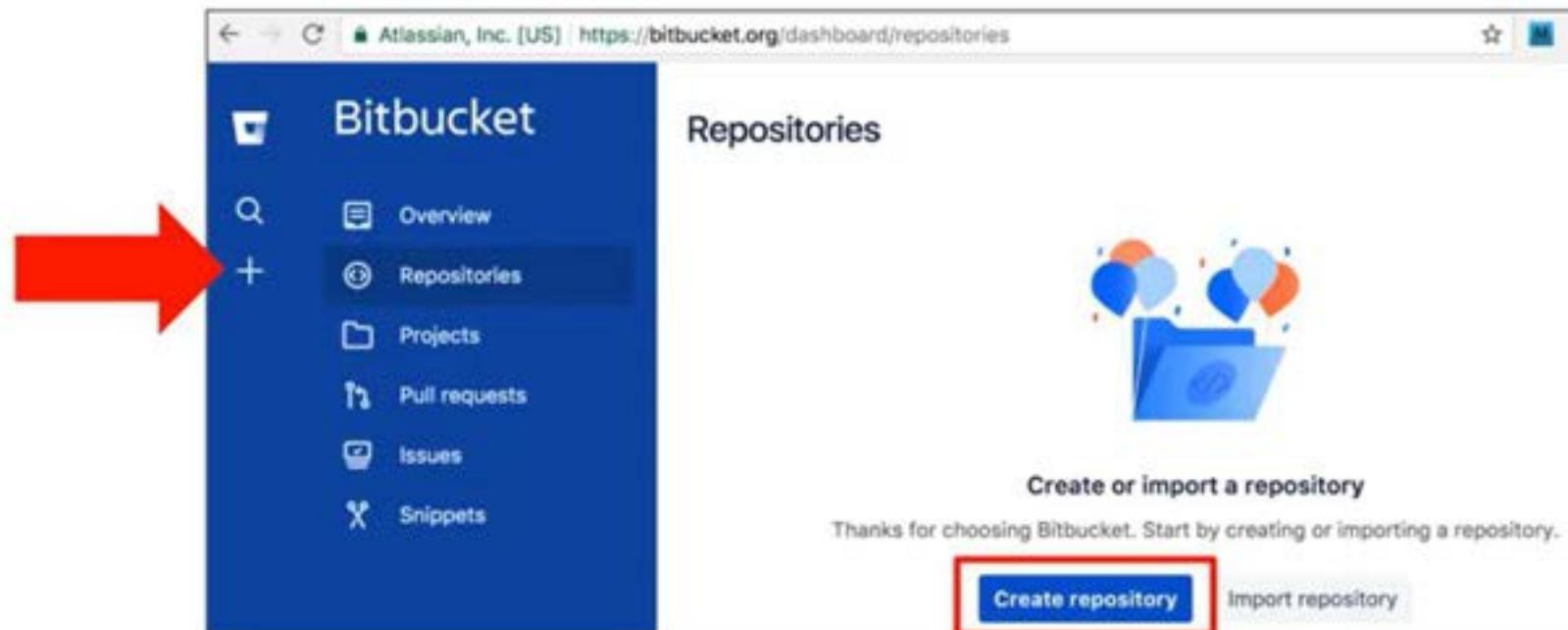


remote
repository

<https://bitbucket.org/<team-name>/repoa.git>

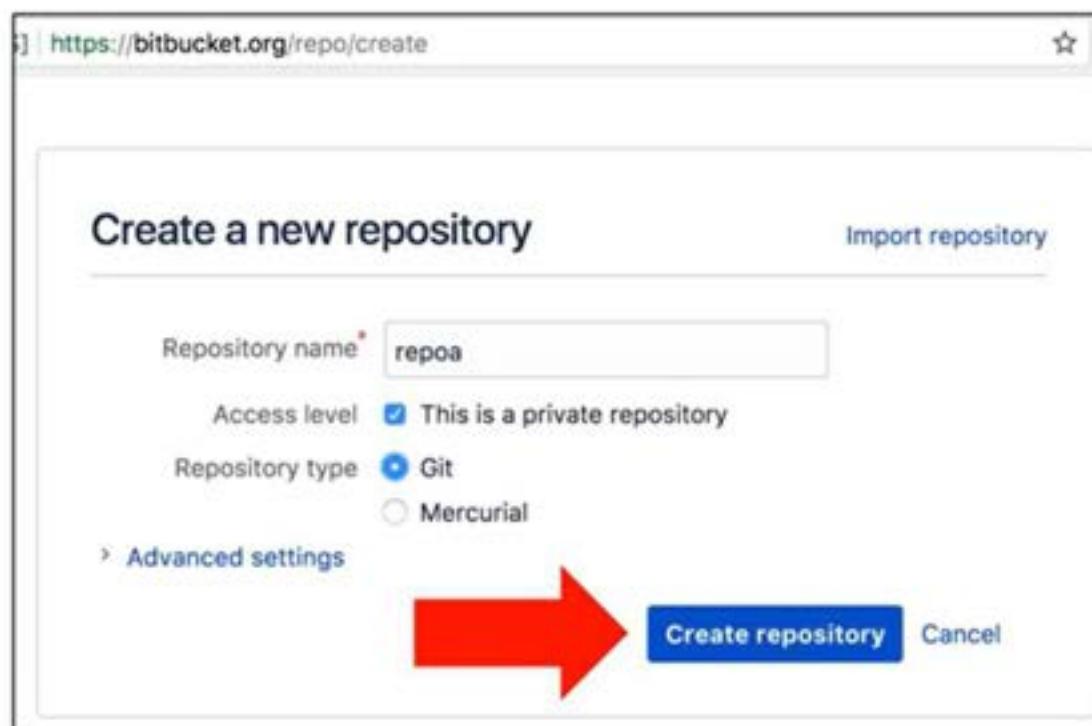
behind the scenes, `git init --bare` is called to create the remote repository

CREATE A REMOTE REPOSITORY



NAME THE REPOSITORY

The Git hosting provider will append ".git" to the remote URL



REPOSITORY CREATED

The screenshot shows the Bitbucket interface. On the left, a sidebar menu has the following items:

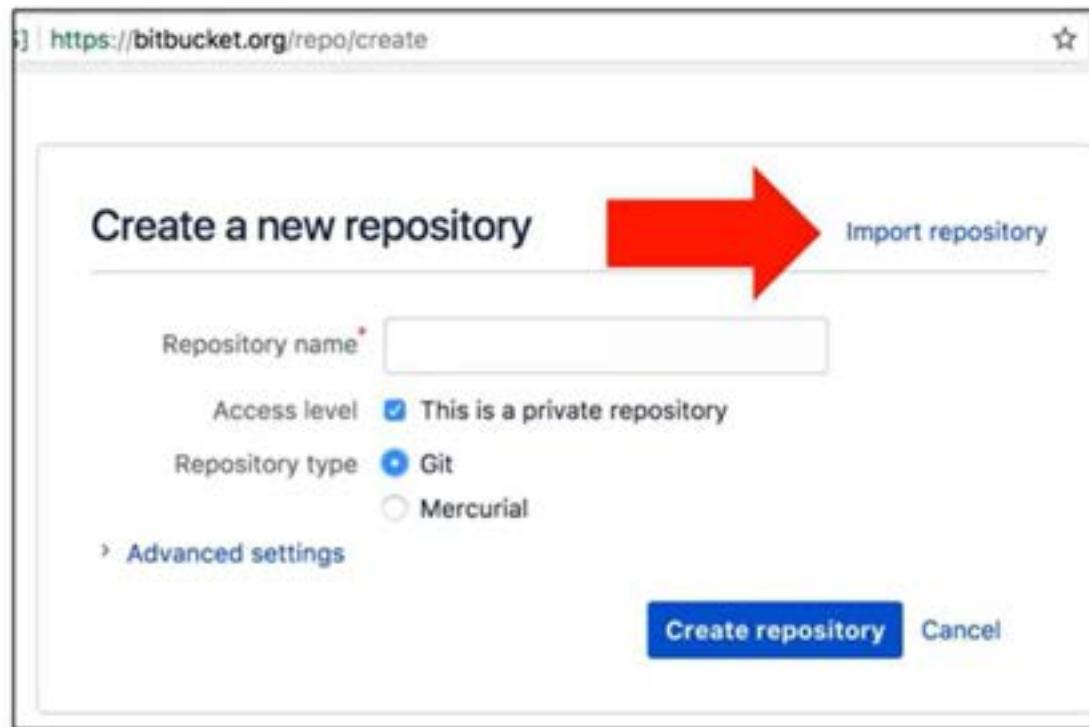
- Overview
- Repositories** (selected)
- Projects
- Pull requests
- Issues
- Snippets

The main area is titled "Repositories" and shows a list with one item:

Repository
<code></> repoa</code>

A large red arrow points upwards towards the repository name in the list.

IMPORT AN EXISTING REMOTE REPOSITORY



IMPORT AN EXISTING REPOSITORY

Specify the old repository type and URL

Import existing code Create new repository

Old repository

Source: **Git** ▼

URL*: []

CodePlex
Git
Mercurial

New repository

Repository name*:

Access level: This is a private repository

Repository type: Git
 Mercurial

› Advanced settings

Import repository [Cancel](#)

REVIEW

- A remote repository is a bare repository
- Often serves as the project's source of truth
- Hosting providers make creating remote repositories easy



Topics

Remotes- clone vs. add

Clone a Bitbucket repository

Clone any remote repository

Add a remote repository to a local repository

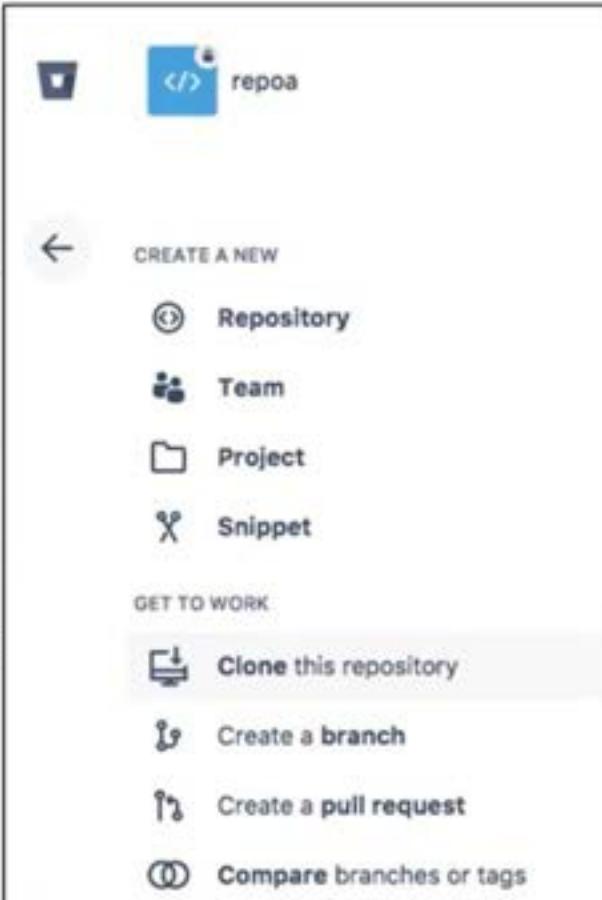
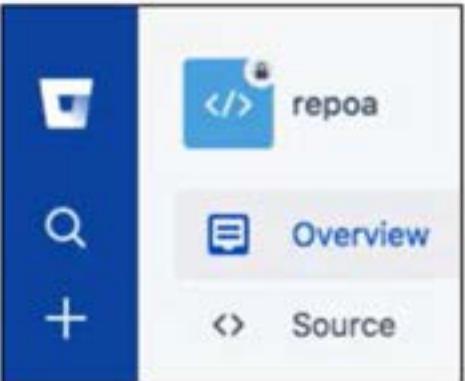
Push commits to a remote repository

WHAT IS A CLONE?

A *clone* is a local copy of a remote repository



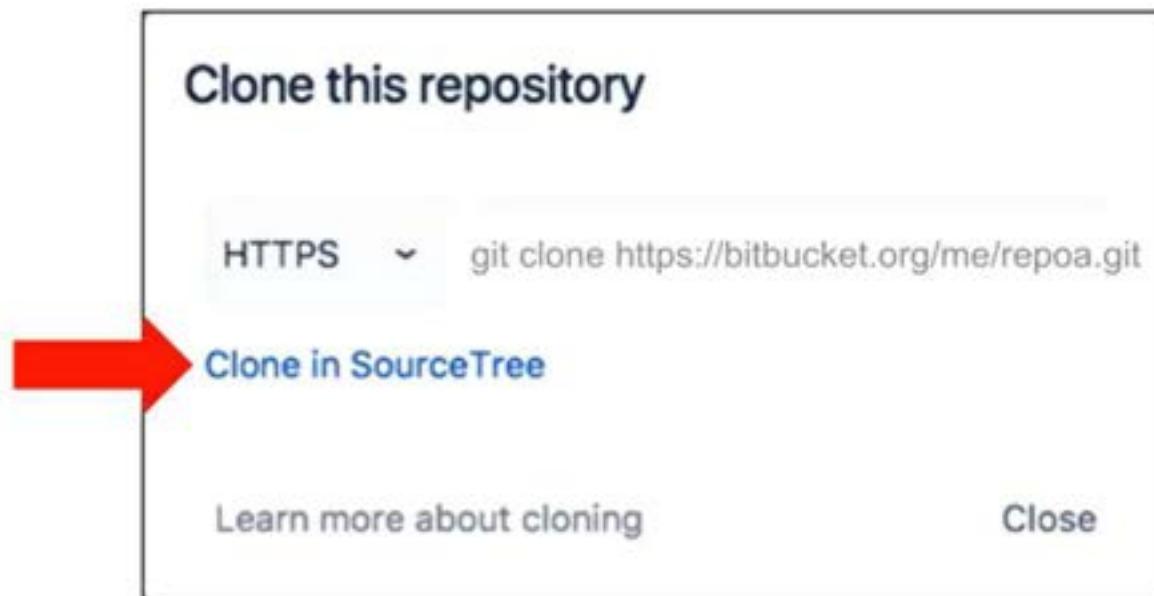
CLONING TO SOURCETREE FROM BITBUCKET



The image displays two screenshots side-by-side. On the left, the SourceTree application interface is shown, featuring a sidebar with icons for Home, Search, and Add, and a main panel displaying a repository named 'repoa'. The 'Overview' tab is selected. On the right, a screenshot of a Bitbucket repository page for 'repoa' is shown. It includes a back arrow, a 'CREATE A NEW' section with 'Repository', 'Team', 'Project', and 'Snippet' options, and a 'GET TO WORK' section with 'Clone this repository', 'Create a branch', 'Create a pull request', and 'Compare branches or tags' links. The 'Clone this repository' link is highlighted.

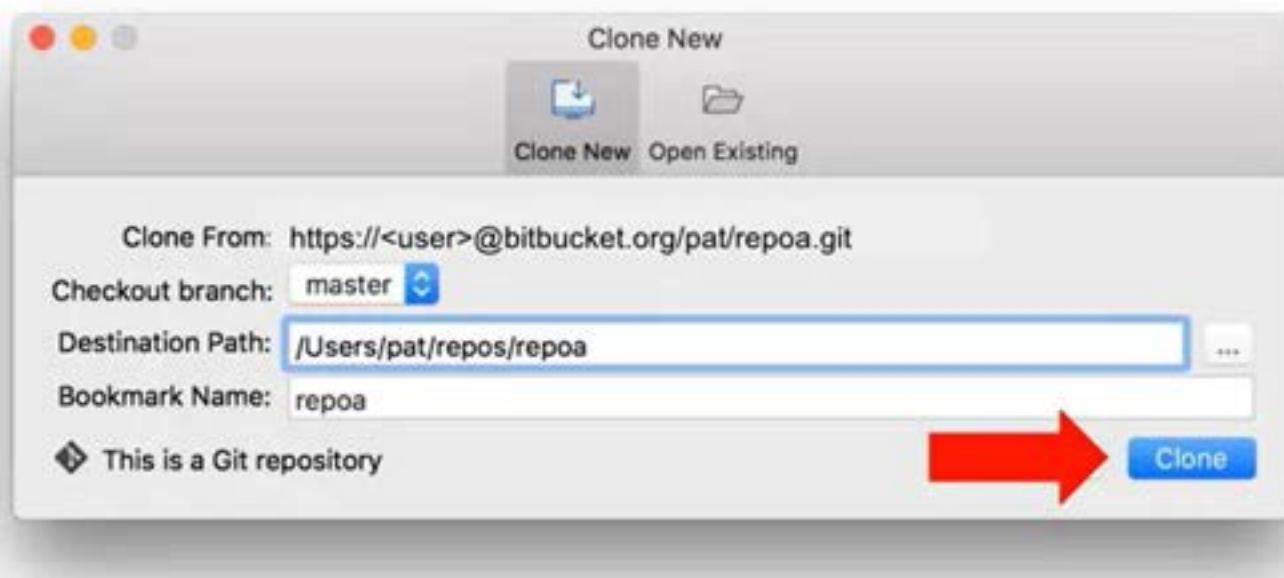
git clone can be used to clone a remote repository using a command line

CLONE IN SOURCETREE



CLONE INFORMATION

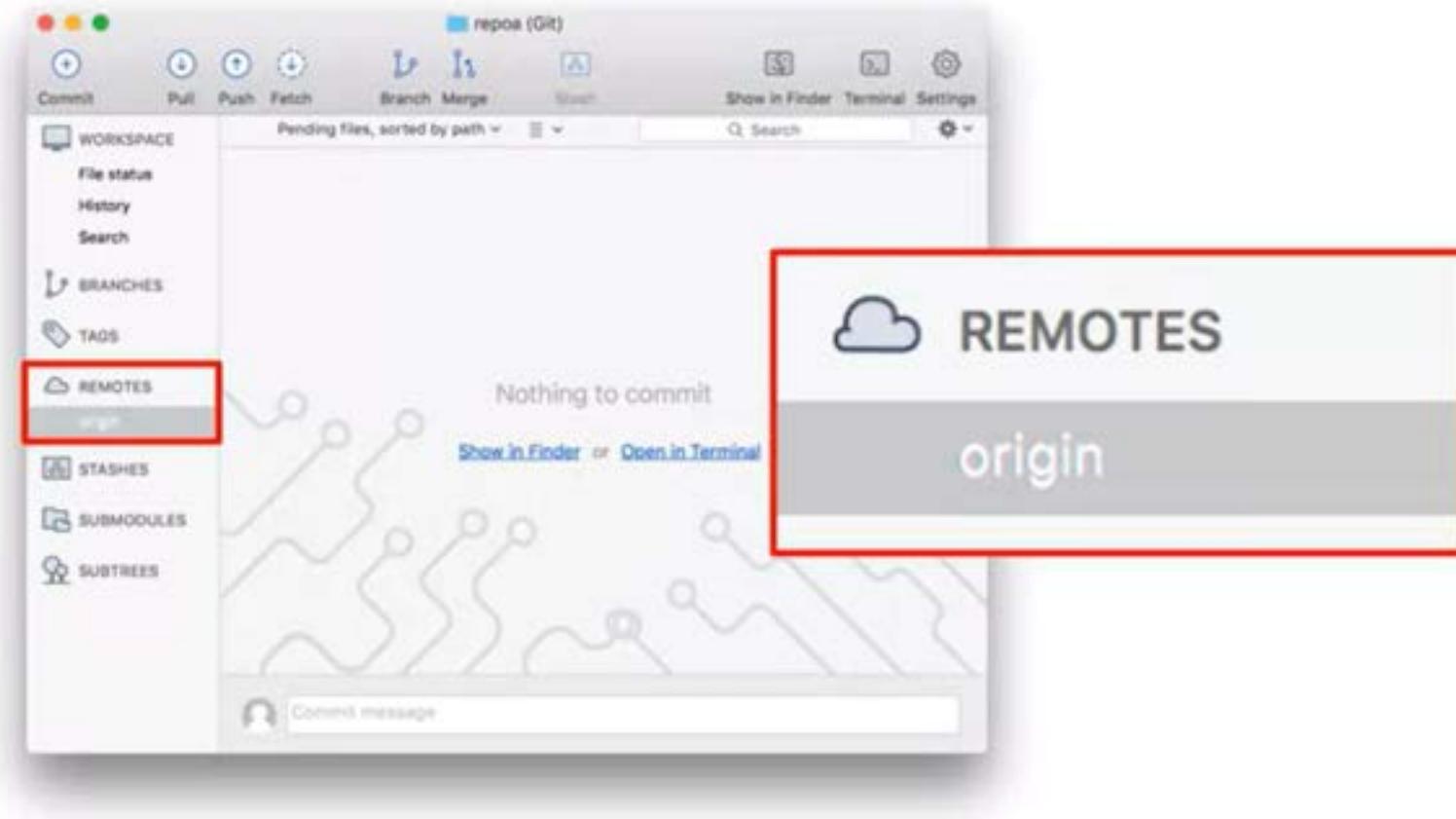
Default information is filled in for you



LOCAL REPOSITORY CREATED

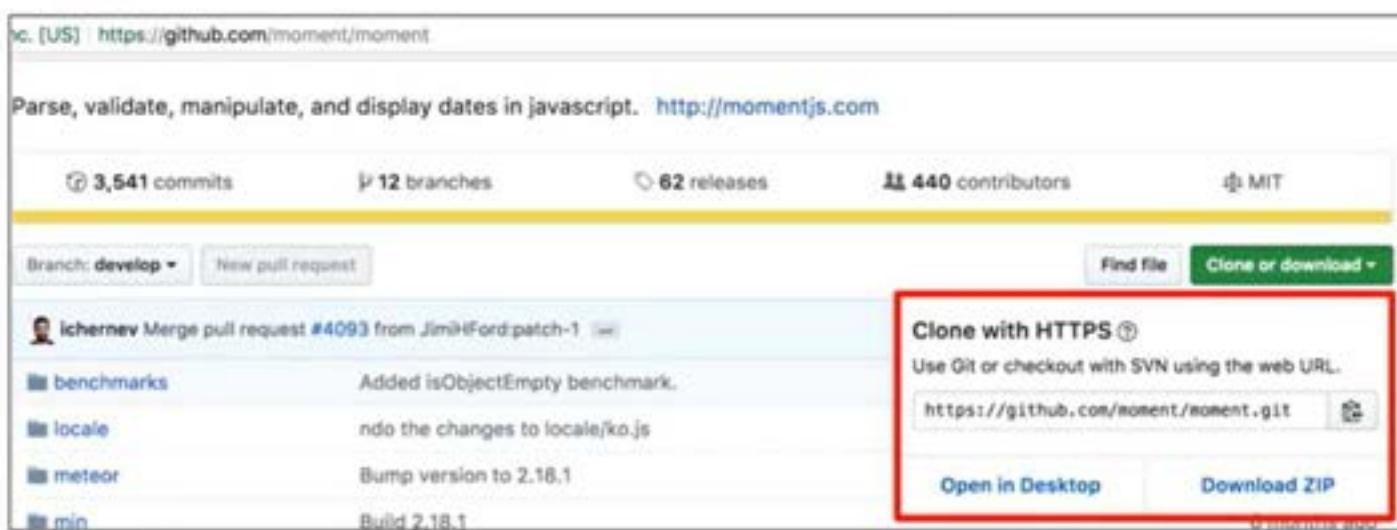
The remote information is associated with the local repository

- *origin* is the default alias for the remote repository URL



CLONE IN SOURCETREE FROM URL

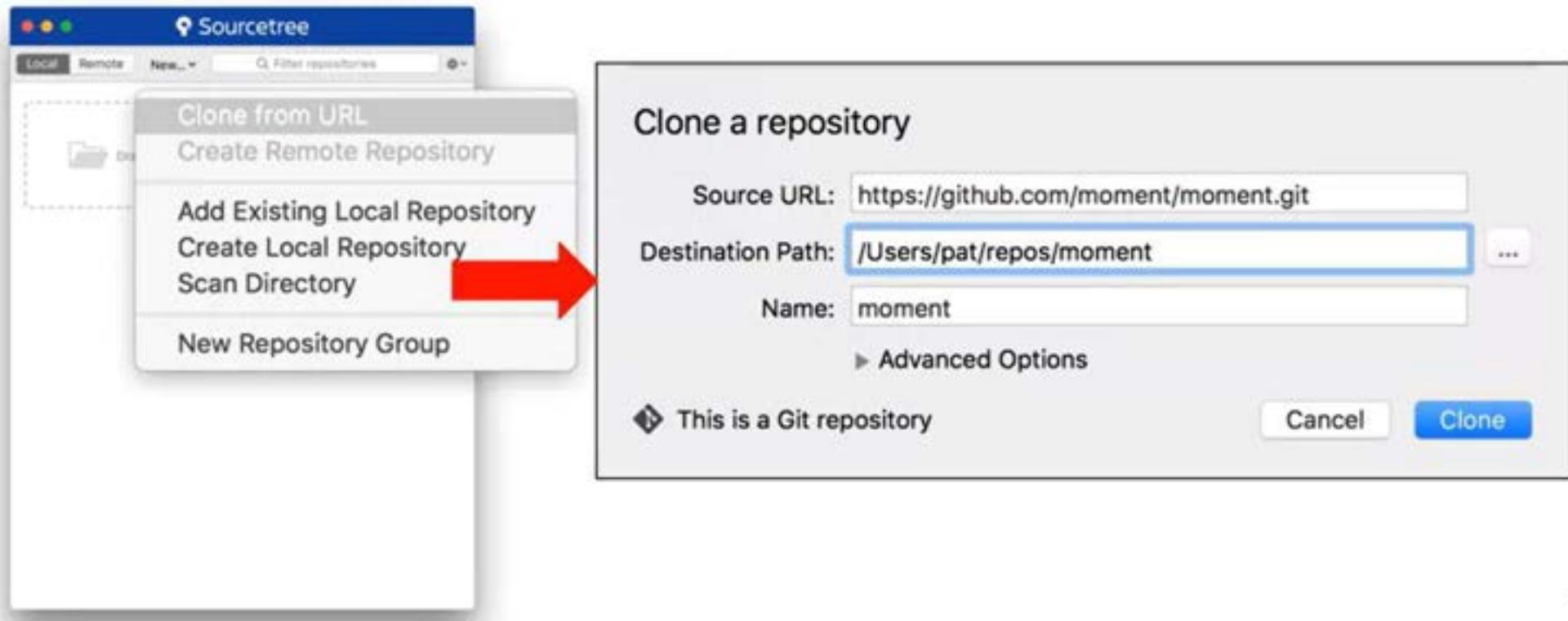
- You can clone any repository into Sourcetree if you have the URL
- Copy the URL from Bitbucket or from any other source



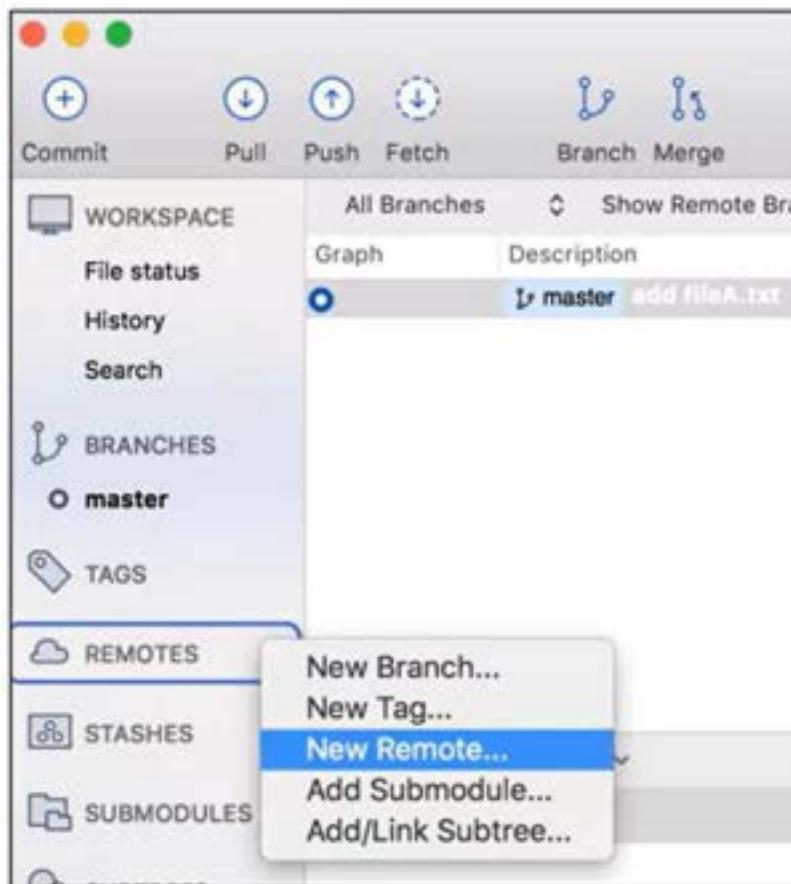
CLONE FROM URL

Open Sourcetree and paste the copied **Source URL**

- **Destination Path** and **Name** are automatically filled in

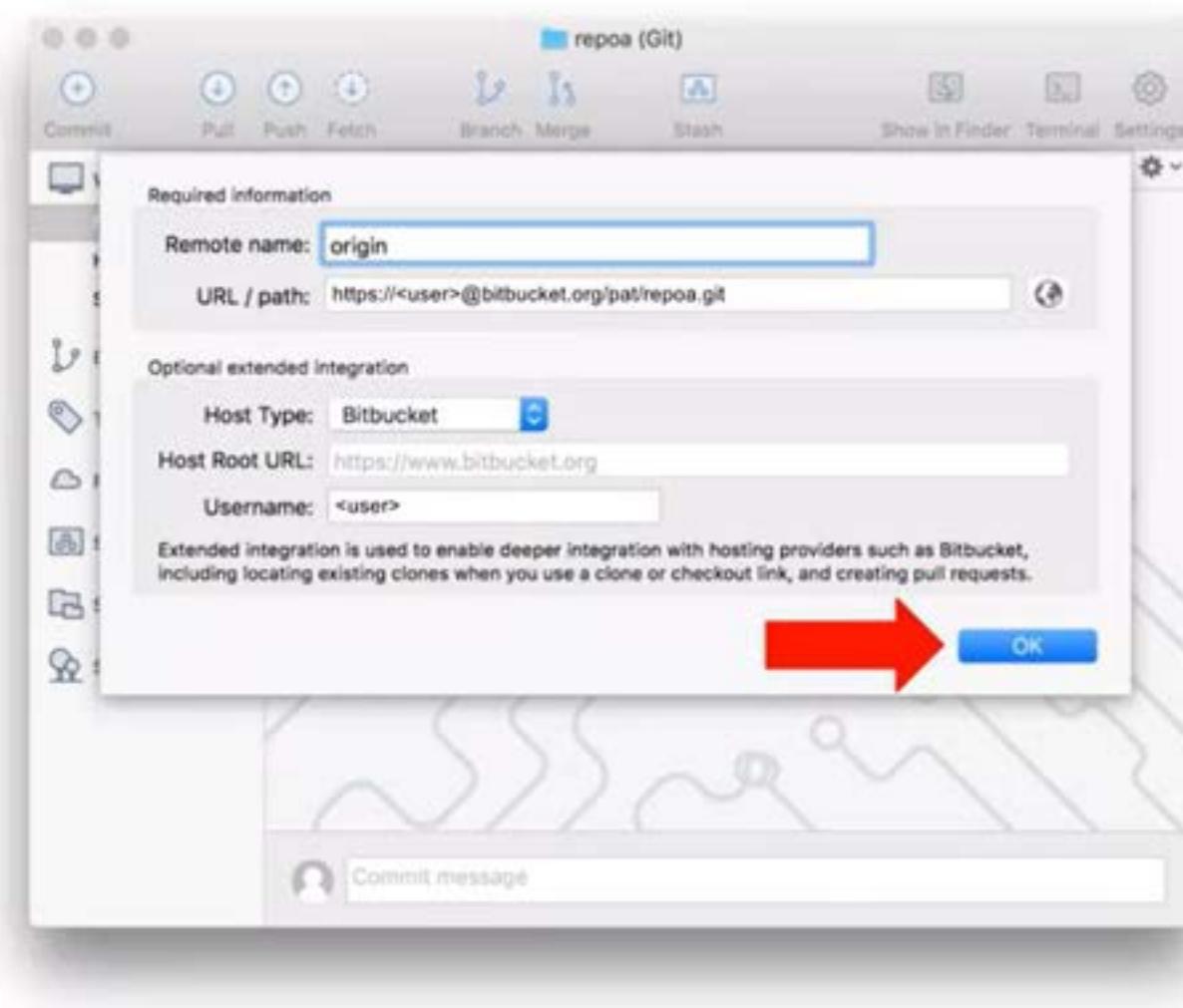


ADD A REMOTE REPOSITORY



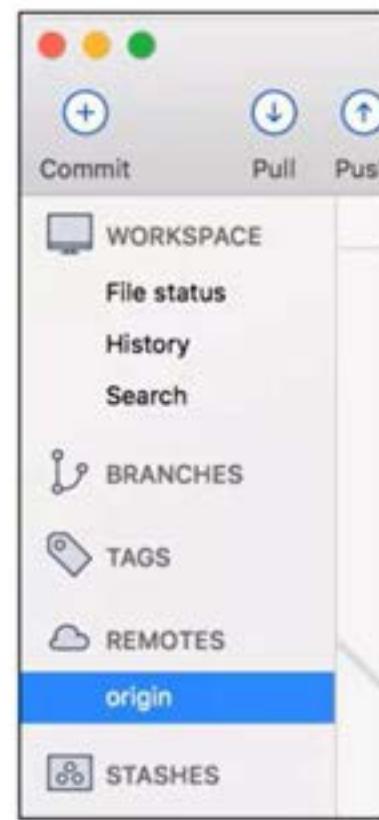
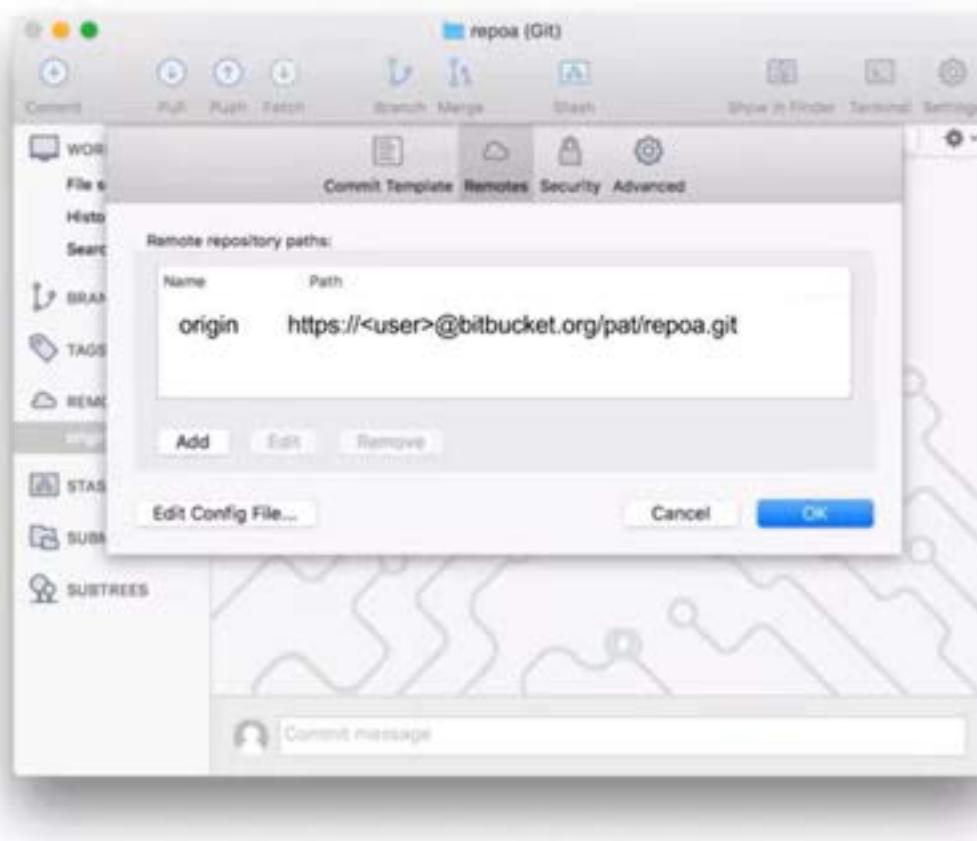
git remote add can be used to add a remote repository using a command line

SPECIFY REMOTE INFORMATION



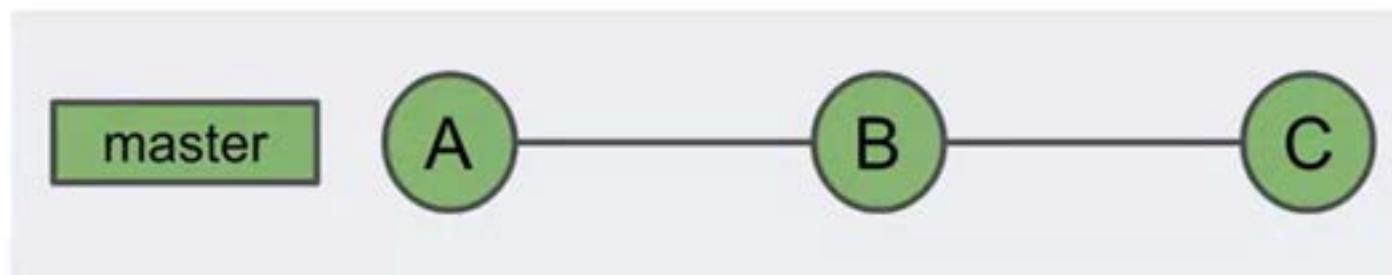
FINISH ADDING THE REMOTE

A remote repository named "origin" should now be associated with the local repository



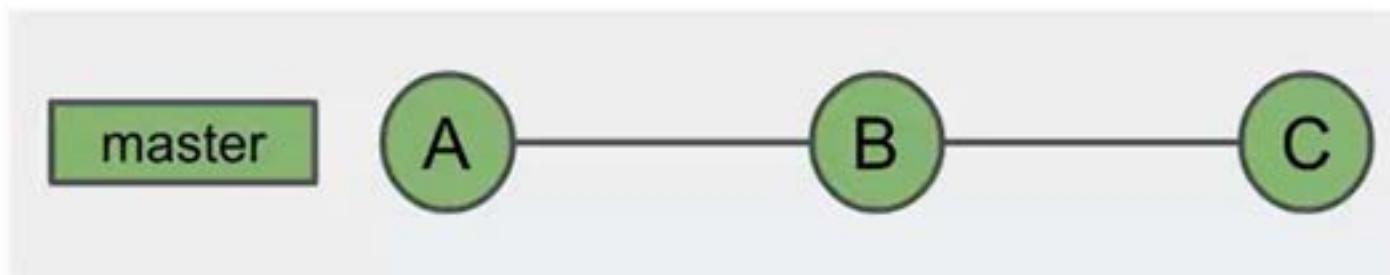
REMINDER- BRANCHES

- All commits belong to a *branch*
- By default, there is a single branch and it is called *master*



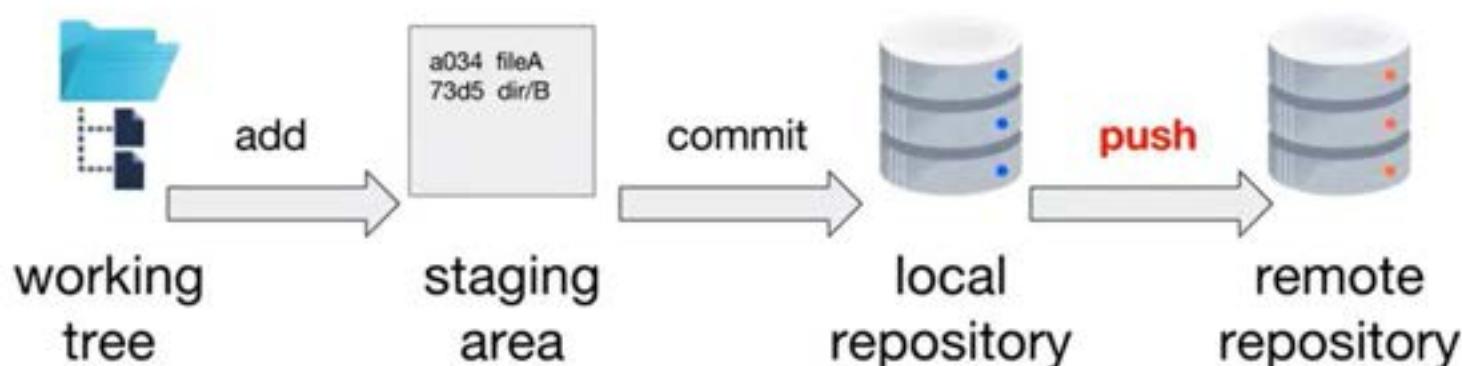
REMINDER- BRANCHES

- All commits belong to a *branch*
- By default, there is a single branch and it is called *master*

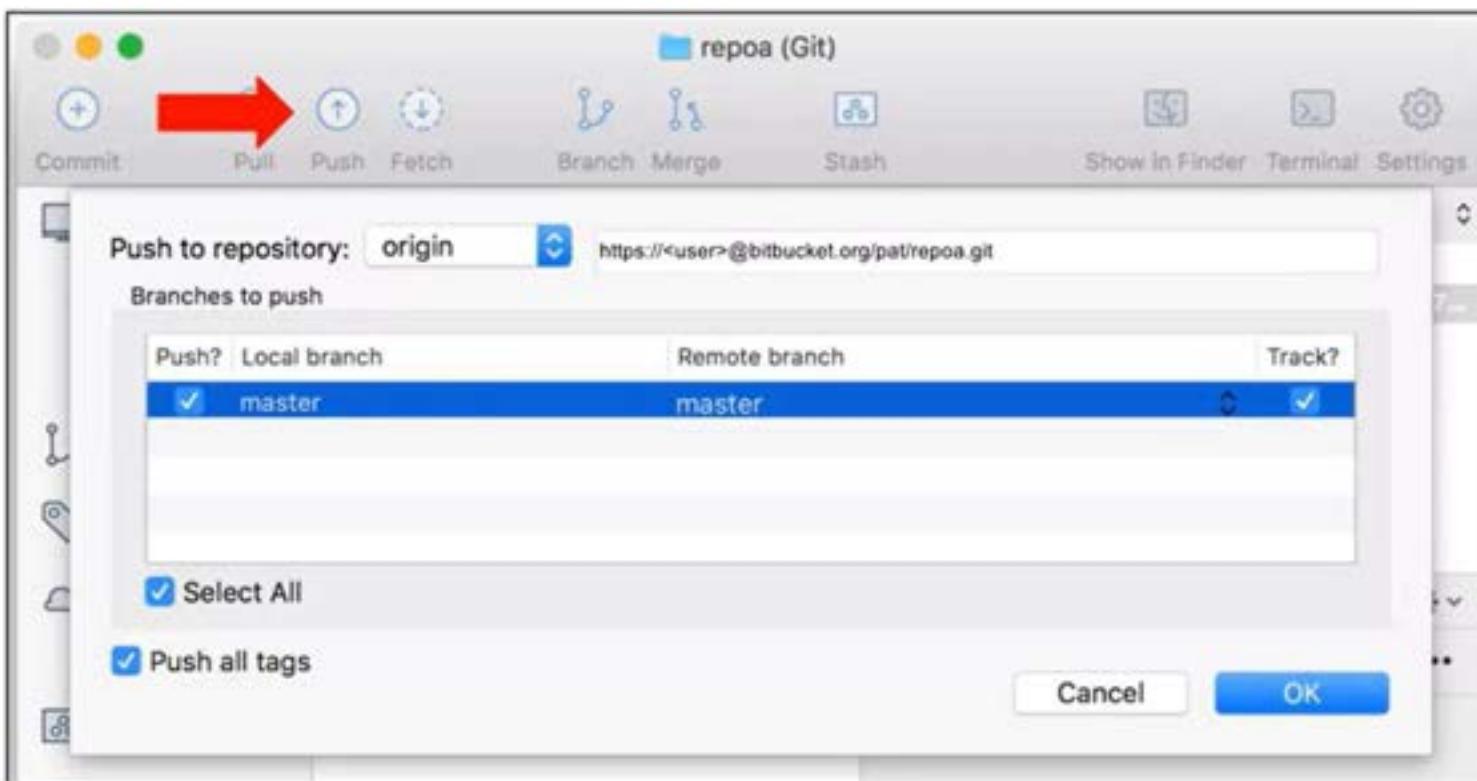


PUSH A COMMIT TO A REMOTE REPOSITORY

A *push* add commits for a branch to a remote repository



PUSH



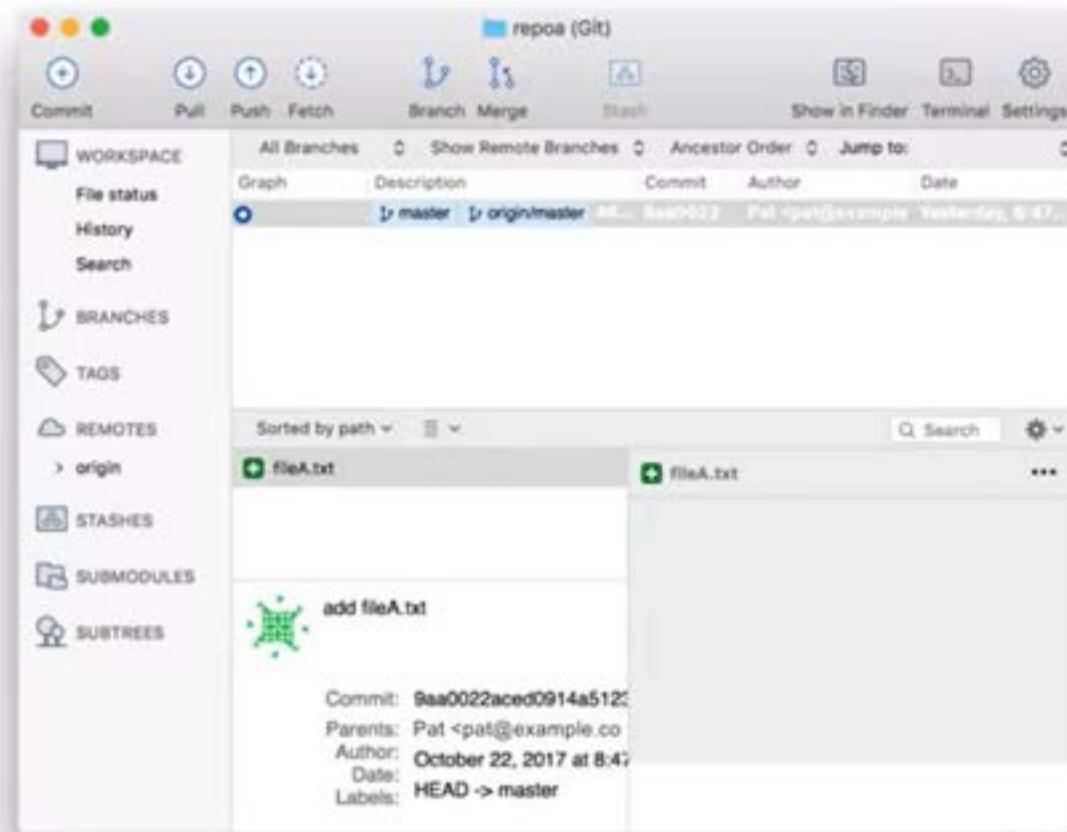
git push can be used to push commits to a remote repository using a command line

VIEW THE COMMIT ON BITBUCKET

The screenshot shows the Bitbucket interface for a repository named 'repoa'. A red arrow points to the 'Commits' link in the sidebar menu. The main area displays the commit history with one entry:

Author	Commit	Message
Pat	9aa0022	add fileA.txt

LOCAL AND REMOTE BRANCHES ARE SYNCHRONIZED

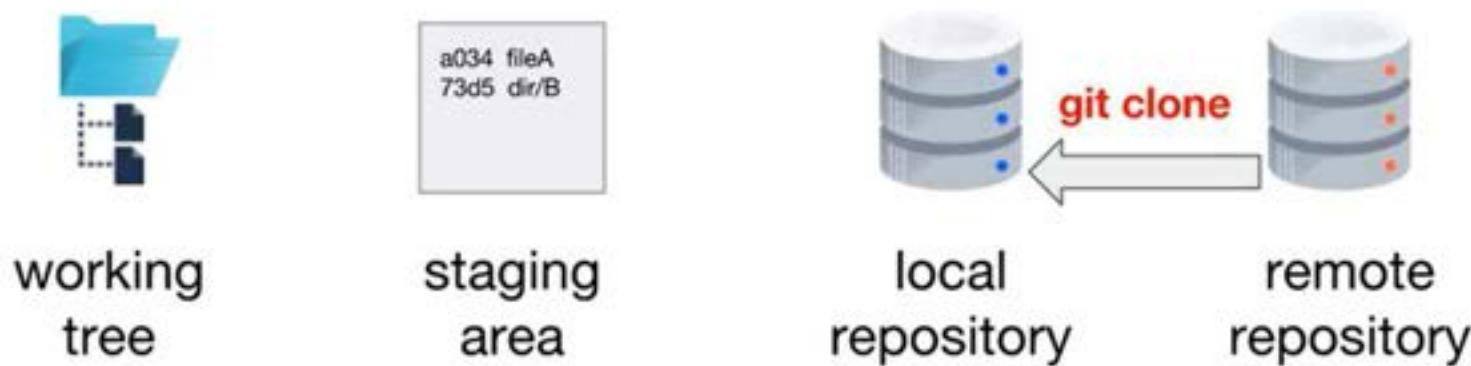


TWO SCENARIOS- STARTING WITH A REMOTE REPOSITORY

Have a local repository?	Task
no	<i>clone</i> the remote
yes	<i>add</i> the remote

WHAT IS git clone?

`git clone` is used to create a local copy of a remote repository



CLONING A REMOTE REPOSITORY

```
git clone <url/to/projectname.git> [localprojectname]
```

```
repos$ git clone https://bitbucket.org/atlassian_tutorial/helloworld.git
```

CLONING A REMOTE REPOSITORY

```
git clone <url/to/projectname.git> [localprojectname]
```

```
repos$ git clone https://bitbucket.org/atlassian_tutorial/helloworld.git
Cloning into 'helloworld'...
remote: Counting objects: 35, done.
remote: Compressing objects: 100% (21/21), done.
remote: Total 35 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (35/35), done.
$ cd helloworld
$ ls -a
.  ..  .git LICENSE README pom.xml src
```



REMOTE REPOSITORY INFORMATION

git remote --verbose

- Displays information about remote repositories associated with the local repository

```
repos$ git clone https://bitbucket.org/atlassian_tutorial/helloworld.git
Cloning into 'helloworld'...
$ cd helloworld
helloworld$ git remote -v
origin  https://bitbucket.org/atlassian_tutorial/helloworld.git (fetch)
origin  https://bitbucket.org/atlassian_tutorial/helloworld.git (push)
```

ADD A REMOTE REPOSITORY

```
git remote add <name> <url>
```

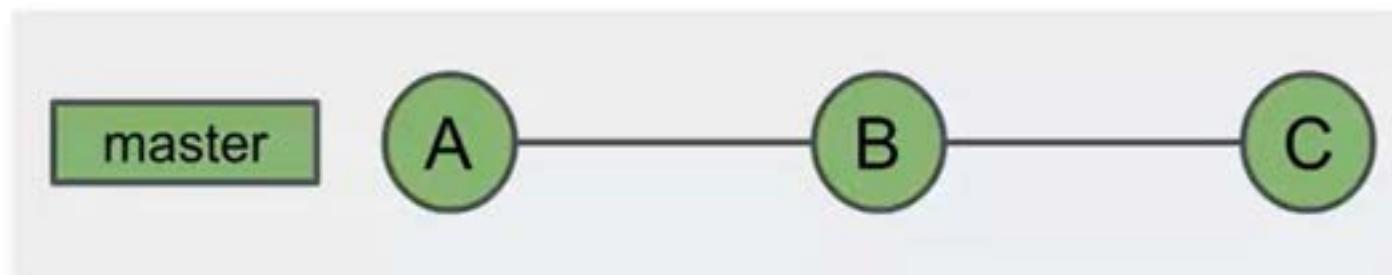
ADD A REMOTE REPOSITORY

```
git remote add <name> <url>
```

```
(create a remote Bitbucket repository named repoa)
repoa$ git remote add origin https://me@bitbucket.org/me/repoa.git
```

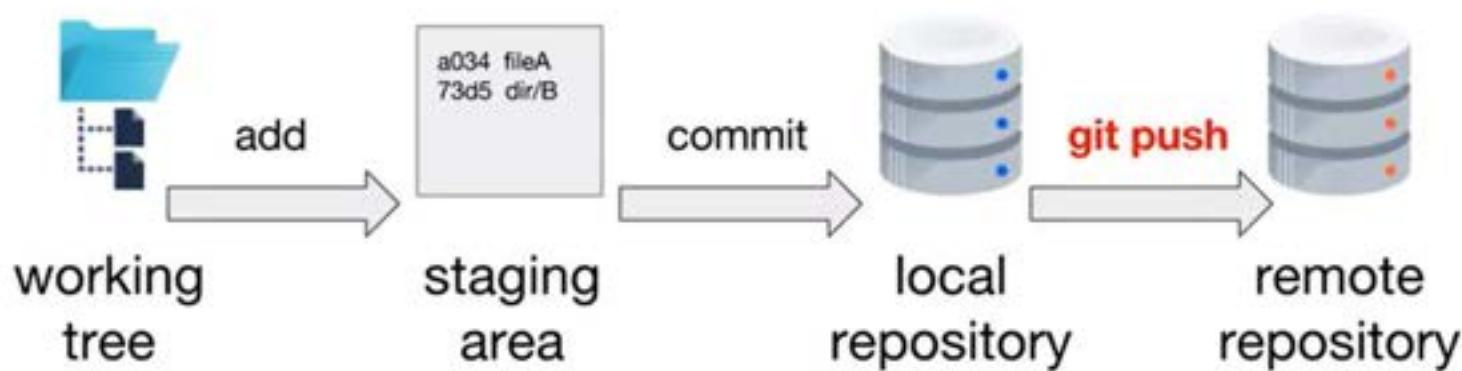
REMINDER- BRANCHES

- All commits belong to a *branch*
- By default, there is a single branch and it is called *master*



PUSH A COMMIT TO A REMOTE REPOSITORY

git push writes commits for a branch to a remote repository



PUSHING LOCAL COMMITS TO REMOTE

git push [-u] [<repository>] [<branch>]

- <repository> can be a name (shortcut) or URL
- -u track this branch (--set-upstream)

```
repoa$ git remote -v
origin  https://bitbucket.org/me/repoa.git (fetch)
origin  https://bitbucket.org/me/repoa.git (push)
repoa$ git push -u origin master
Username for 'https://me@bitbucket.org': me # you may need to set up an app pw
Password for 'https://me@bitbucket.org':
Counting objects: 3, done.
Writing objects: 100% (3/3), 223 bytes | 223.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://bitbucket.org/me/repoa.git
 * [new branch]      master -> master
Branch master set up to track remote branch master from origin.
```



ADDING COMMITS- STARTING WITH NO LOCAL REPOSITORY

Put some bits in your bucket

Add some code or content and start bringing your ideas to life. [Learn how](#)

Get started the easy way

Creating a README or a .gitignore is a quick and easy way to get something into your repository.

[Create a README](#)

[Create a .gitignore](#)

Get started with command line

› [I have an existing project](#)

✗ [I'm starting from scratch](#)

```
1 git clone https://me@bitbucket.org/me/repoa.git
2 cd repoa
3 echo "# My project's README" >> README.md
4 git add README.md
5 git commit -m "Initial commit"
6 git push -u origin master
```

ADDING COMMITS- STARTING WITH NO LOCAL REPOSITORY

Put some bits in your bucket

Add some code or content and start bringing your ideas to life. [Learn how](#)

Get started the easy way

Creating a README or a .gitignore is a quick and easy way to get something into your repository.

[Create a README](#)

[Create a .gitignore](#)

Get started with command line

› I have an existing project

✗ I'm starting from scratch

```
1 git clone https://me@bitbucket.org/me/repoa.git
2 cd repoa
3 echo "# My project's README" >> README.md
4 git add README.md
5 git commit -m "Initial commit"
6 git push -u origin master
```



ADDING COMMITS- STARTING WITH NO LOCAL REPOSITORY

```
repos$ git clone https://me@bitbucket.org/me/repoa.git
repos$ cd repoa
repoa$ echo "# My project's README" >> README.md
repoa$ git add README.md
repoa$ git commit -m "Initial commit"
repoa$ git push -u origin master
```

ADDING COMMITS- STARTING WITH NO LOCAL REPOSITORY

Press Esc to exit full screen

```
repos$ git clone https://me@bitbucket.org/me/repoa.git
repos$ cd repoa
repoa$ echo "# My project's README" >> README.md
repoa$ git add README.md
repoa$ git commit -m "Initial commit"
repoa$ git push -u origin master
```

ADDING COMMITS- EXISTING LOCAL REPOSITORY

Get started the easy way

Creating a README or a .gitignore is a quick and easy way to get something into your repository.

[Create a README](#)

[Create a .gitignore](#)

Get started with command line

▼ I have an existing project

Step 1: Switch to your repository's directory

```
1 cd /path/to/your/repo
```

Step 2: Connect your existing repository to Bitbucket

```
1 git remote add origin https://me@bitbucket.org/me/repoa.git  
2 git push -u origin master
```

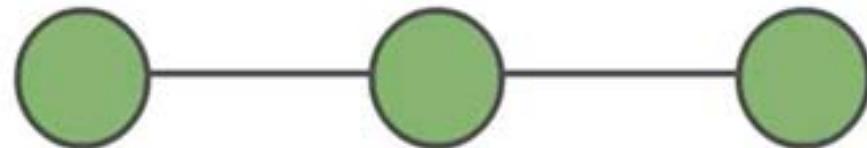
➤ I'm starting from scratch

HANDS ON

- 1) Clone your Bitbucket repository
- 2) Create a commit in the local repository
- 3) Push the commit to the remote repository

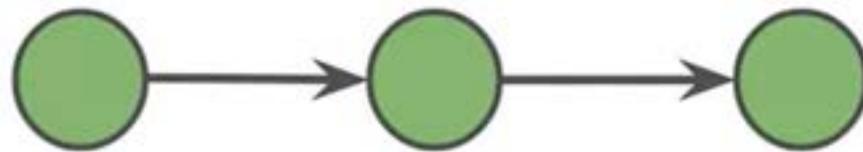
GRAPH

- A way to model connected things



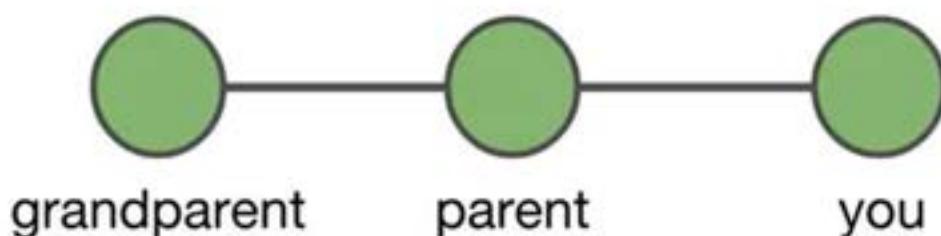
DIRECTED GRAPH

Nodes are connected in a certain direction



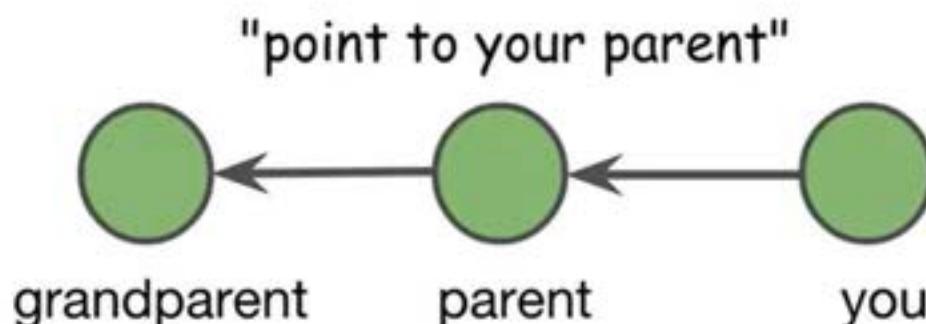
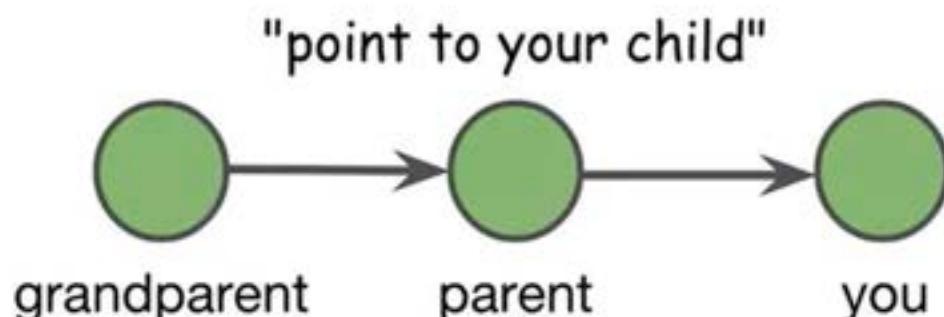
ARROW DIRECTION

Direction depends on how you define the relationship



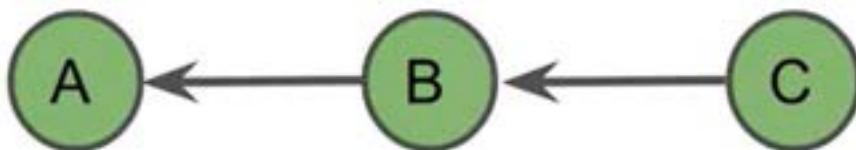
ARROW DIRECTION

Direction depends on how you define the relationship

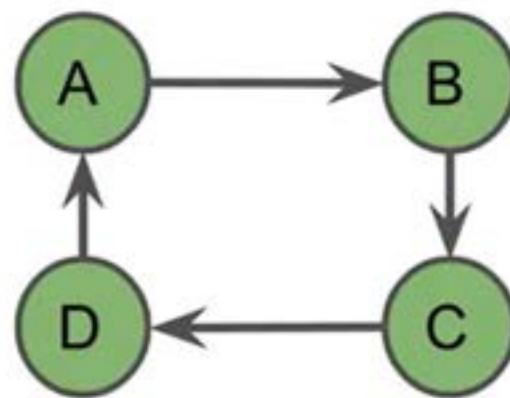


ACYCLIC

Acyclic mean "no cycles" or "non-circular"



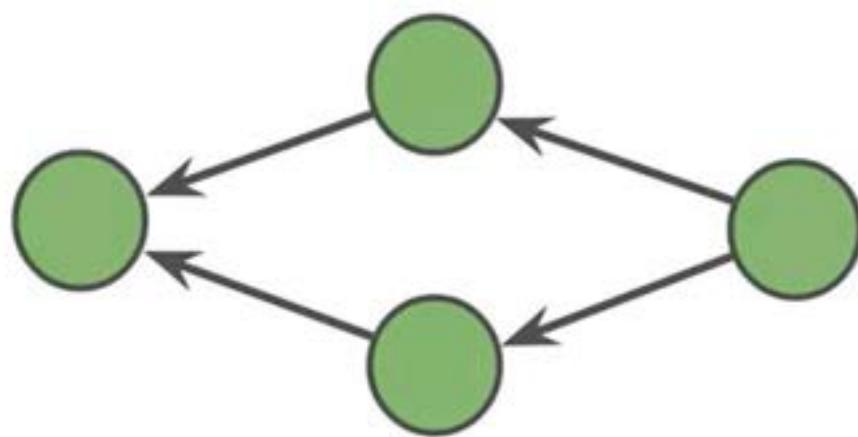
acyclic



cyclic

DIRECTED ACYCLIC GRAPH (DAG)

Contains nodes connected with arrows and has no cycles

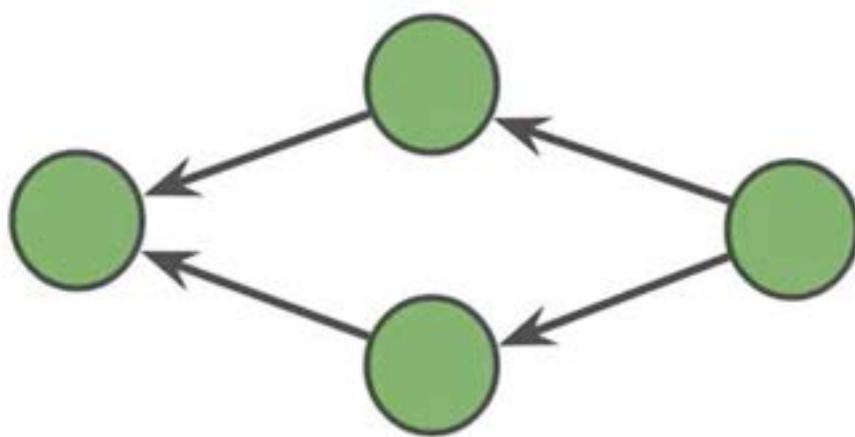


directed acyclic graph (DAG)

DIRECTED ACYCLIC GRAPH

(DA) Press Esc to exit full screen

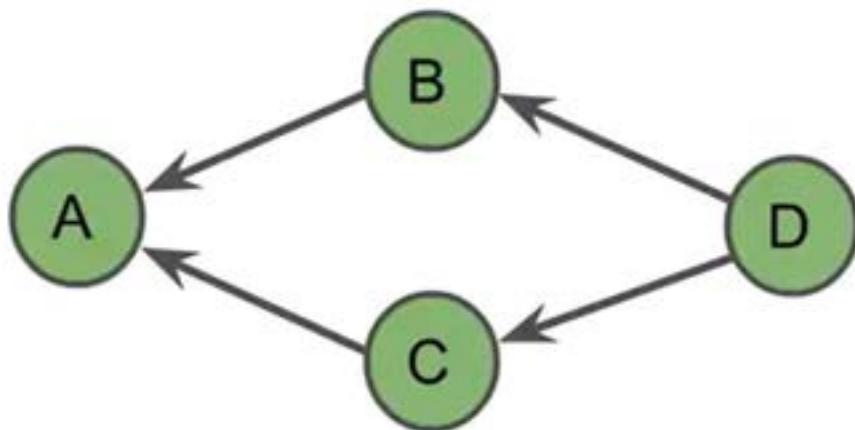
Contains nodes connected with arrows and has no cycles



directed acyclic graph (DAG)

GIT'S DAG

- Git models the relationship of commits with a DAG
- The arrows point at a commit's parent(s)



VIEWING GRAPHS IN GIT CLIENTS



Graph	Description	Commit
	⚡ master Merge branch 'feature1'	f8cbf90
	add feature 1	4684dcf
	add feature 1 wip	1cd70bb
	add fileA.txt	c126179

Sourcetree

VIEWING GRAPHS IN GIT CLIENTS



Graph	Description	Commit
	⚡ master Merge branch 'feature1'	f8cbf90
	add feature 1	4684dcf
	add feature 1 wip	1cd70bb
	add fileA.txt	c126179

Sourcetree

VIEWING GRAPHS IN GIT CLIENTS



Graph	Description	Commit
	↳ master Merge branch 'feature1'	f8cbf90
	add feature 1	4684dcf
	add feature 1 wip	1cd70bb
	add fileA.txt	c126179

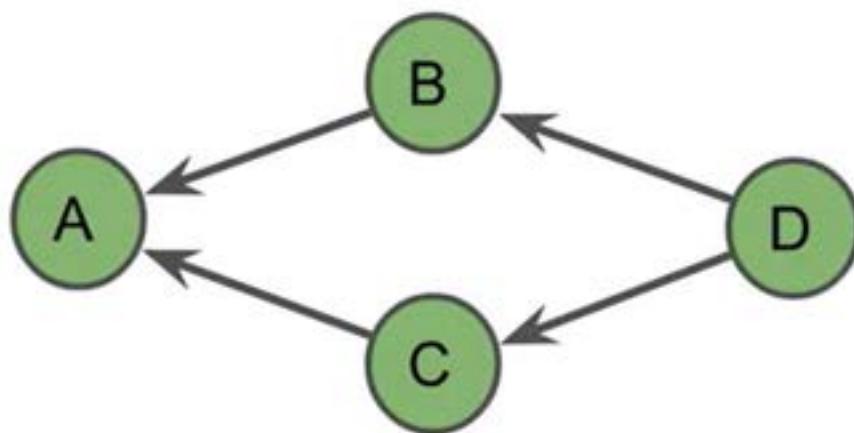
Sourcetree



```
$ git log --oneline --graph
*   f8cbf90 (HEAD -> master) Merge branch 'feature1'
|\ 
| * 4684dcf add feature 1
| * 1cd70bb add feature 1 wip
|/
* c126179 add fileA.txt
```

command line

REVIEW



- Git uses a directed acyclic graph (DAG) to represent commit history
- Commits point to their **parent** commits

Topics

Git Objects

Git IDs

Shortening Git IDs

GIT ID

- The **name** of a Git object
- 40-character hexadecimal string
- Also known as *object ID*, *SHA-1*, *hash* and *checksum*



SECURE HASH ALGORITHM 1 (SHA-1)

- Git IDs are *SHA-1 values*
- Unique for a given piece of content (statistically speaking)

1af751ec24a0b99f8580d947c861c50581447602

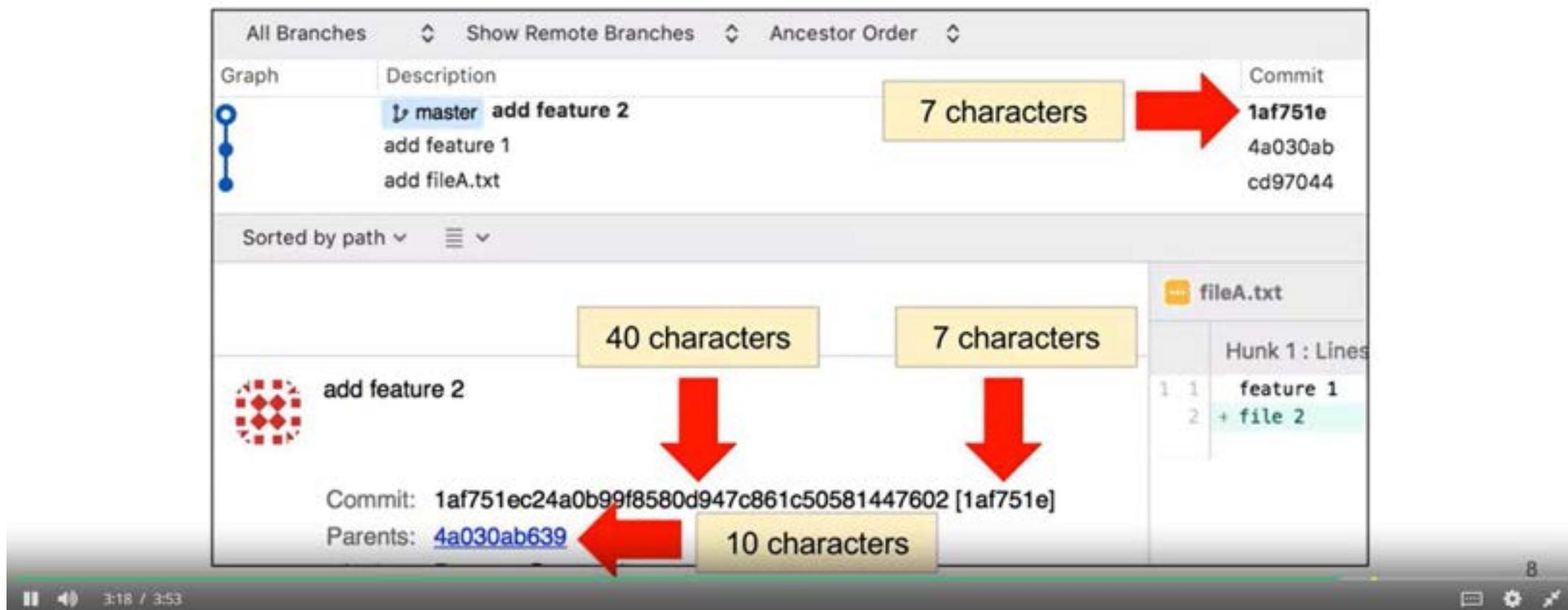
<https://en.wikipedia.org/wiki/SHA-1>

SHA-1 IS DESIGNED TO AVALANCHE

- SHA-1 of "hi"
 - **45b983be**36b73c0788dc9cbb76cbb80fc7bb057
- SHA-1 of "hi " (with a trailing space)
 - **0b5d6ed**361eb779d2212ec820ea5568f3af7762b

SHORTENED GIT IDS

The first portion of the Git ID



REVIEW

- Git object names are also known as Git IDs
- Git objects are named with SHA-1 values
- SHA-1 values are unique for a given piece of content (statistically speaking)
- Git IDs are often shortened to the first four or more characters

SECURE HASH ALGORITHM 1 (SHA-1)

- Git IDs are *SHA-1 values*
- Unique for a given piece of content (statistically speaking)

```
$ git log
commit e8d41c0322e629aa7d7aa8a9a1335bff2f76f72 (HEAD -> master)
...
...
```

<https://en.wikipedia.org/wiki/SHA-1>

SHA-1 IS DESIGNED TO AVALANCHE

- SHA-1 of "hi"
 - **45b983be**36b73c0788dc9cbb76cbb80fc7bb057
- SHA-1 of "hi " (with a trailing space)
 - **0b5d6ed**361eb779d2212ec820ea5568f3af7762b

CREATING A SHA-1 FOR FILE CONTENTS

Use `git hash-object <file>` to create an SHA-1 for any content

```
$ echo "hi" > fileA.txt
$ git hash-object fileA.txt
45b983be36b73c0788dc9cbcb76cbb80fc7bb057
$ git hash-object fileA.txt
45b983be36b73c0788dc9cbcb76cbb80fc7bb057
$ echo "hi " > fileA.txt # add a space after hi
$ git hash-object fileA.txt
0b5d6ed361eb779d2212ec820ea5568f3af7762b
```

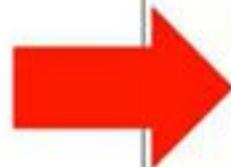
<https://git-scm.com/docs/git-hash-object>

PLUMBING VS. PORCELAIN

<https://git-scm.com/docs>

Plumbing Commands

- cat-file
- check-ignore
- commit-tree
- count-objects
- diff-index
- for-each-ref
- hash-object
- ls-files
- merge-base
- read-tree
- rev-list
- rev-parse
- show-ref
- symbolic-ref
- update-index
- update-ref
- verify-pack
- write-tree



<https://git-scm.com/book/en/v2/Git-Internals-Plumbing-and-Porcelain>

SHORTENED GIT IDS

Four or more characters of the beginning of a Git ID

```
$ git log --oneline  
483d057 (HEAD -> master) add README.md
```

SHORTENED GIT IDS

Press Esc to exit full screen

Four or more characters of the beginning of a Git ID

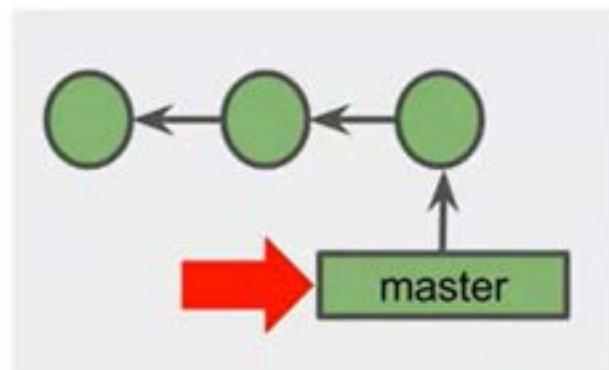
```
$ git log --oneline  
483d057 (HEAD -> master) add README.md  
$ git log  
commit 483d0572148a7bfed924a1f7bee20de6d9364942 (HEAD -> master)  
Author: ...  
$ git show 483d  
commit 483d0572148a7bfed924a1f7bee20de6d9364942 (HEAD -> master)  
Author: ...
```

REVIEW

- Git object names are also known as Git IDs
- Git objects are named with SHA-1 values
- SHA-1 values are unique for a given piece of content (statistically speaking)
- Git IDs are often shortened to the first four or more characters

BRANCH LABEL

- Points to the most recent commit in the branch
 - The "tip of the branch"
- Implemented as a reference

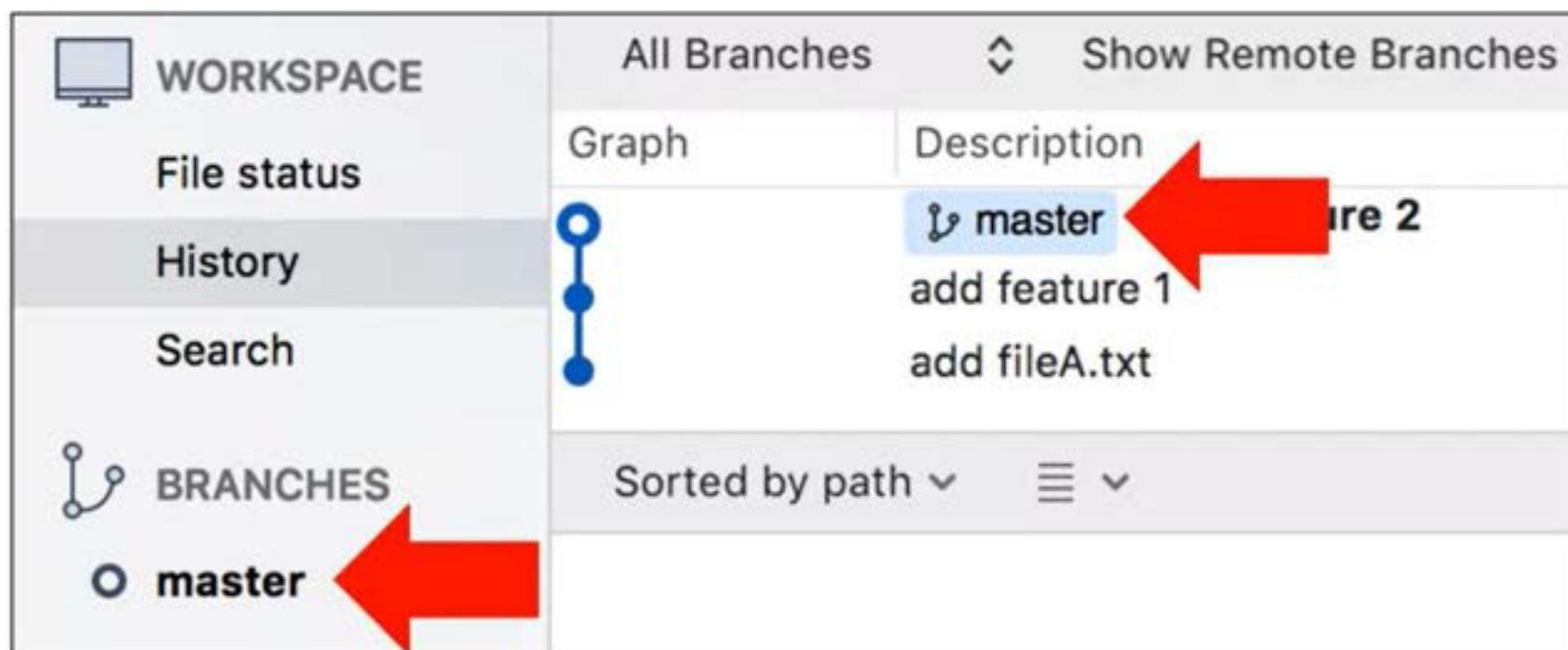


Graph	Description
	▶ master
	add feature 1
	add fileA.txt

A screenshot of a software interface showing a commit history. The "Graph" column displays a vertical sequence of three blue circular nodes connected by a single vertical line. The "Description" column lists three commits: "▶ master", "add feature 1", and "add fileA.txt". A red arrow points from the word "master" in the "Description" column towards the top node in the "Graph" column. The word "master" is highlighted with a light blue background.

MASTER

master is the default name of the main branch in the repository



REFERENCE

User-friendly name that points to:

- a commit SHA-1 hash
- another reference
 - known as a *symbolic reference*

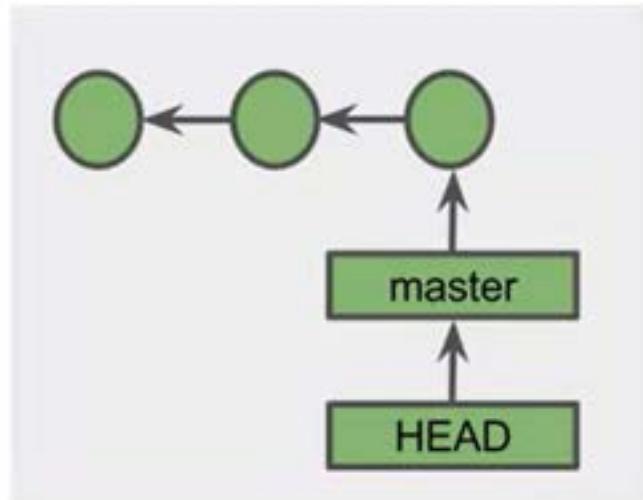
add feature 2

Commit: 1af751ec24a0b99f8580d947c861c50581447602 [1af751e]
Parents: [4a030ab639](#)
Author: Pat <pat@example.com>
Date: November 17, 2017 at 10:44:53 AM PST
Labels: HEAD -> master

A large red arrow points from the text "HEAD -> master" back towards the heading "User-friendly name that points to:" in the slide's content area.

HEAD

- A reference to the current commit
- Usually points to the branch label of the current branch
- One *HEAD* per repository



add feature 2

Commit: 1af751ec24a0b99f8580d947c861c50581447602 [1af751e]

Parents: [4a030ab639](#)

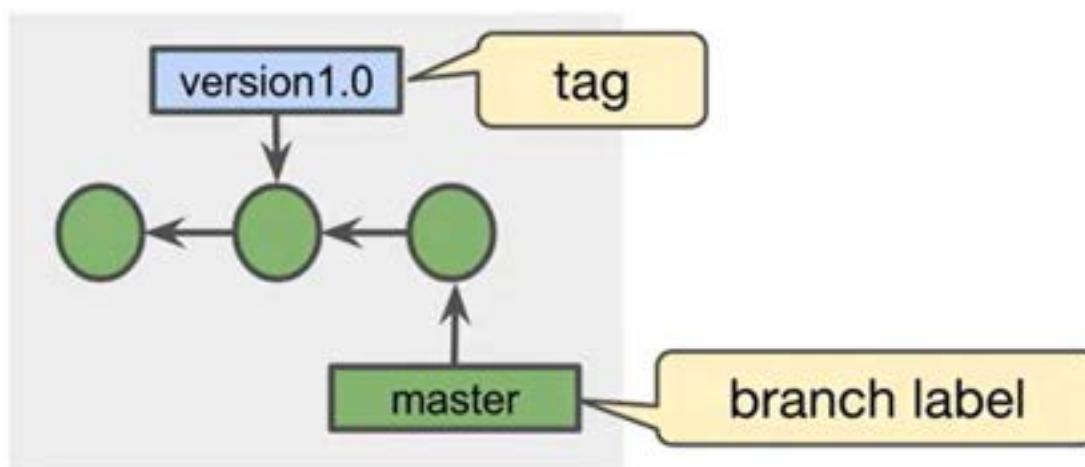
Author: Pat <pat@example.com>

Date: November 17, 2017 at 10:44:53 AM PST

Labels: HEAD -> master

TAGS

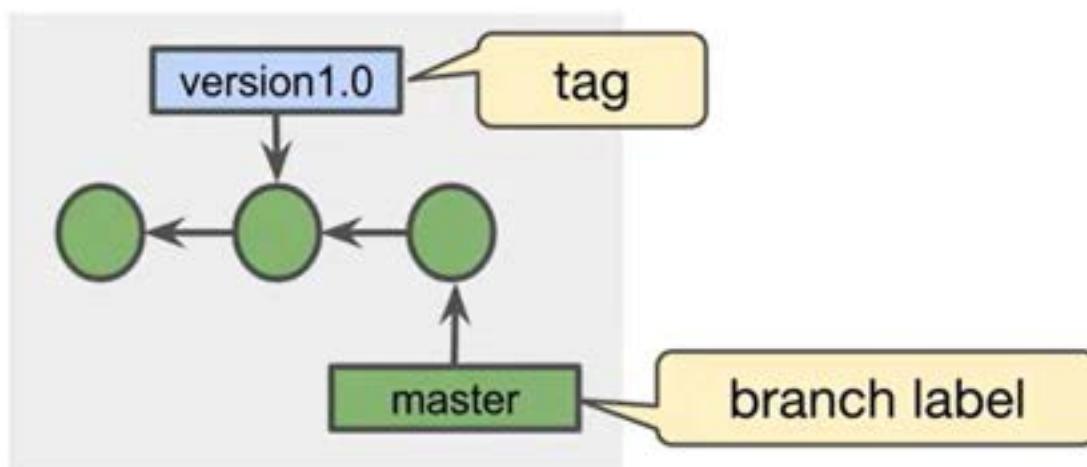
Reference/label attached to a specific commit



<https://git-scm.com/docs/git-tag>

TAGS

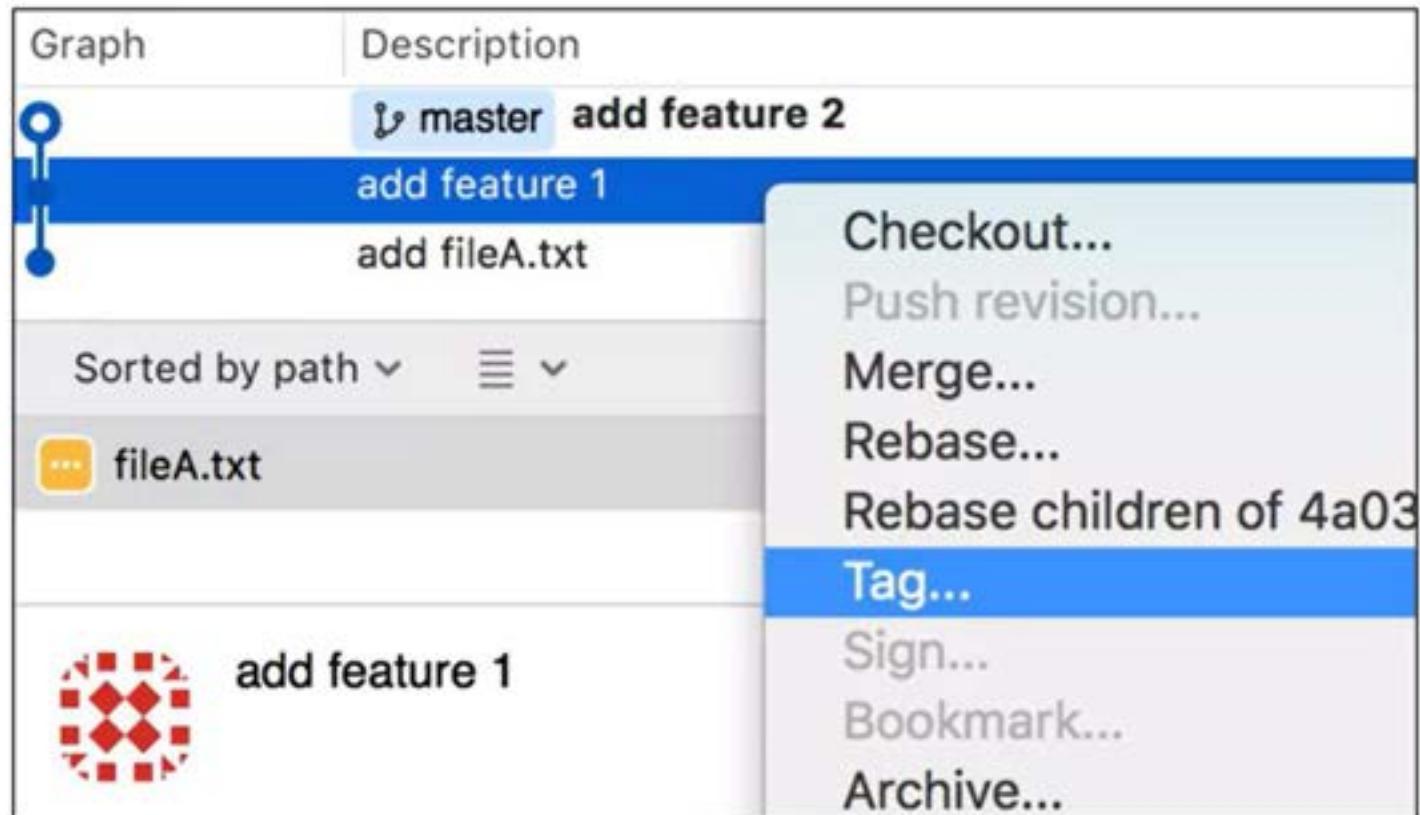
Reference/label attached to a specific commit



<https://git-scm.com/docs/git-tag>

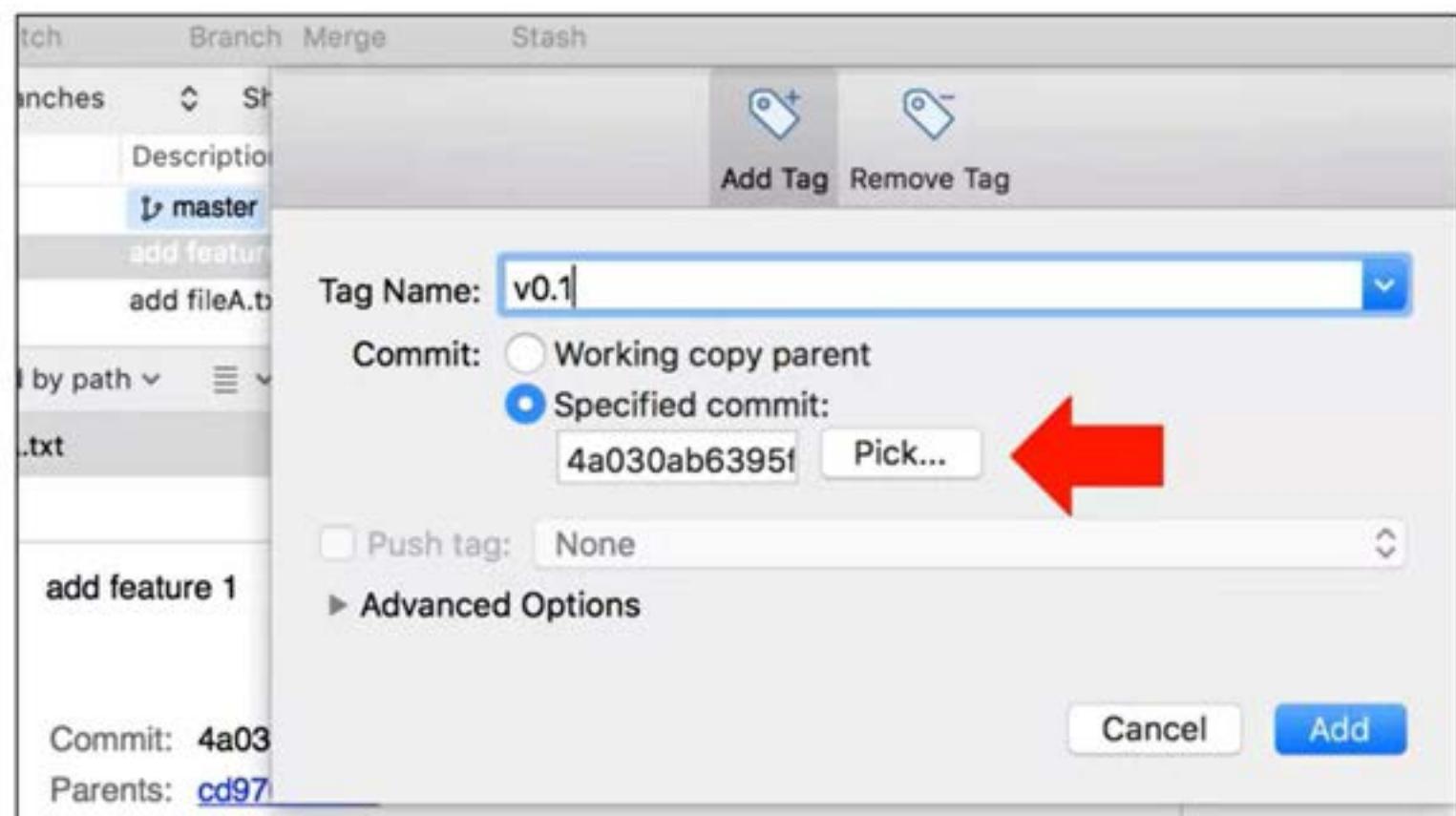
ADDING A TAG

Right-click on the commit that you want to tag and select **Tag...**



ADDING A TAG

- The commit should already be specified
- Add a **Tag Name** and click **Add**



REVIEW

- A branch label is a reference that points to the tip of the branch
- HEAD is a reference that points to the current commit
- Create tags to place labels on specific commits
- Tags are not automatically pushed to remote repositories

Topics

Overview of references

Branch labels and HEAD

Referencing prior commits with
~ and ^

Tags

REFERENCE

User-friendly name that points to:

- a commit SHA-1 hash
- another reference
 - known as a *symbolic reference*

```
$ git log --oneline
1ef16ac (HEAD -> master) add README.md
e8d41c0 initial commit
```

USING REFERENCES WITH GIT COMMANDS

Use references instead of SHA-1 hashes

```
$ git log  
commit lef16ac15dbd60a779a6c8b5f3cc1ce39383fa2c (HEAD -> master)  
...  
$ git show HEAD  
commit lef16ac15dbd60a779a6c8b5f3cc1ce39383fa2c (HEAD -> master)  
Author: Pat <pat@example.com>  
Date:   Tue Sep 19 16:50:07 20xx -0700  
    changed fileA.txt to version 2  
diff --git a/fileA.txt b/fileA.txt  
...
```

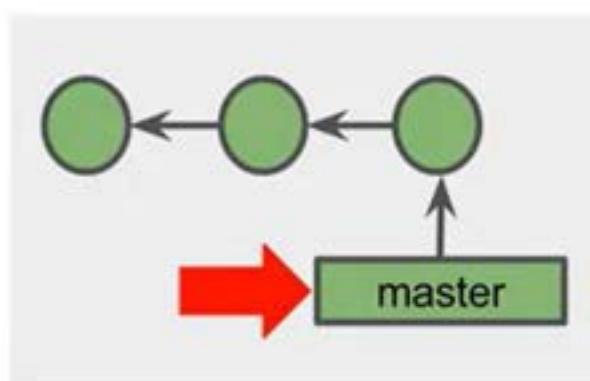
MASTER

master is the default name of the main branch in the repository

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

BRANCH LABEL

- Points to the most recent commit in the branch
 - The "tip of the branch"
- Implemented as a reference

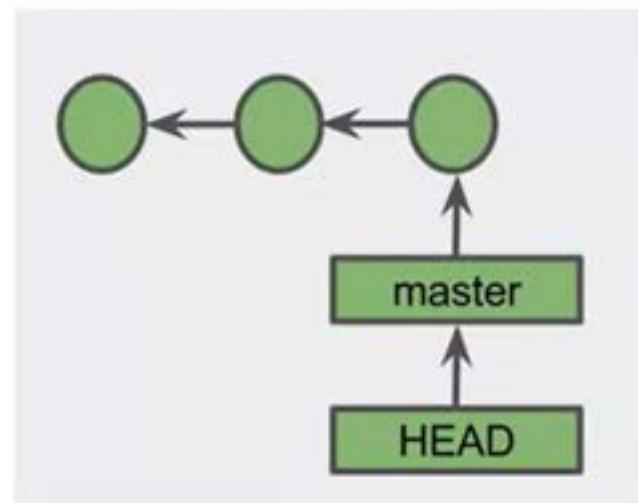


VIEWING LOCAL BRANCH REFERENCES IN .git/refs/heads

```
myproj$ cd .git
.git$ ls
COMMIT_EDITMSG ORIG_HEAD config      hooks      info      objects
HEAD           branches  description index    logs      refs
.git$ cd refs
refs$ ls
heads   tags
refs$ cd heads
heads$ ls
master
heads$ cat master
1ef16ac15dbd60a779a6c8b5f3cc1ce39383fa2c
```

HEAD

- A reference to the current commit
- Usually points to the branch label of the current branch
- One *HEAD* per repository



```
$ git log --oneline -1  
483d057 (HEAD -> master) add feature 2
```

VIEWING HEAD IN THE .GIT DIRECTORY

A reference file named .git/HEAD

```
myproj$ cd .git
.git$ ls
COMMIT_EDITMSG ORIG_HEAD config      hooks      info      objects
HEAD           branches  description index      logs      refs
.git$ cat HEAD
ref: refs/heads/master
```

APPENDING TILDE (~) TO GIT IDS AND REFERENCES

Refers to a prior commit

- ~ or ~1 = parent
- ~2 or ~~ = parent's parent

```
$ git log --oneline --graph
* e0cb6c5 (HEAD -> master, tag: v2.0) added feature 2 to fileA.txt
* lef16ac changed fileA.txt to version 2
* e8d41c0 updated fileA.txt feature 1
* 14b97e6 added fileA.txt
$ git show HEAD
(shows commit info for e0cb6c5)
$ git show HEAD~ # same as HEAD~1
(shows commit info for lef16ac)
$ git show master~3
(shows commit info for 14b97e6)
$ git show e0cb6c5~~~
(shows commit info for 14b97e6)
```

APPENDING CARET (^) TO GIT IDS AND REFERENCES

Refers to a parent in a merge commit (^parentnum)

- ^ or ^1- first parent of the commit
- ^2- second parent of a merge commit
- ^^ - first parent's first parent

```
$ git log --oneline --graph
* e0cb6c5 (HEAD -> master, tag: v2.0) added feature 2 to fileA.txt
* lef16ac changed fileA.txt to version 2
* e8d41c0 updated fileA.txt feature 1
* 14b97e6 added fileA.txt
$ git show master^
(shows commit info for lef16ac)
$ git show HEAD^2
fatal: ambiguous argument 'HEAD^2': unknown revision or path not in the working tree.
$ git show HEAD^^
(shows commit info for e8d41c0)
```

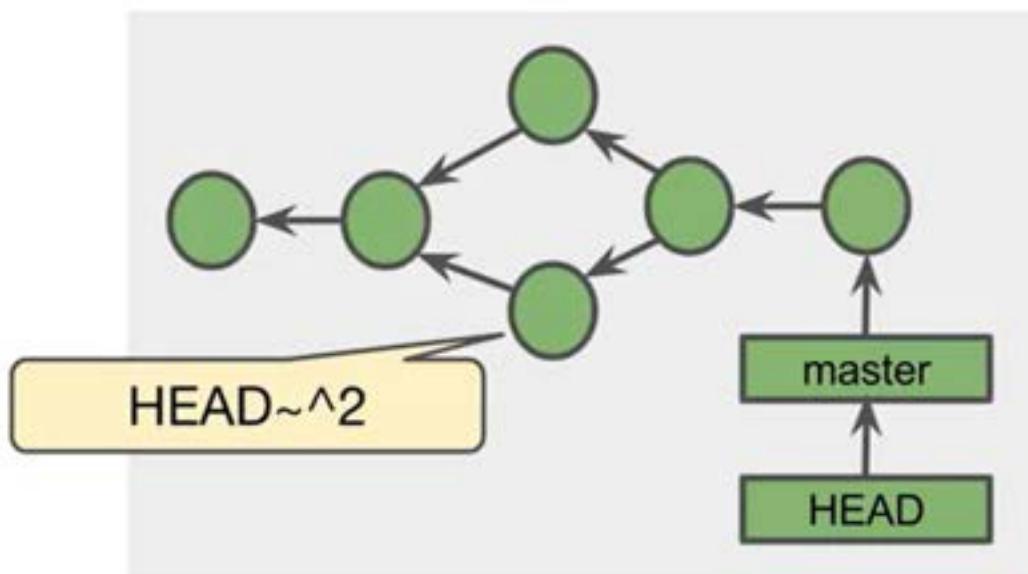
APPENDING CARET (^) TO GIT IDS AND REFERENCES

Refers to a parent in a merge commit (^parentnum)

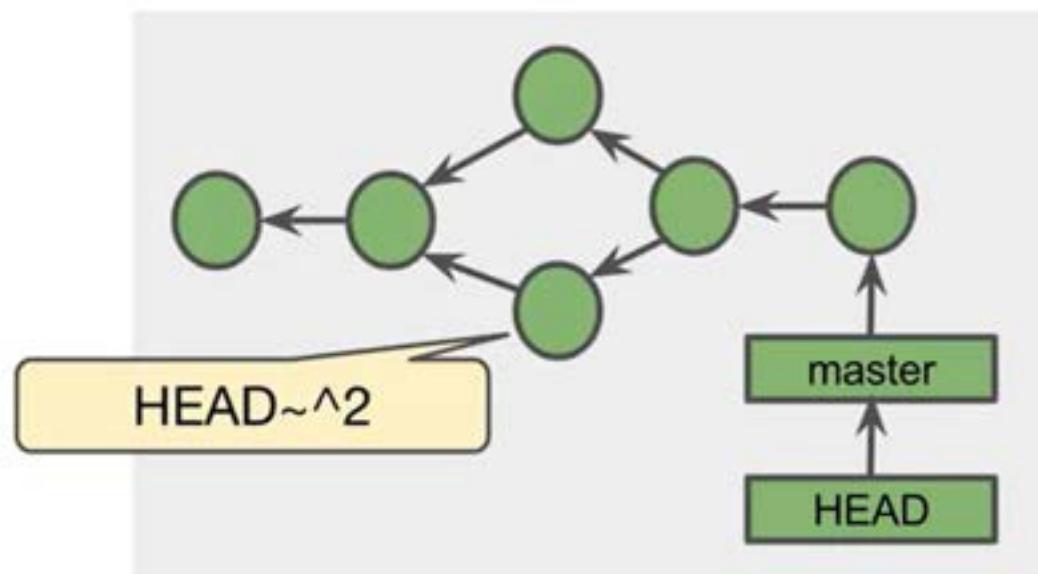
- ^ or ^1- first parent of the commit
- ^2- second parent of a merge commit
- ^^ - first parent's first parent

```
$ git log --oneline --graph
* e0cb6c5 (HEAD -> master, tag: v2.0) added feature 2 to fileA.txt
* lef16ac changed fileA.txt to version 2
* e8d41c0 updated fileA.txt feature 1
* 14b97e6 added fileA.txt
$ git show master^
(shows commit info for lef16ac)
$ git show HEAD^2
fatal: ambiguous argument 'HEAD^2': unknown revision or path not in the working tree.
$ git show HEAD^^
(shows commit info for e8d41c0)
```

COMBINING ~ AND ^

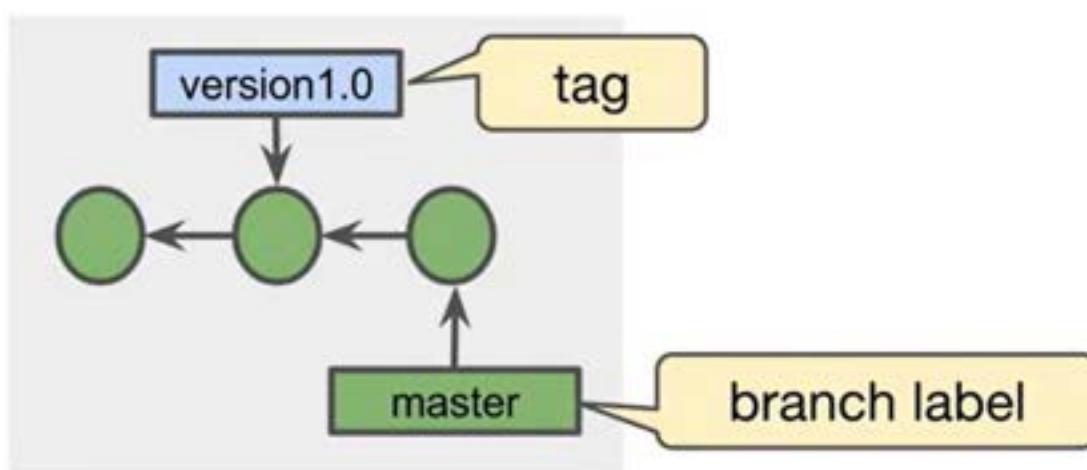


COMBINING ~ AND ^



TAGS

Reference/label attached to a specific commit



<https://git-scm.com/docs/git-tag>

TYPES OF TAGS

- **Lightweight**
 - A simple reference to a commit
- **Annotated**
 - A full Git object that references a commit
 - Includes tag author information, tag date, tag message, the commit ID
 - Optionally can be signed and verified with GNU Privacy Guard (GPG)

VIEWING AND USING TAGS

- `git tag`- View all tags in the repository
- Tags can be used instead of branch labels or Git IDs in Git commands

```
$ git tag
v0.1
v1.0
$ git show v0.1
commit e8d41c0322e629aaaf7d7aa8a9a1335bff2f76f72 (tag: v0.1)
(commit info)
```

CREATING A LIGHTWEIGHT TAG

To tag a commit with a lightweight tag:

- `git tag <tagname> [<commit>]`
- `<commit>` defaults to HEAD

```
$ git tag v1.0 # tag the current commit
$ git tag # view tags
v1.0
$ git tag v0.1 HEAD^ # tag the previous commit
$ git tag
v0.1
v1.0
$ git show v0.1
commit e8d41c0322e629aa7d7aa8a9a1335bff2f76f72 (tag: v0.1)
(commit info)
```

CREATING AN ANNOTATED TAG

- To tag a commit with an annotated tag:
 - `git tag -a [-m <msg> | -F <file>] <tagname> [<commit>]`
 - `<commit>` defaults to HEAD
- `git show` displays the tag object information followed by the commit information

```
$ git tag -a -m "includes feature 2" v2.0
$ git show v2.0
tag v2.0
Tagger: Pat <pat@example.com>
Date:   Thu Sep 21 18:39:58 20XX-0700
includes feature 2

commit e0cb6c5eb3ebaceb30a860c713eaeae3901fa79e (HEAD -> master, tag: v2.0)
Author: ...
(commit information)
```

CREATING AN ANNOTATED TAG

- To tag a commit with an annotated tag:
 - `git tag -a [-m <msg> | -F <file>] <tagname> [<commit>]`
 - `<commit>` defaults to HEAD
- `git show` displays the tag object information followed by the commit information

```
$ git tag -a -m "includes feature 2" v2.0
$ git show v2.0
tag v2.0
Tagger: Pat <pat@example.com>
Date:   Thu Sep 21 18:39:58 20XX-0700
includes feature 2

commit e0cb6c5eb3ebaceb30a860c713eaeae3901fa79e (HEAD -> master, tag: v2.0)
Author: ...
(commit information)
```



TAGS AND REMOTE REPOSITORIES

- git push does not automatically transfer tags to the remote repository
- To transfer a single tag:
 - git push <remote> <tagname>
- To transfer all of your tags
 - git push <remote> --tags

VIEWING TAGS ON REMOTE

After pushing tags, log into Bitbucket and view them on the remote repository

The screenshot shows a commit history table with three rows. The first row has a green vertical branch line on the left, followed by the author 'Pat' with a question mark icon, the commit hash '1af751e', and the message 'add feature 2'. The second row has a green vertical branch line, followed by the author 'Pat' with a question mark icon, the commit hash '4a030ab', and the message 'add feature 1'. The third row has a green vertical branch line, followed by the author 'Pat' with a question mark icon, the commit hash 'cd97044', and the message 'add fileA.txt'. To the right of the commit details, there are two small rectangular boxes, each containing a dark grey tag icon and the tag name: 'v1.0' and 'v0.1'. A red rectangular box highlights these two tag boxes.

All branches	Author	Commit	Message	v1.0	v0.1
1	Pat	1af751e	add feature 2		
2	Pat	4a030ab	add feature 1		
3	Pat	cd97044	add fileA.txt		

REVIEW

- A branch label is a reference that points to the tip of the branch
- HEAD is a reference that points to the current commit
- In Git commands, use ~ and ^ to conveniently refer to previous commits
- Create tags to place labels on specific commits
- Tags are not automatically pushed to remote repositories

Topics

Branch overview

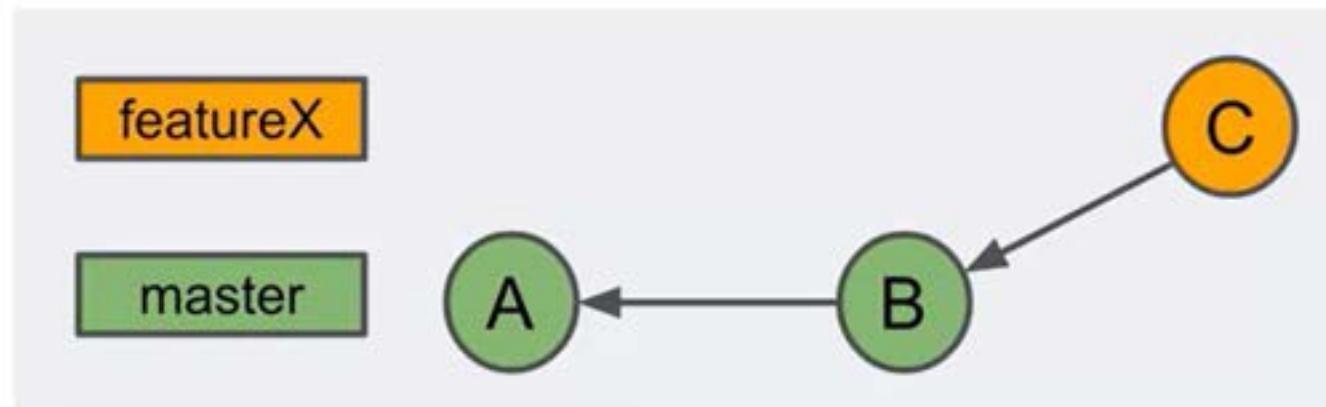
Creating a branch

Checkout

Detached HEAD

Deleting a branch label

WHAT IS A BRANCH?



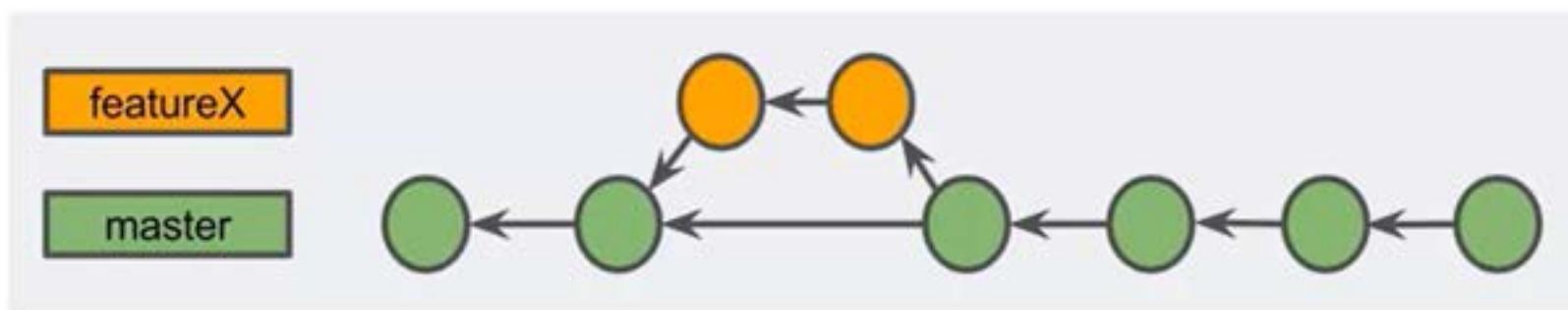
The set of commits that trace back to the project's first commit

BENEFITS OF BRANCHES

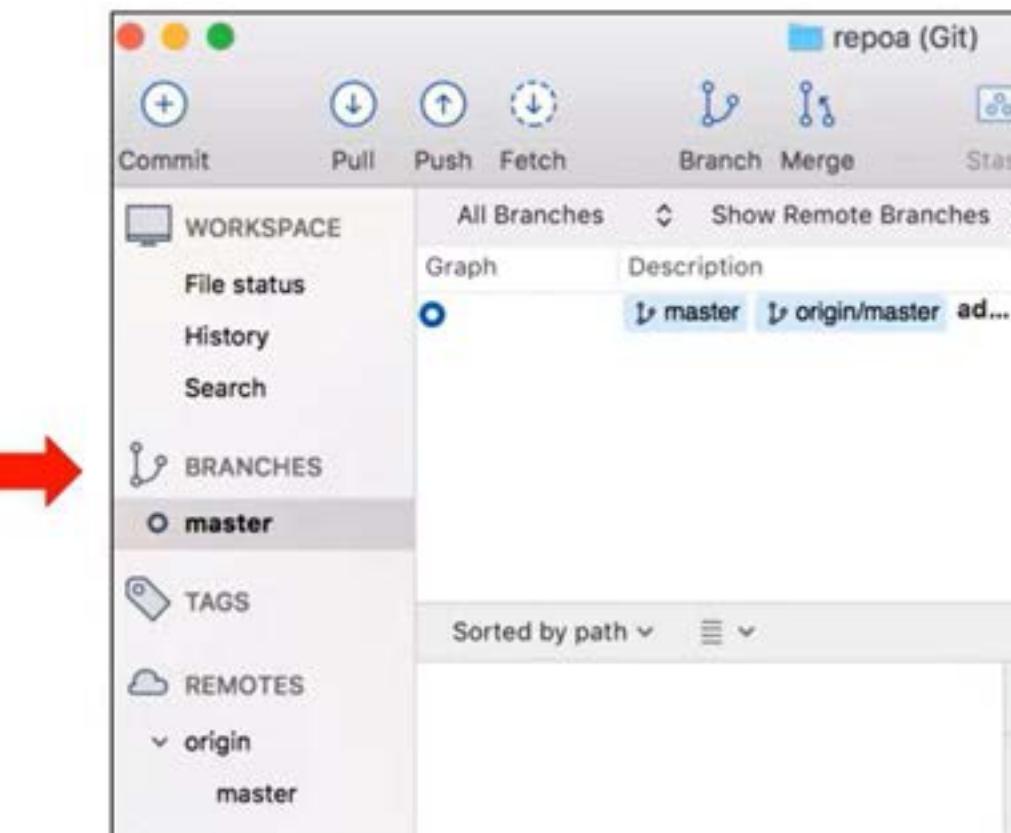
- Fast and easy to create
- Enable experimentation
- Enable team development
- Support multiple project versions

TOPIC AND LONG-RUNNING BRANCHES

- Topic
 - A feature, a bug fix, a hotfix, a configuration change, etc.
- Long-lived
 - **master, develop, release**, etc.

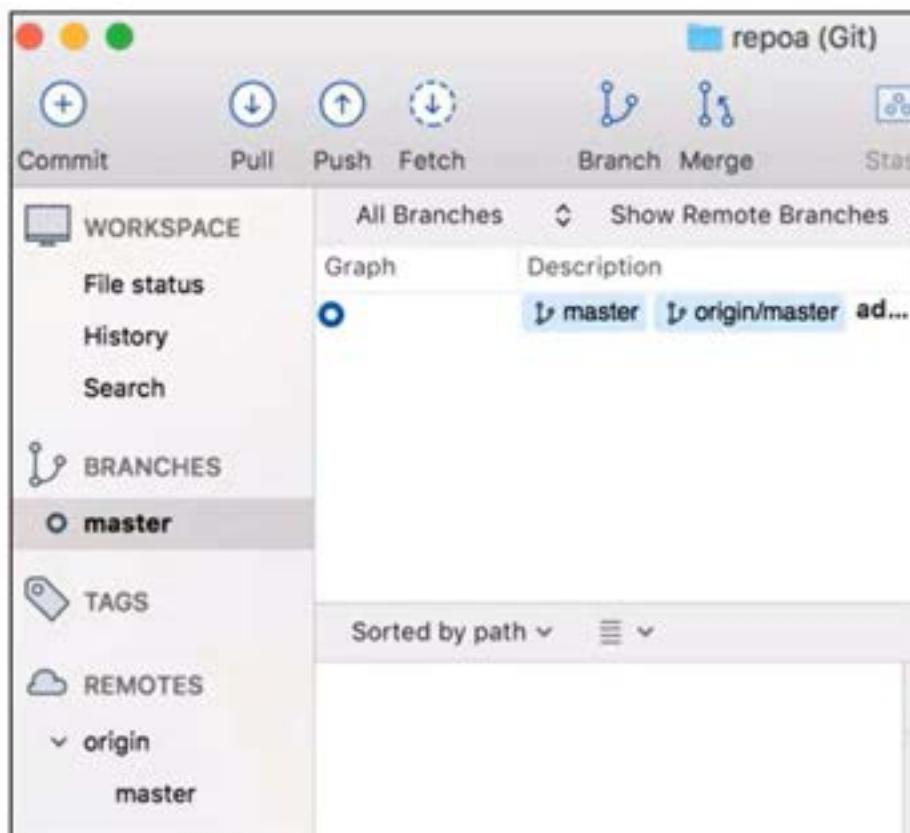


VIEWING BRANCHES



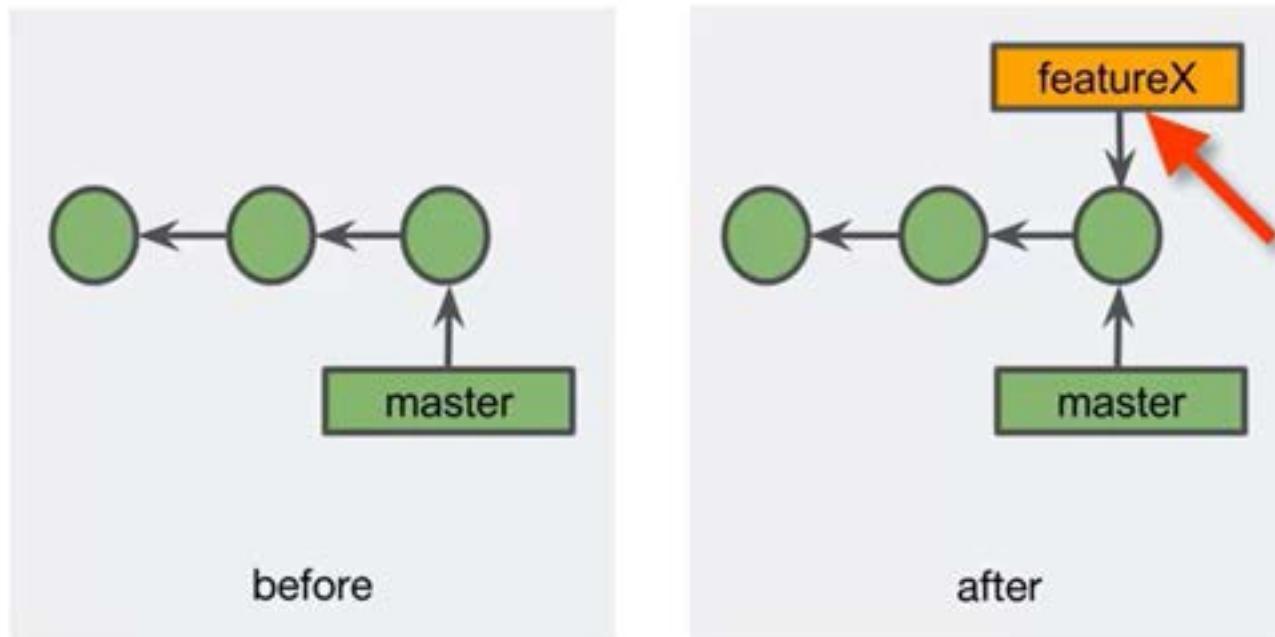
Using a command line, `git branch -a` can be used to see local and remote tracking branches

VIEWING BRANCHES

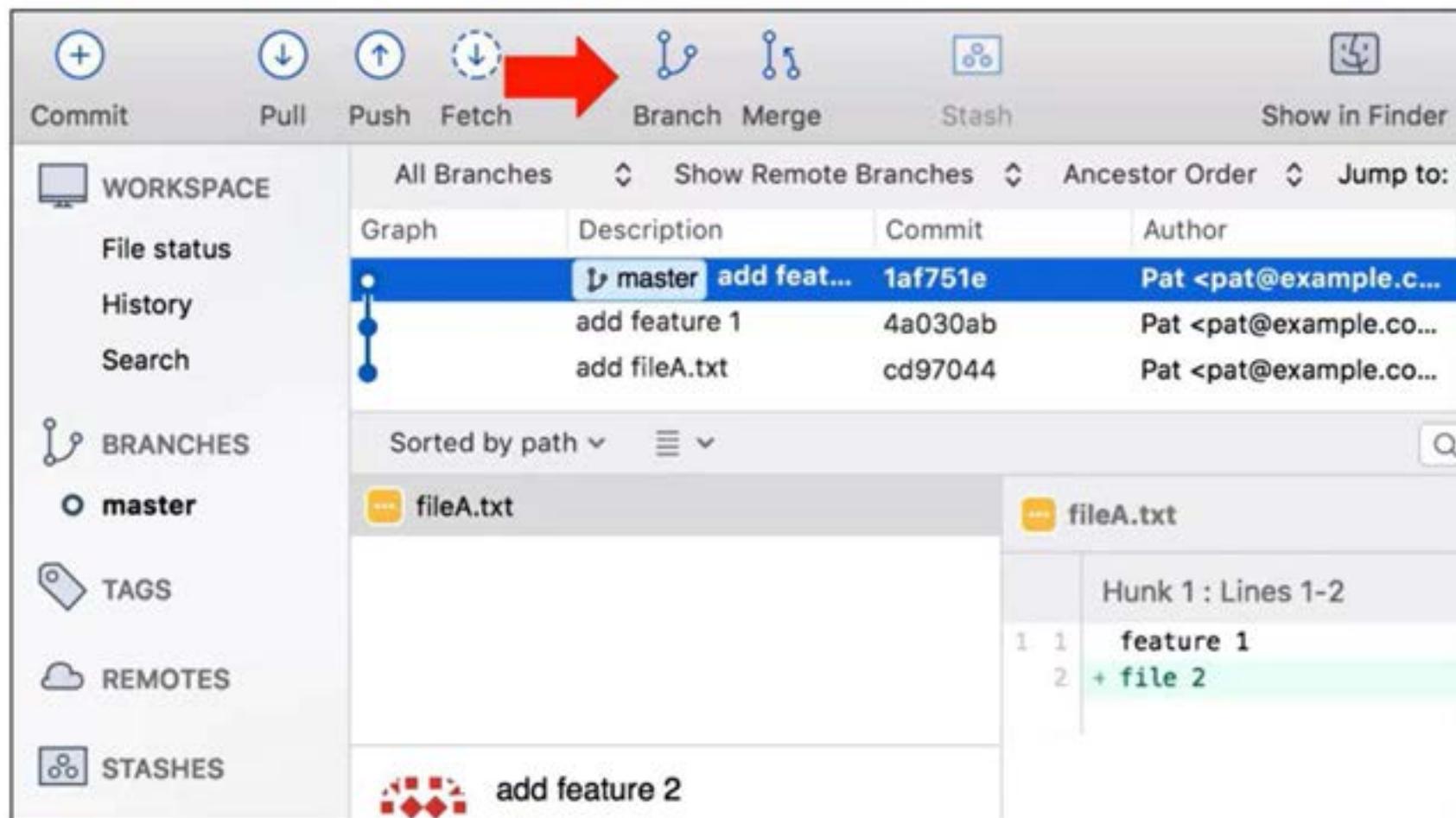


Using a command line, `git branch -a` can be used to see local and remote tracking branches

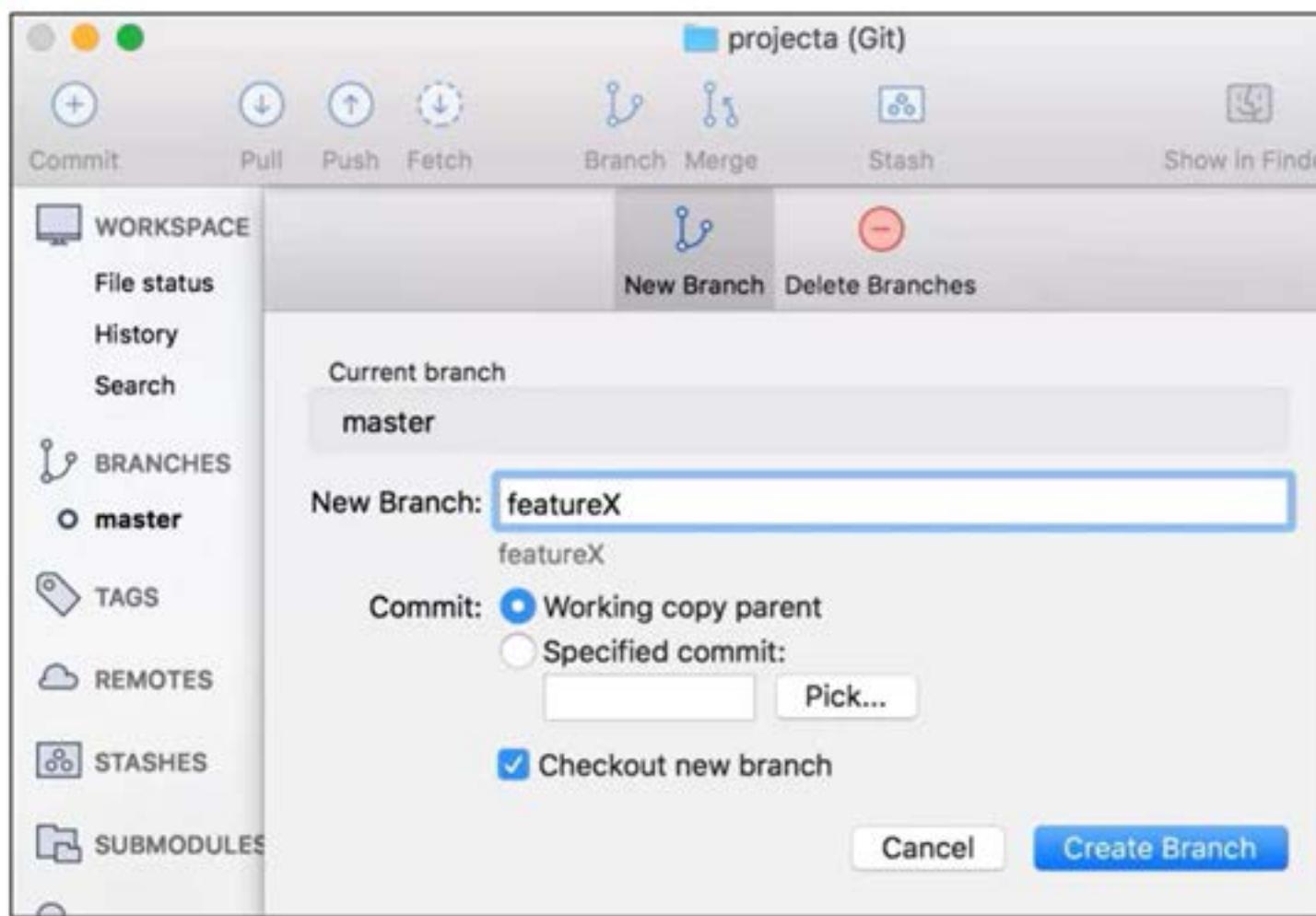
CREATING A BRANCH



CREATING A BRANCH

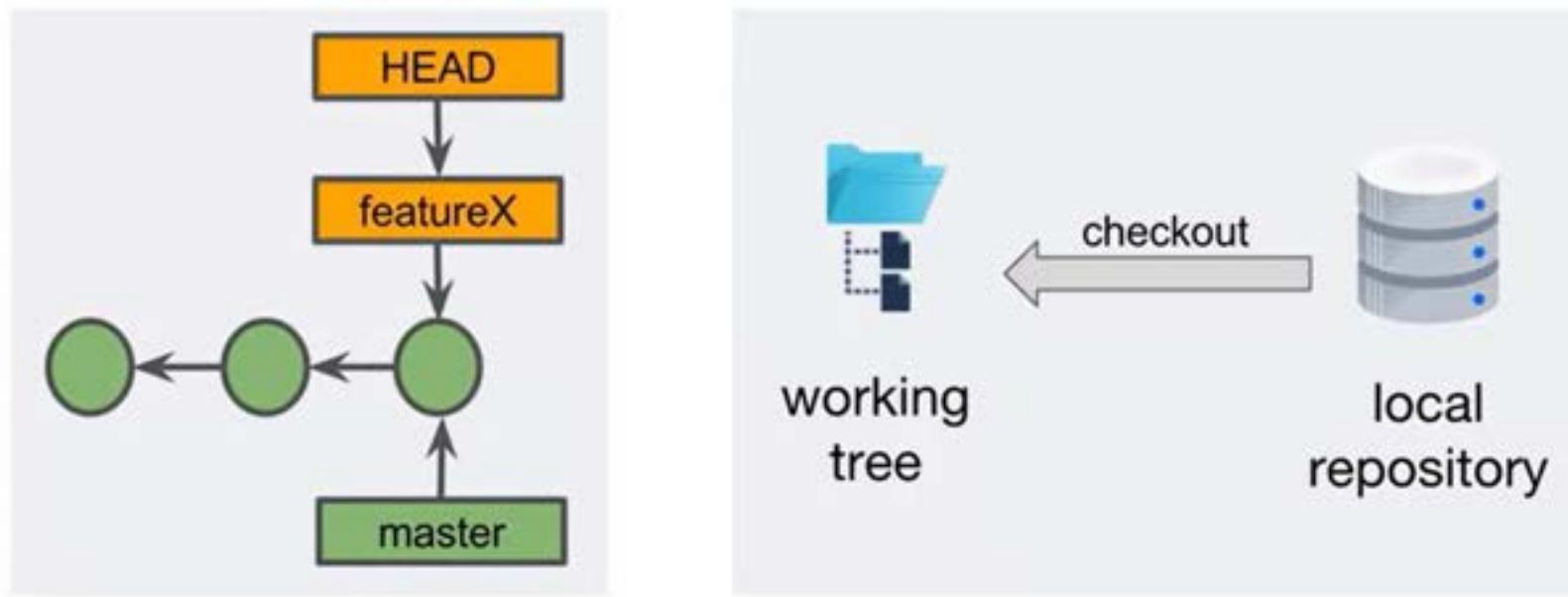


CREATING A BRANCH

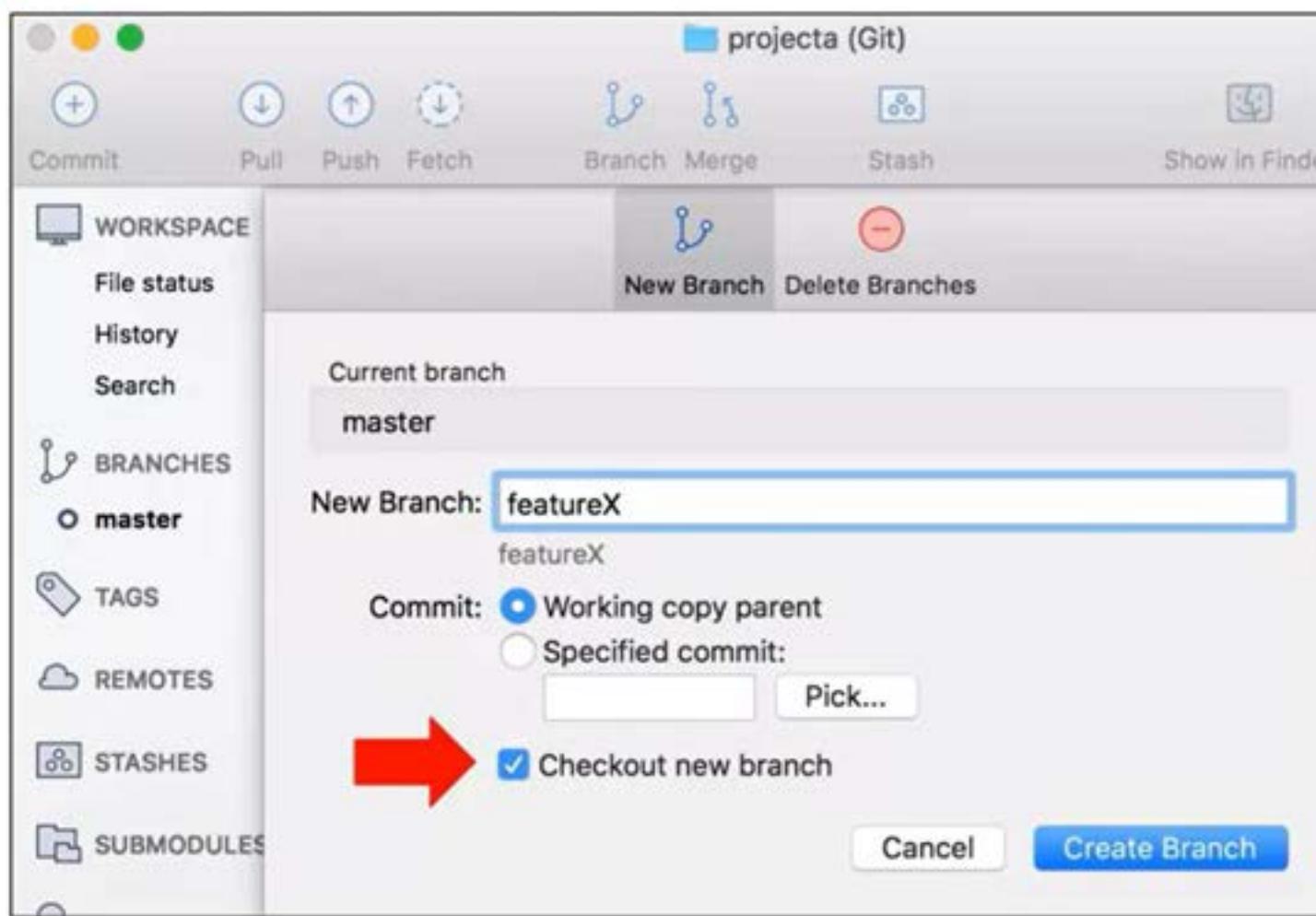


CHECKOUT

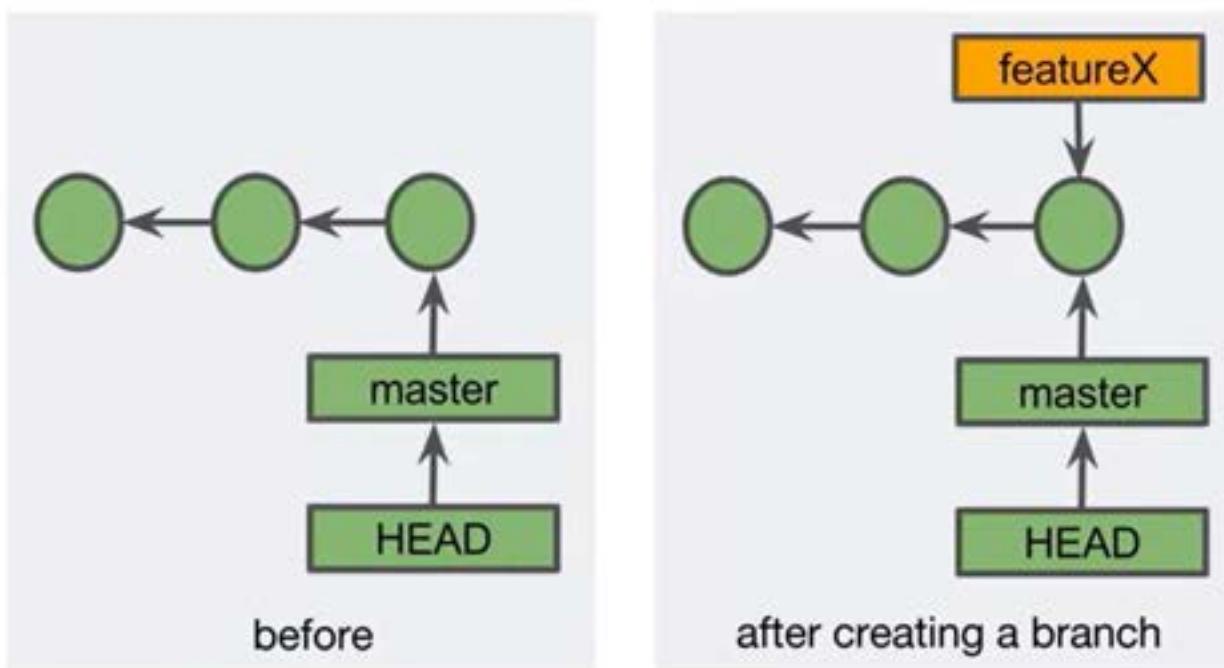
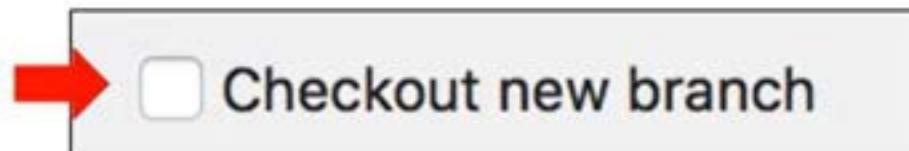
1. Updates the HEAD reference
2. Updates the working tree with the commit's files



CHECKOUT NEW BRANCH



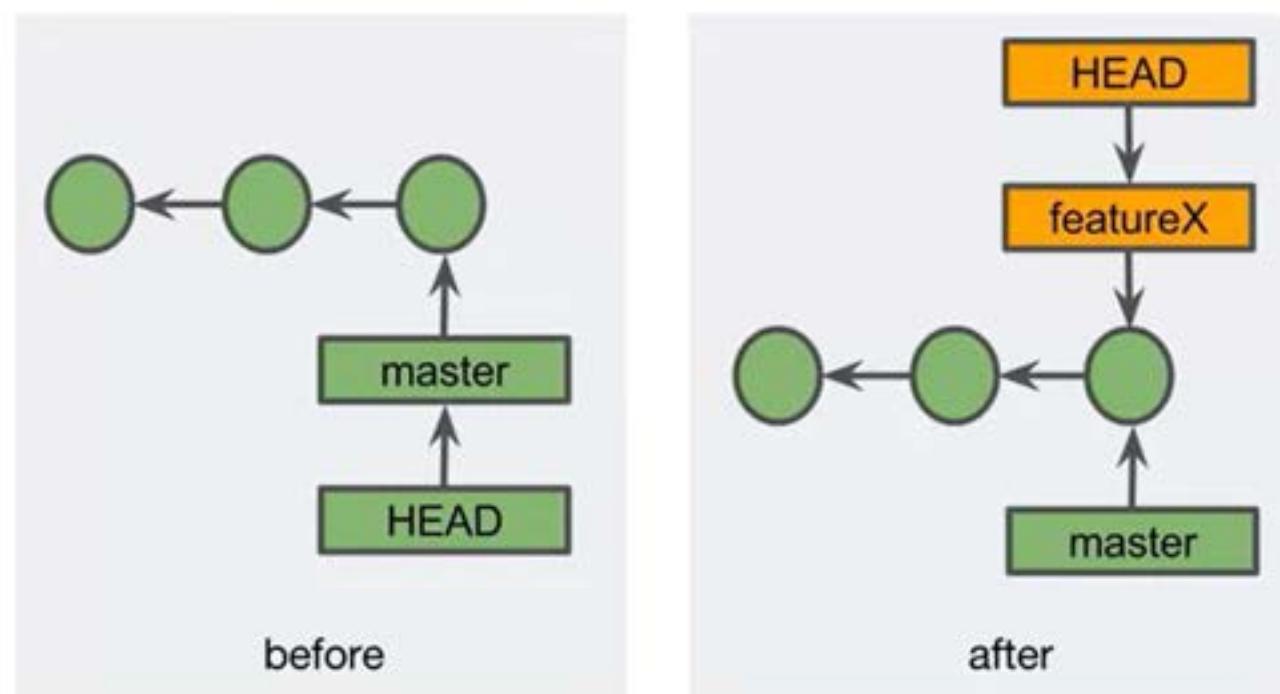
CREATING A BRANCH WITHOUT CHECKING IT OUT



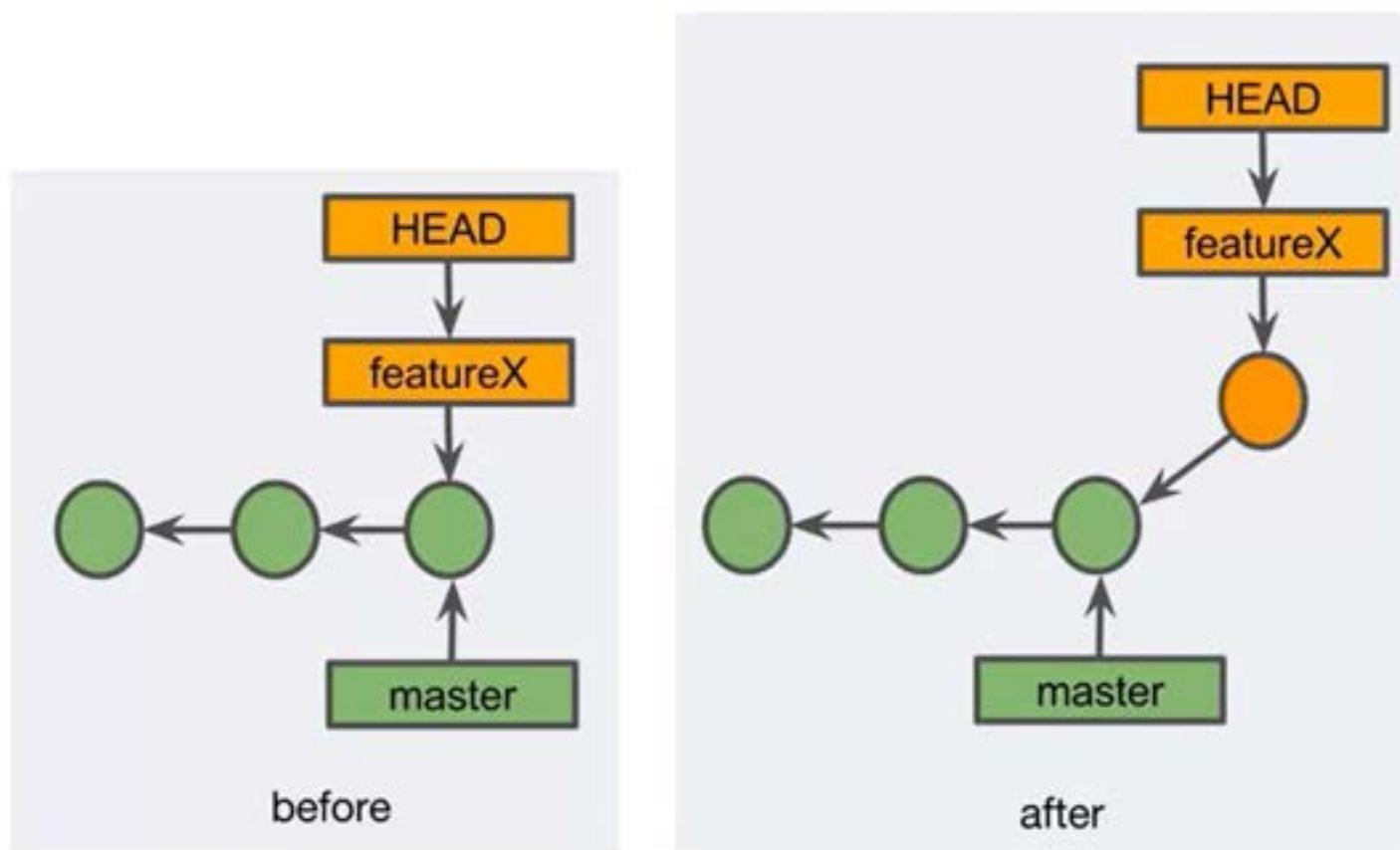
At the command line, `git branch <name>` is used to create a branch

CREATING AND CHECKING OUT A BRANCH

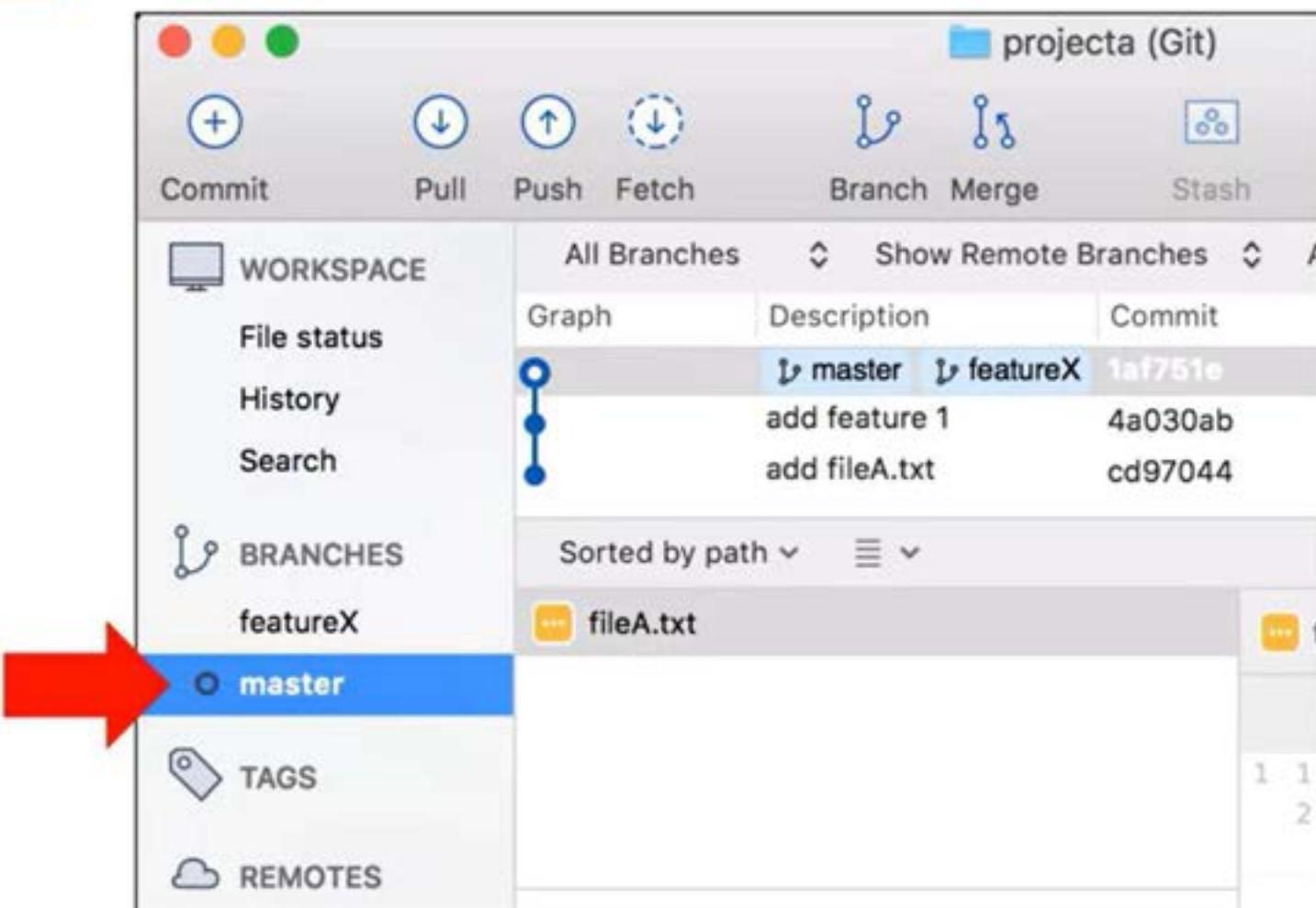
➡ Checkout new branch



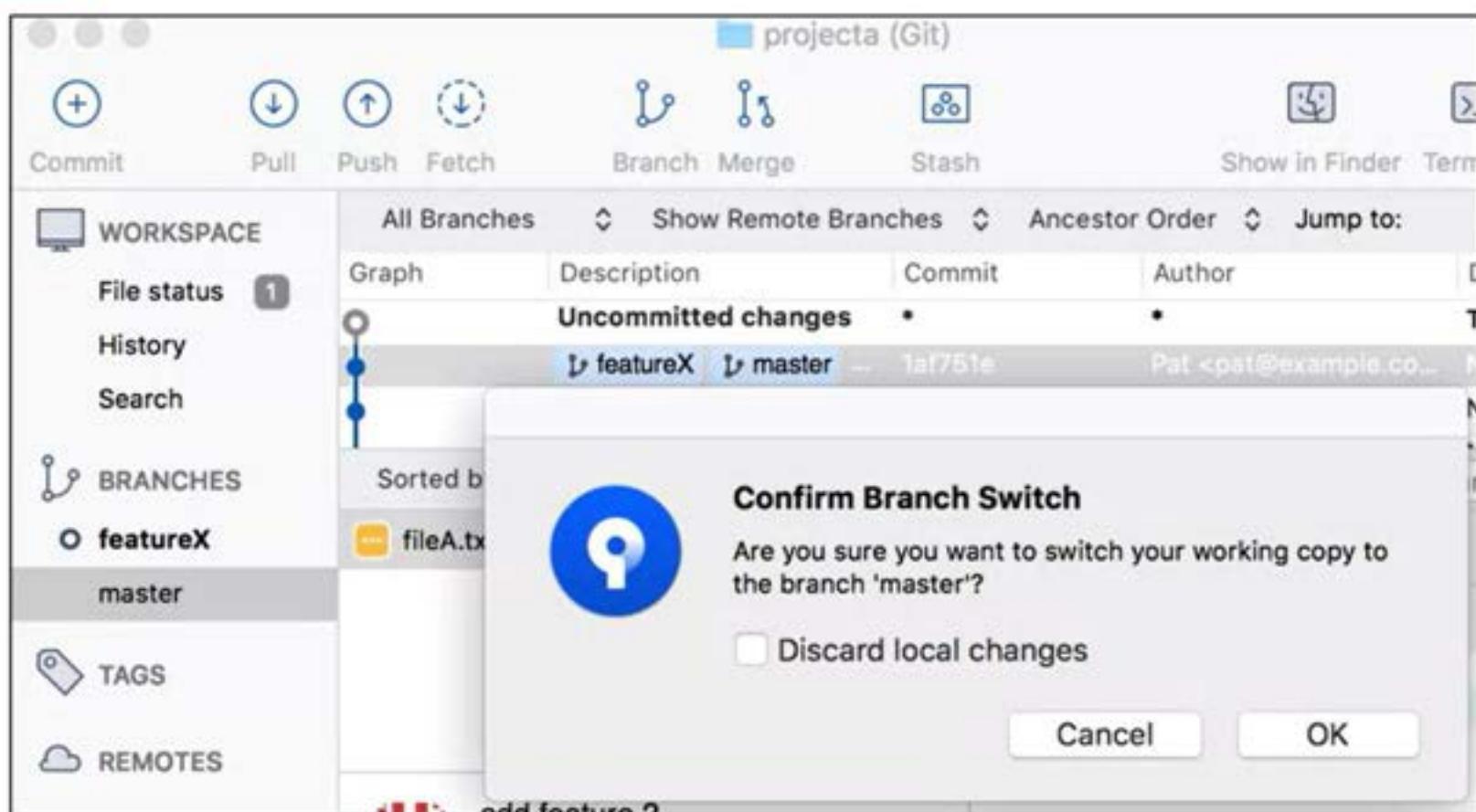
THE NEXT COMMIT ON A BRANCH



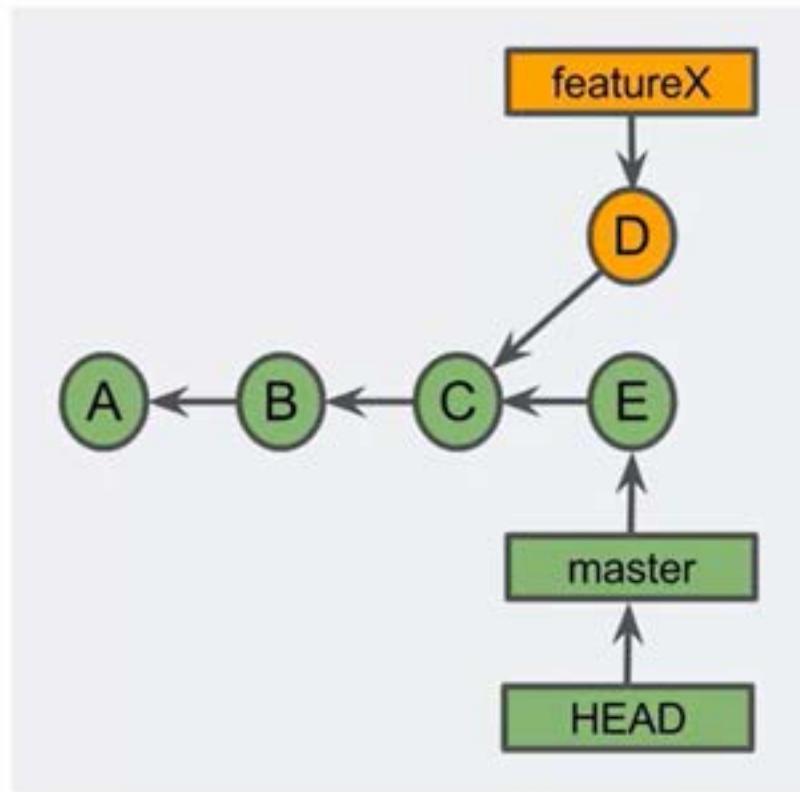
SWITCHING BRANCHES (CHECKOUT)



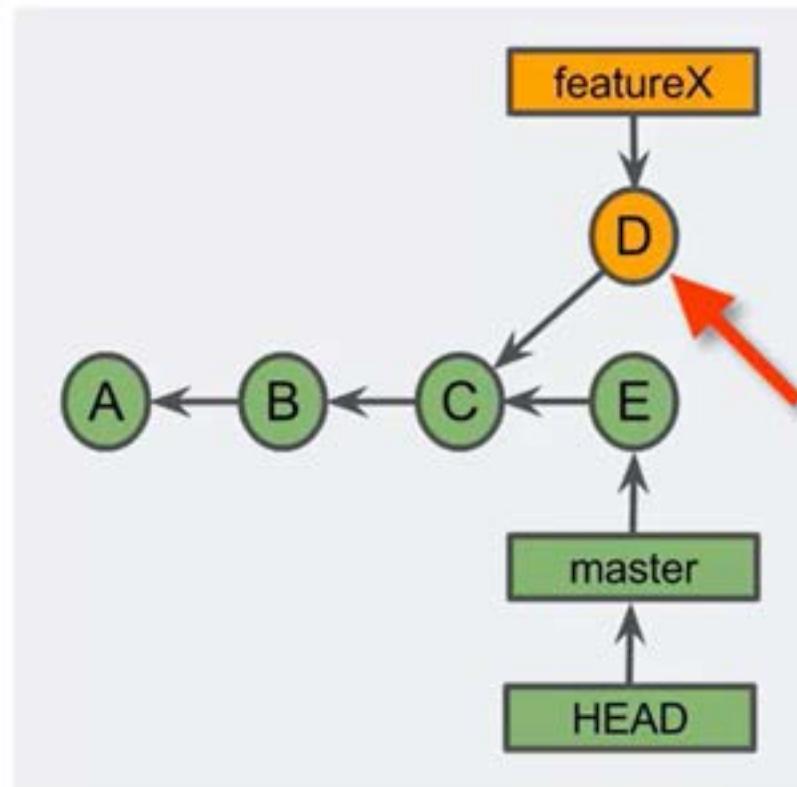
STASHING UNCOMMITTED CHANGES



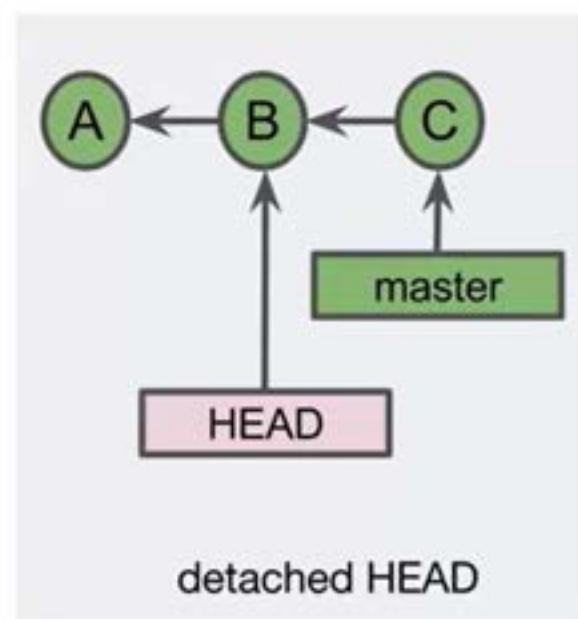
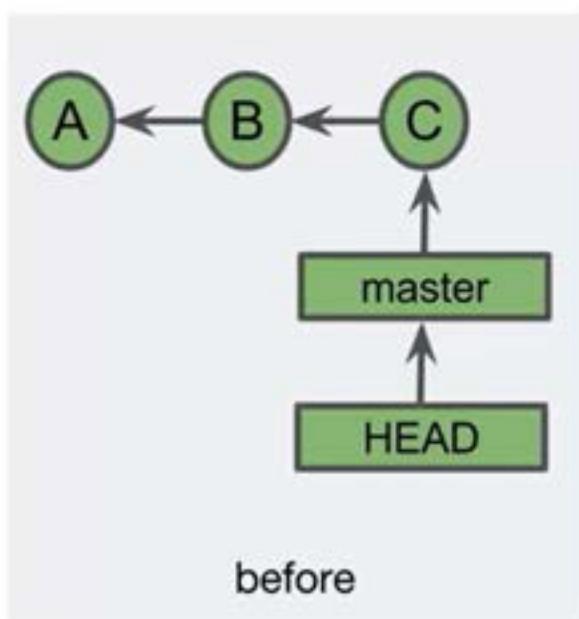
THE NEXT COMMIT ON MASTER



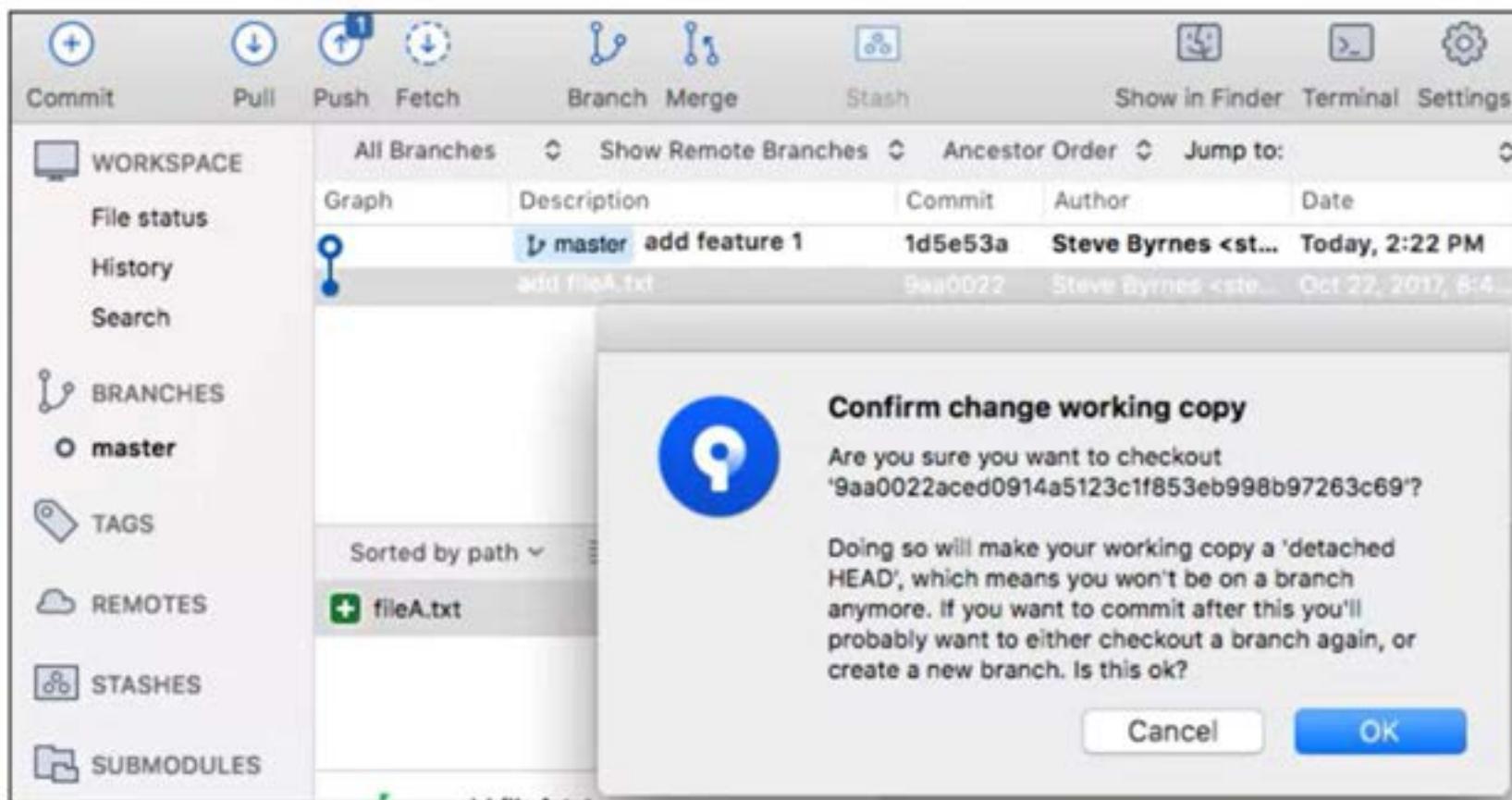
THE NEXT COMMIT ON MASTER



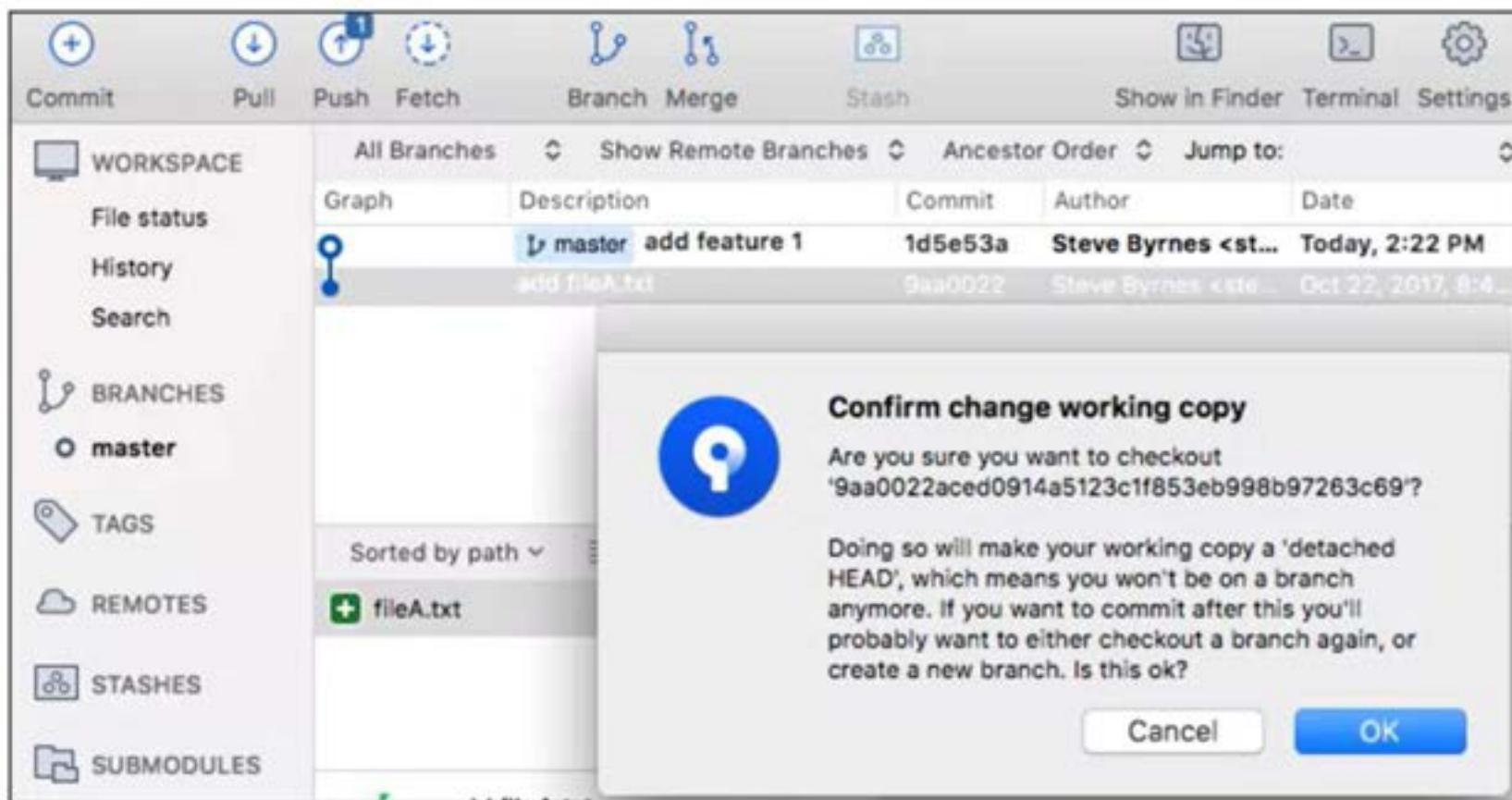
DETACHED HEAD



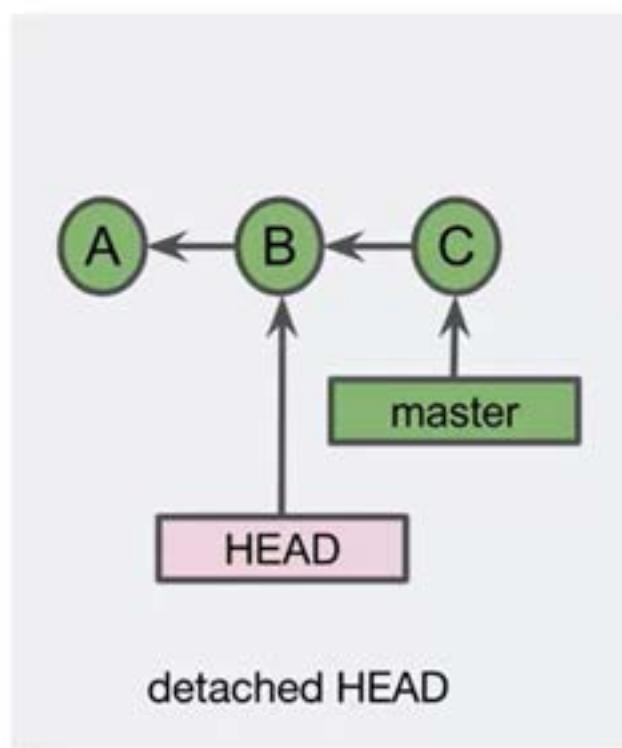
DETACHED HEAD WARNING



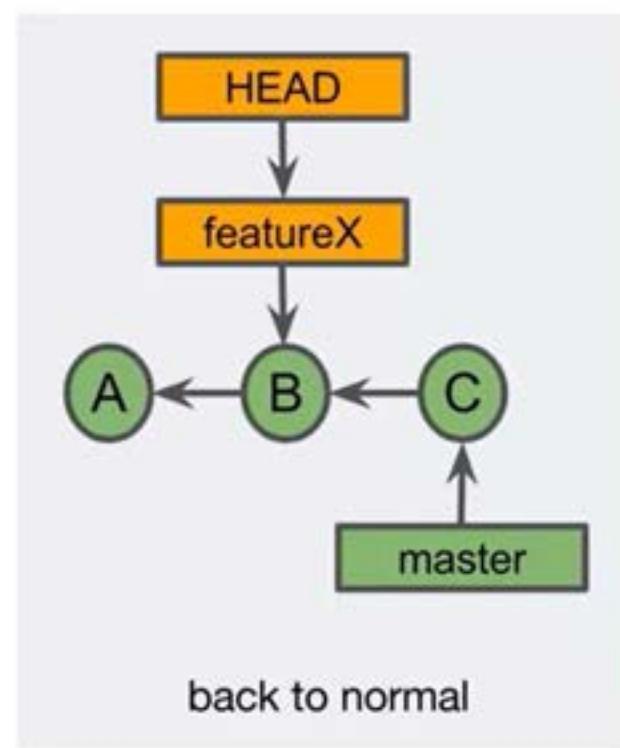
DETACHED HEAD WARNING



FIXING A DETACHED HEAD

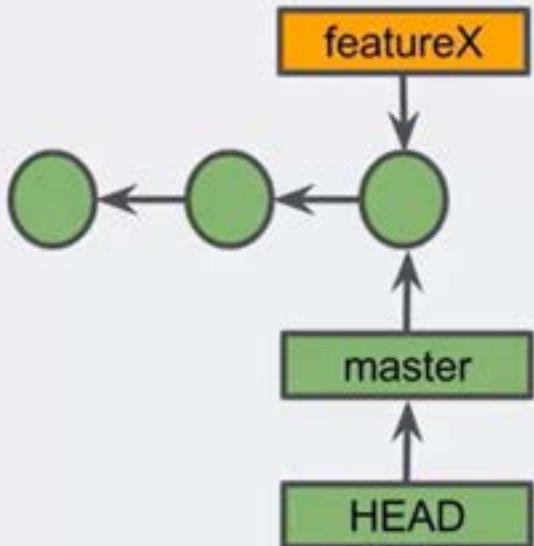


detached HEAD

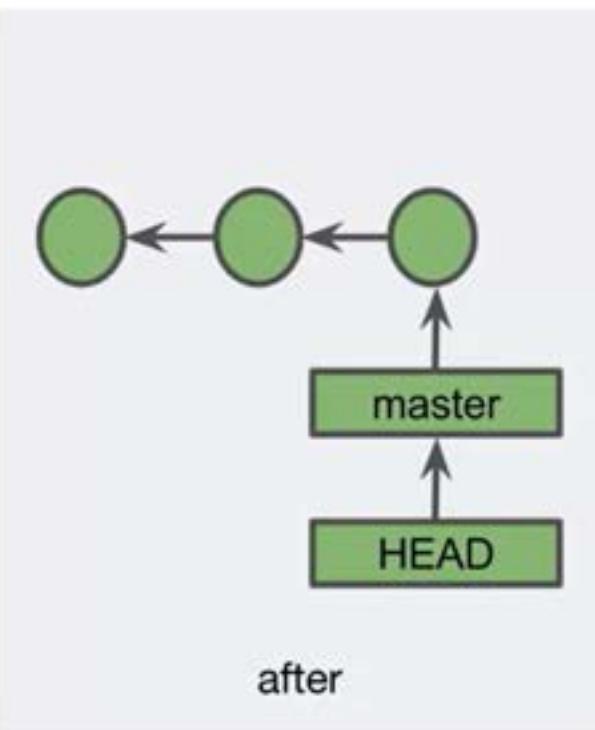


back to normal

DELETING A BRANCH LABEL

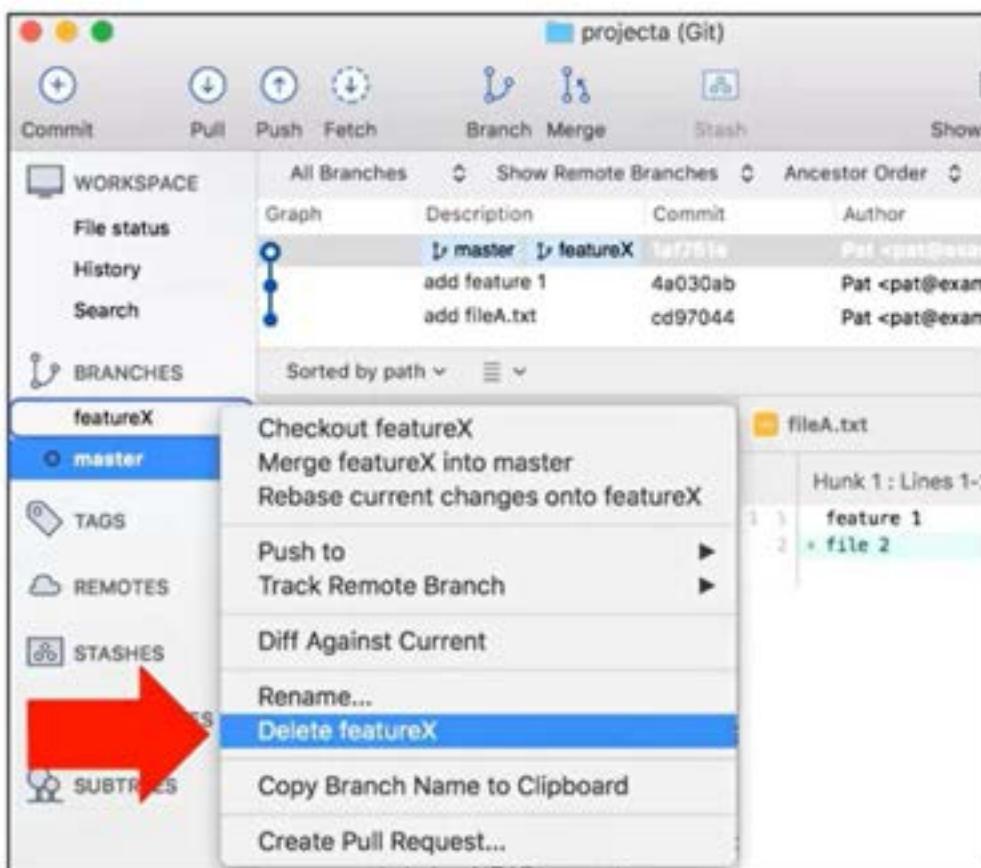


before



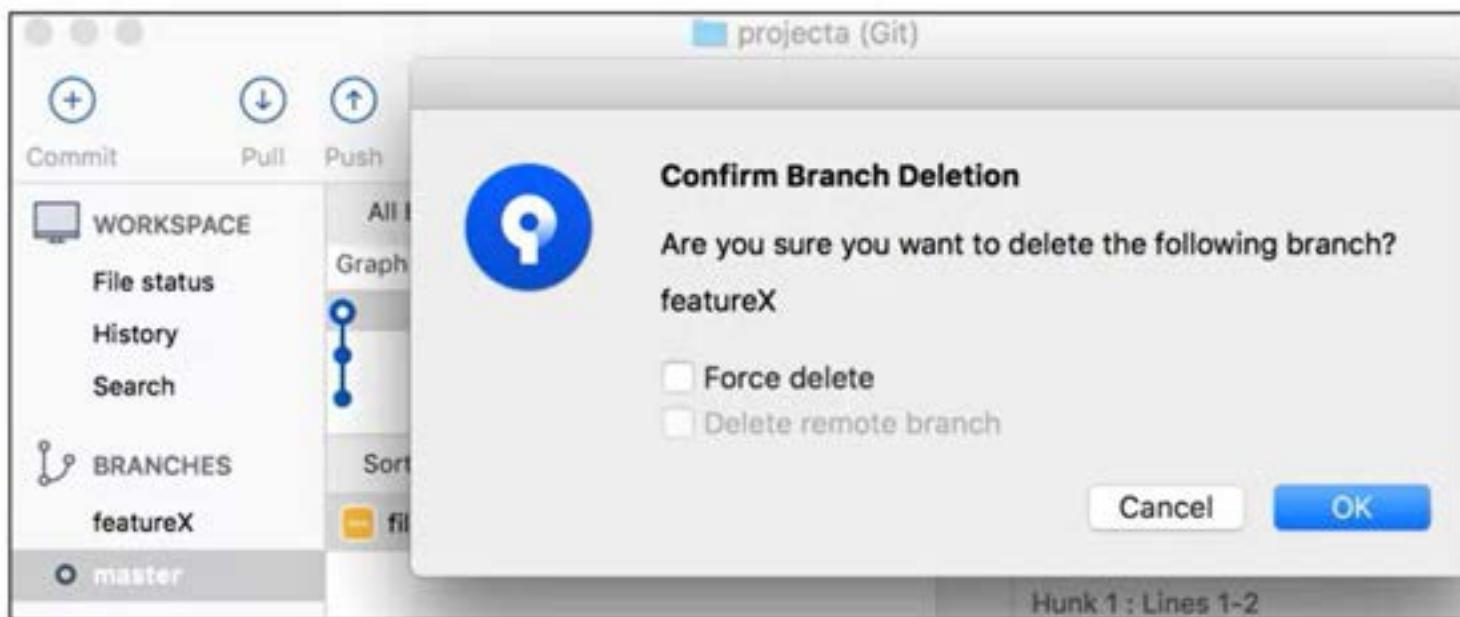
after

DELETING A BRANCH WITH SOURCETREE

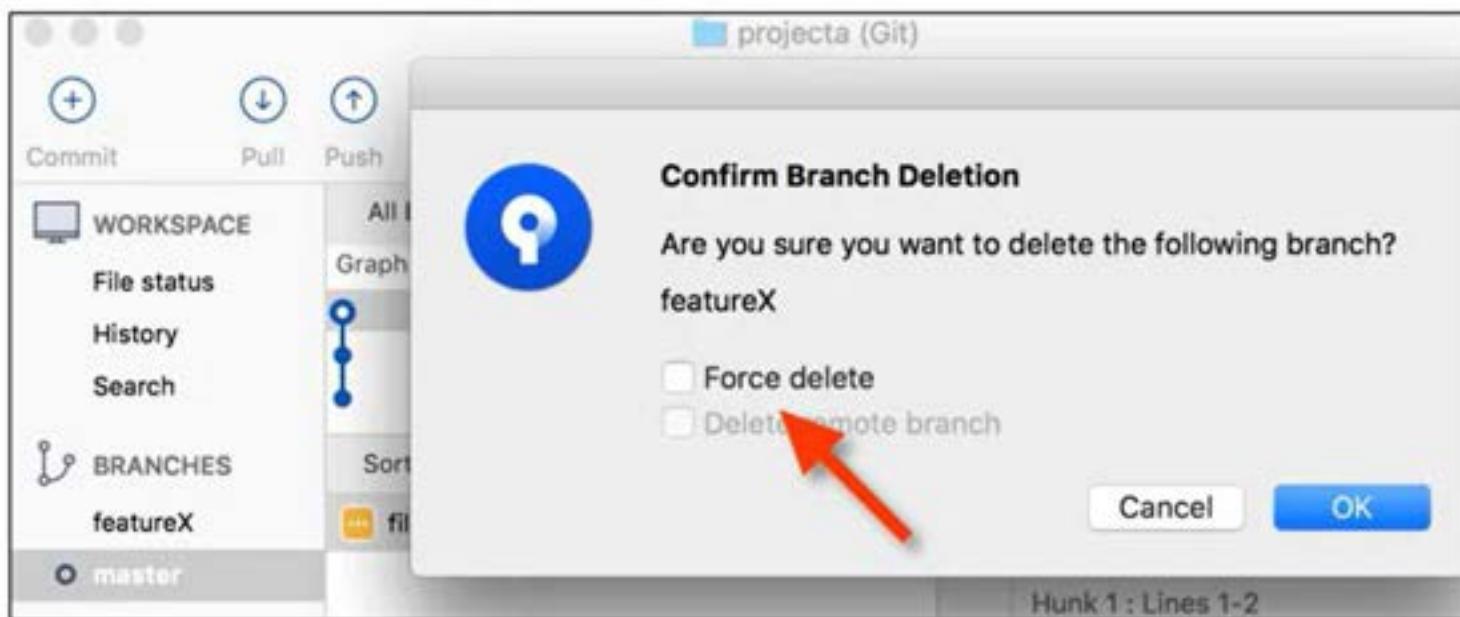


At the command line, `git branch -d <name>` is used to delete a branch label

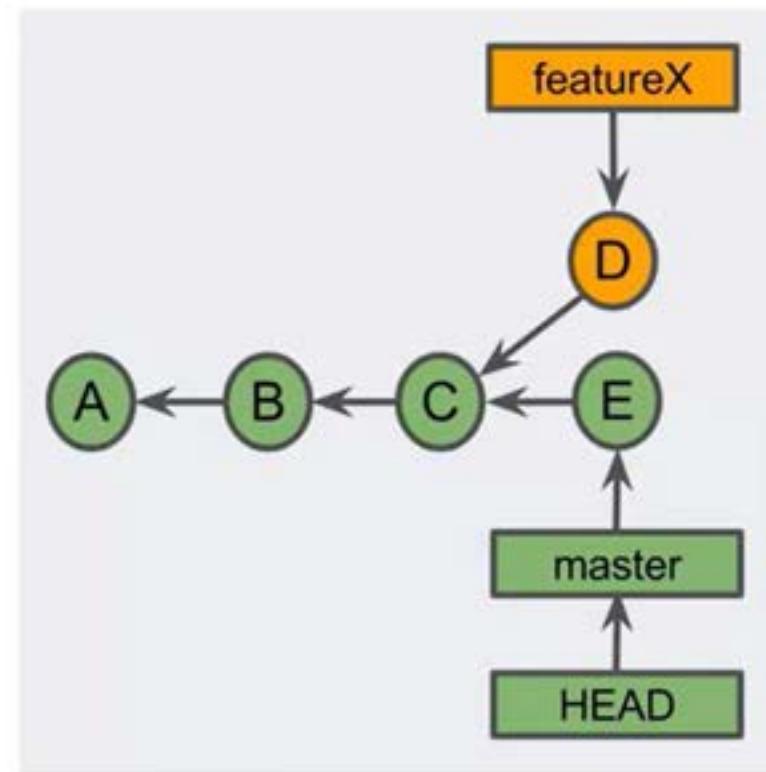
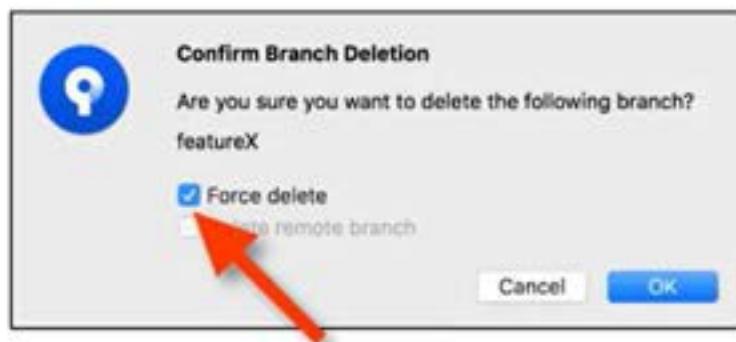
DELETING A BRANCH WITH SOURCETREE



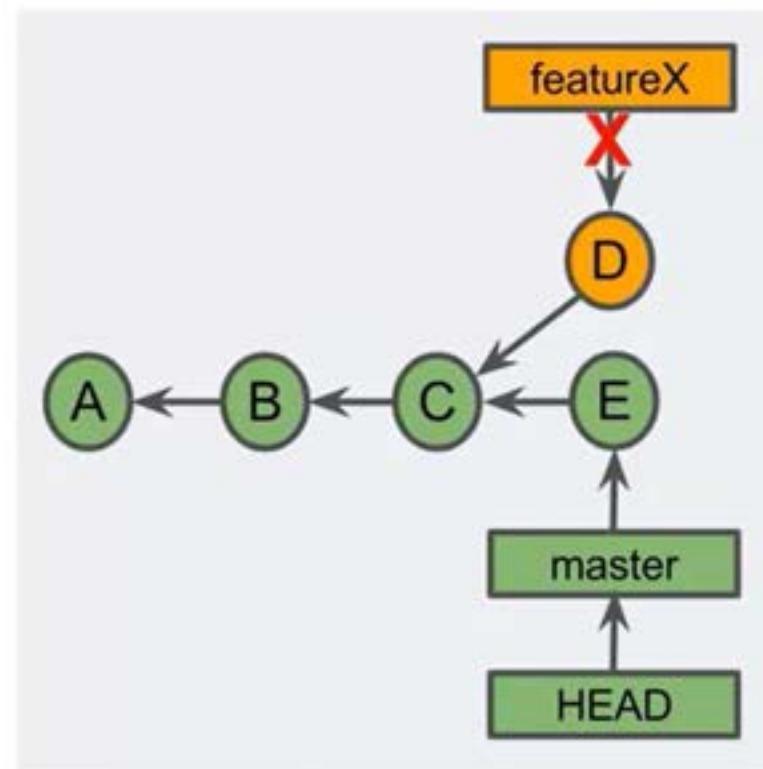
DELETING A BRANCH WITH SOURCETREE



DANGLING COMMITS



DANGLING COMMITS

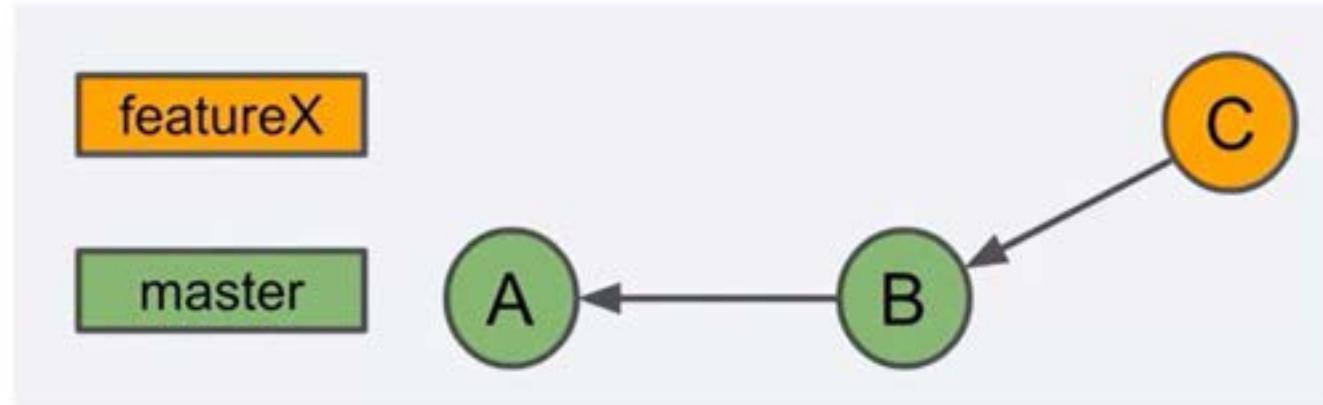


REVIEW

- A branch is a set of commits that trace back to the project's first commit
- Creating a branch creates a branch label
- Checkout involves updating HEAD and updating the working tree
- A detached HEAD reference points directly to a commit
- Fix a detached HEAD by creating a branch
- Deleting a branch deletes a branch label
- Dangling commits will eventually be garbage collected



WHAT IS A BRANCH?



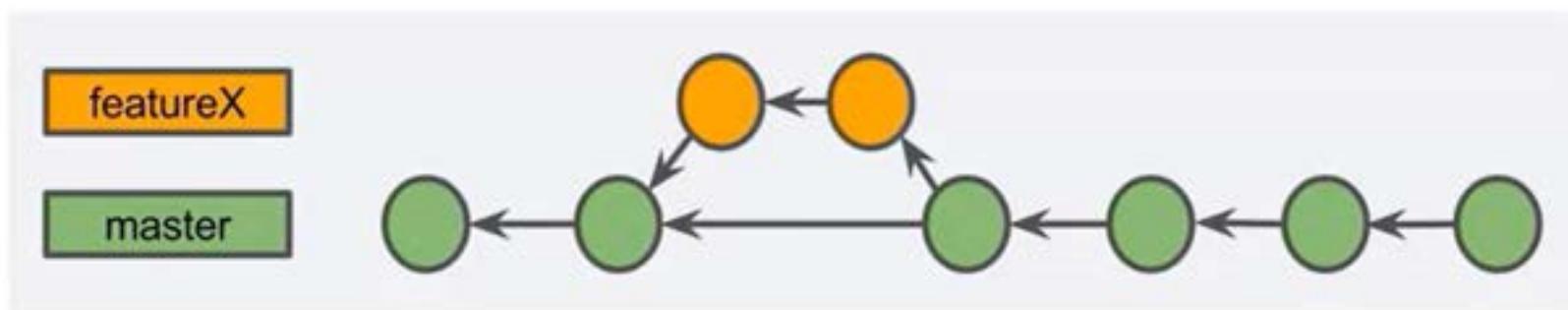
The set of commits that trace back to the project's first commit

BENEFITS OF BRANCHES

- Fast and easy to create
- Enable experimentation
- Enable team development
- Support multiple project versions

TOPIC AND LONG-RUNNING BRANCHES

- Topic
 - A feature, a bug fix, a hotfix, a configuration change, etc.
- Long-lived
 - **master, develop, release**, etc.



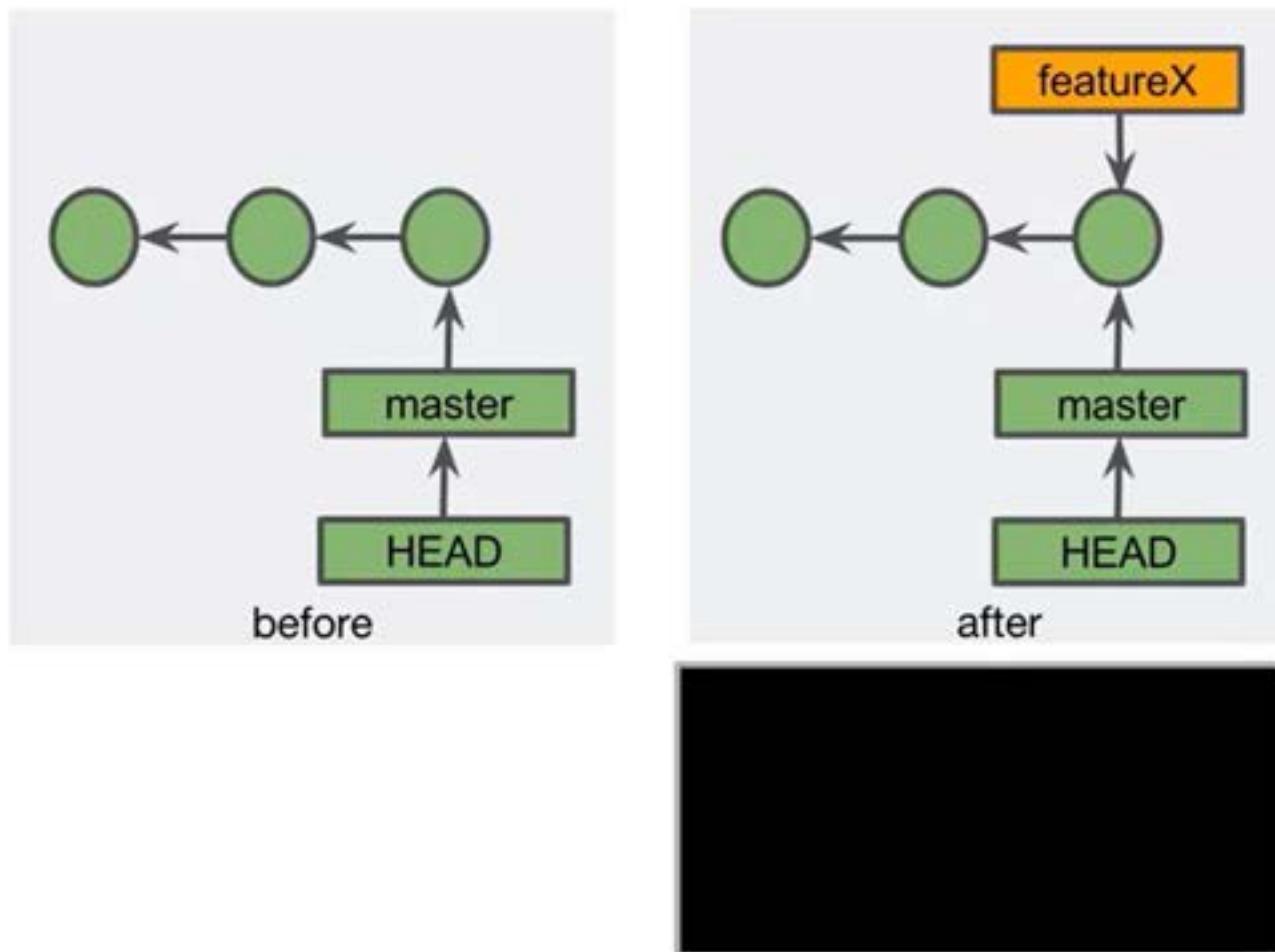
VIEWING BRANCHES

Use `git branch` to see a list of branches

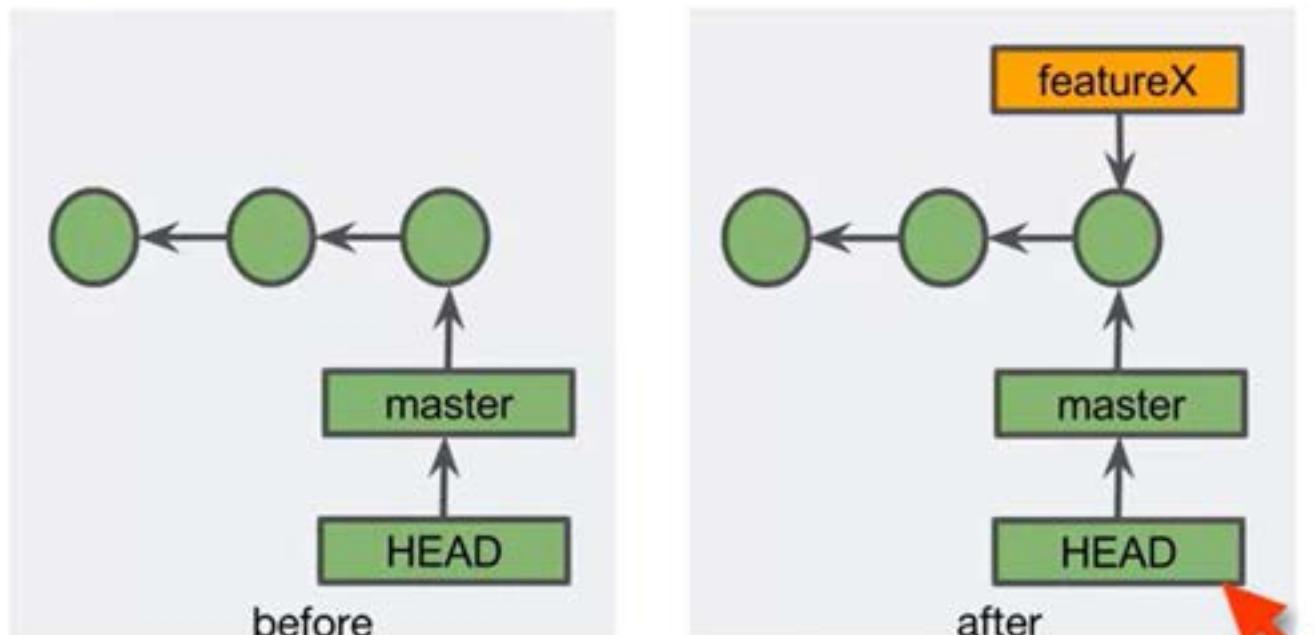
```
$ git branch  
  featureX  
* master
```

We will see later that `git branch -a` can be used to see local and remote tracking branches

CREATING A BRANCH USING `git branch <name>`



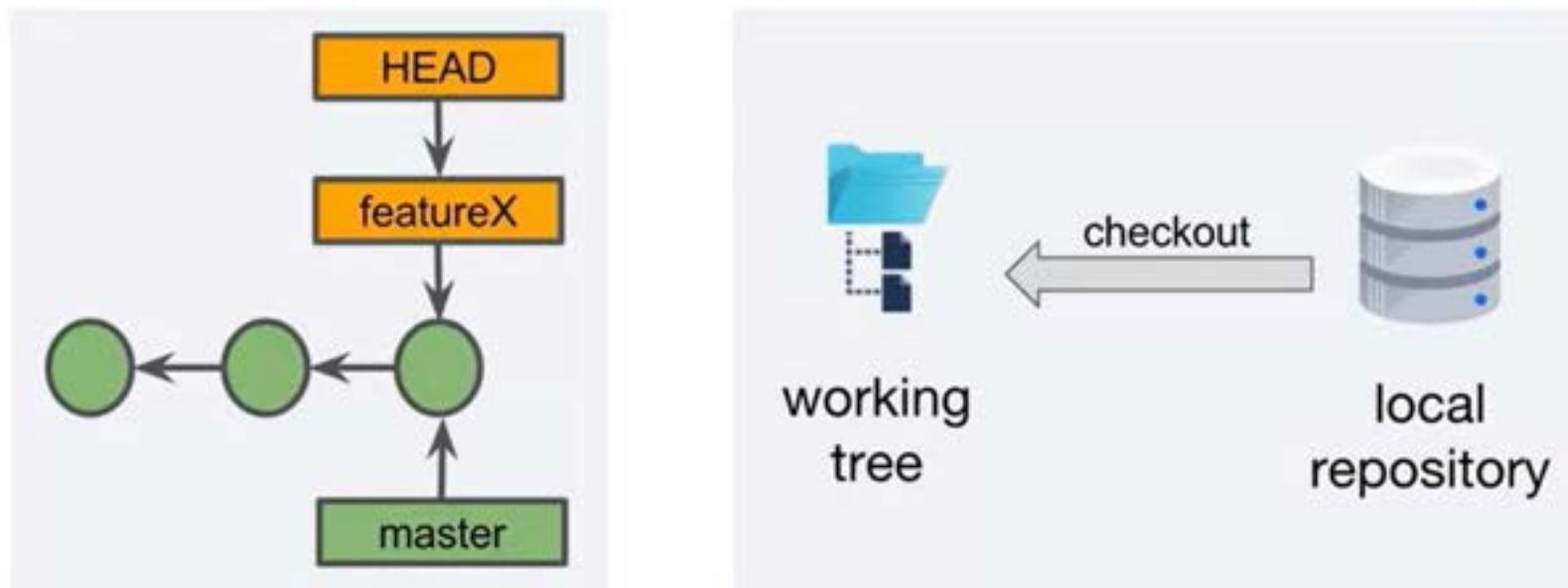
CREATING A BRANCH USING `git branch <name>`



```
# create a branch
$ git branch featureX
$ git branch
* master
  featureX
```

CHECKOUT

1. Updates the HEAD reference
2. Updates the working tree with the commit's files

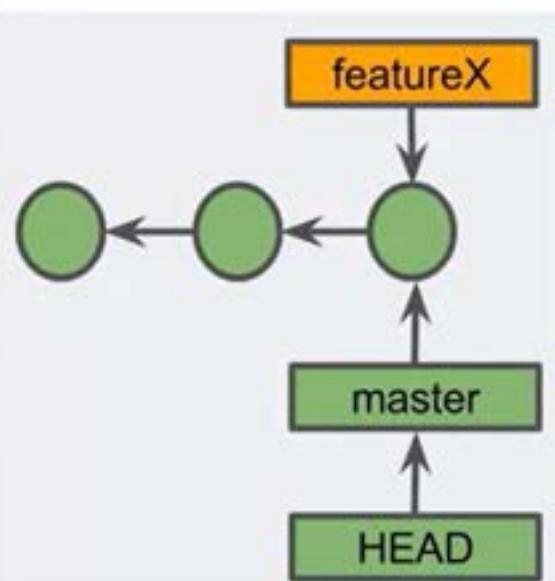


CHECKOUT

Use `git checkout <branch_or_commit>` to checkout a branch or commit

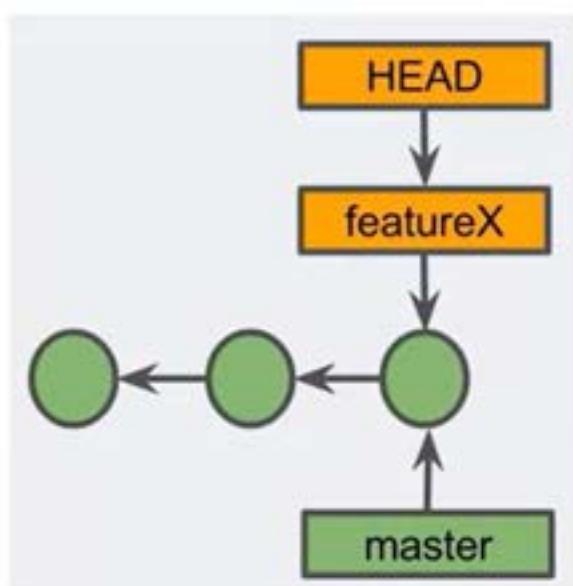
```
# switch to a branch
$ git checkout featureX
$ git branch
  master
* featureX
$ ls
(files for featureX)
```

CREATING AND CHECKING OUT A BRANCH (TWO COMMANDS)



```
# create a branch
$ git branch featureX
$ git branch
* master
  featureX
```

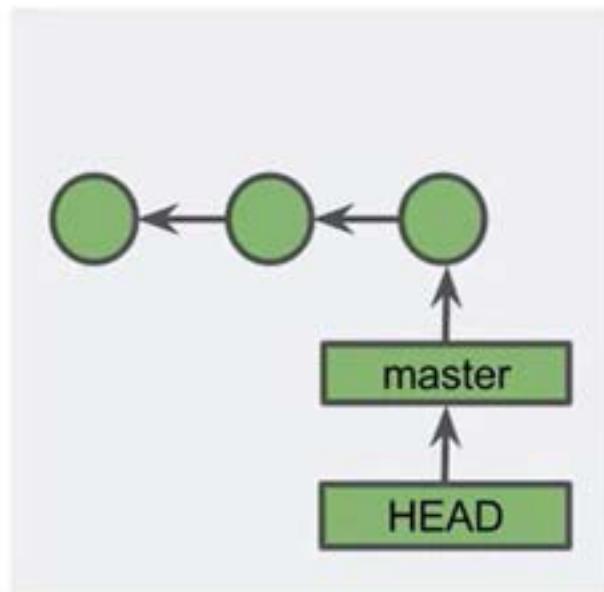
after creating a branch



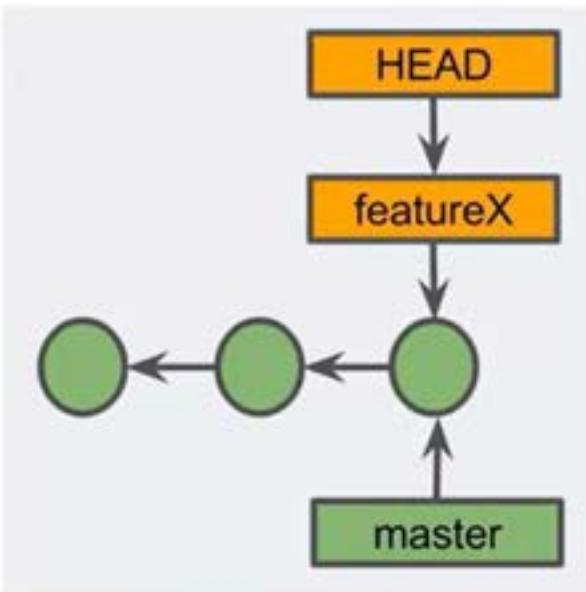
```
# switch to a branch
$ git checkout featureX
$ git branch
  master
* featureX
```

after creating and checking
out a branch

CREATING AND CHECKING OUT A BRANCH (SINGLE COMMAND)



before

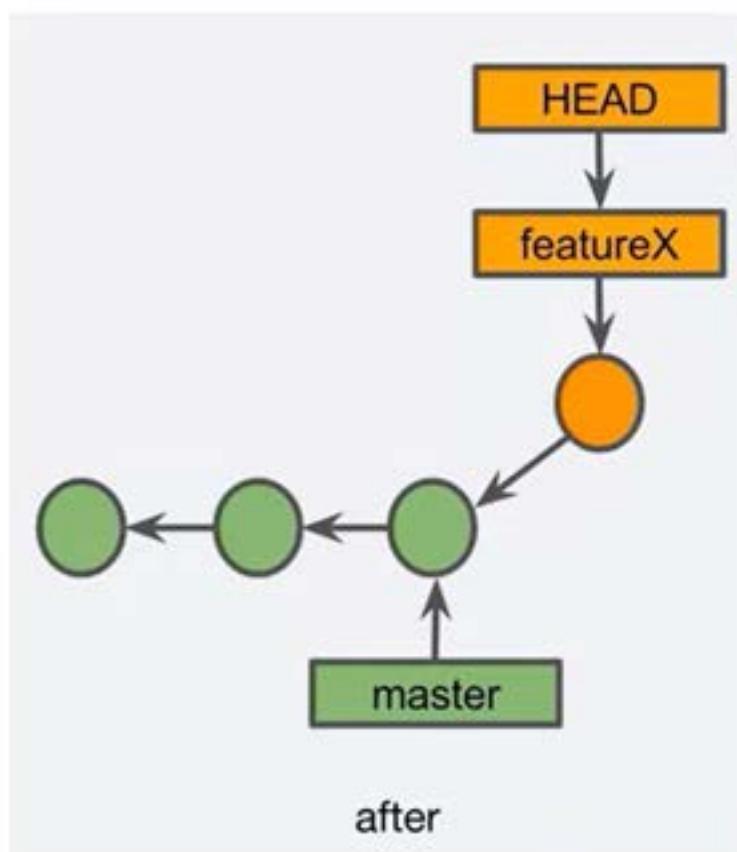
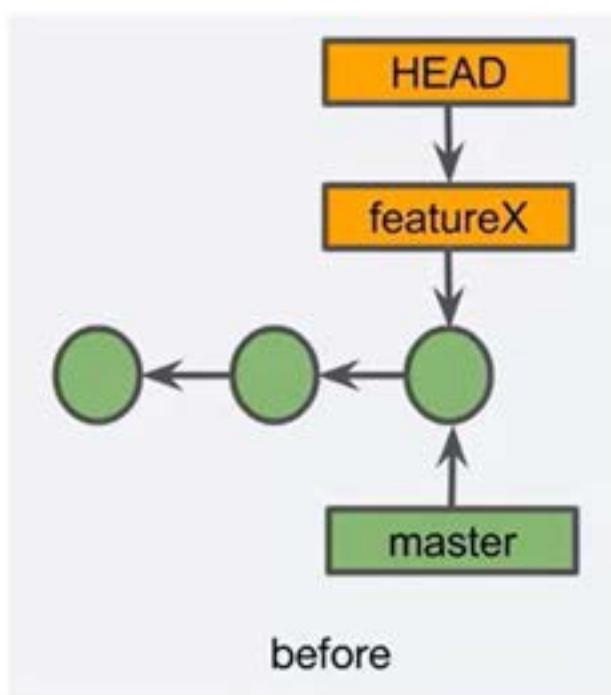


after

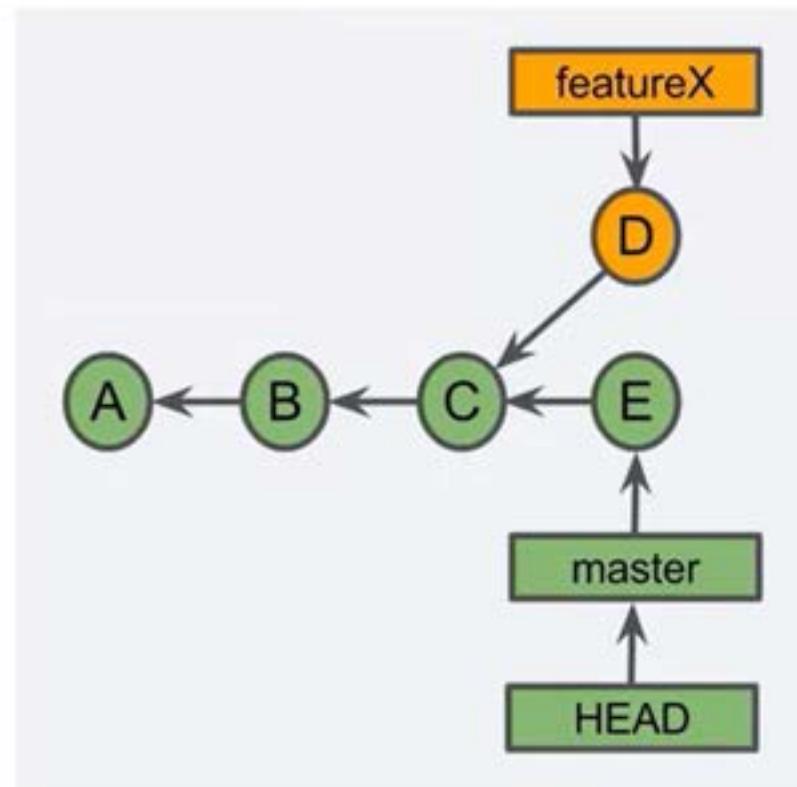
The `-b` option
combines two
commands (`git
branch` and `git
checkout`)

```
$ git checkout -b featureX
```

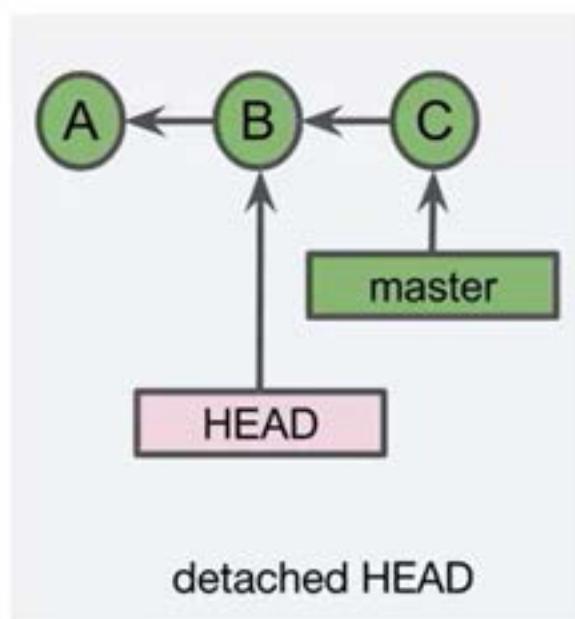
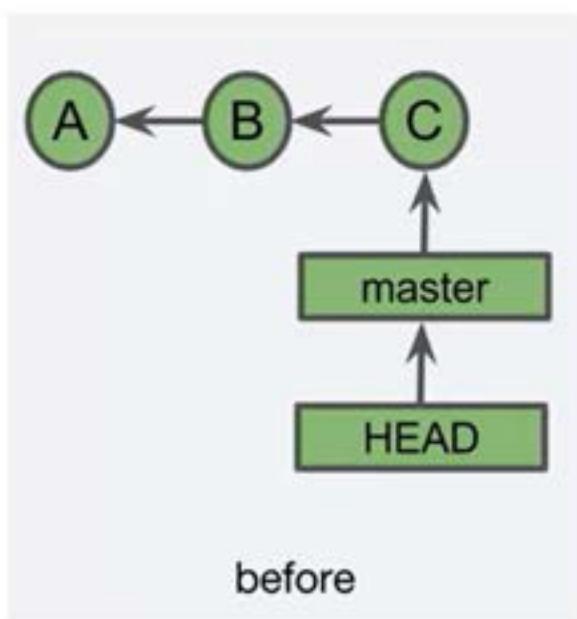
THE NEXT COMMIT ON A BRANCH



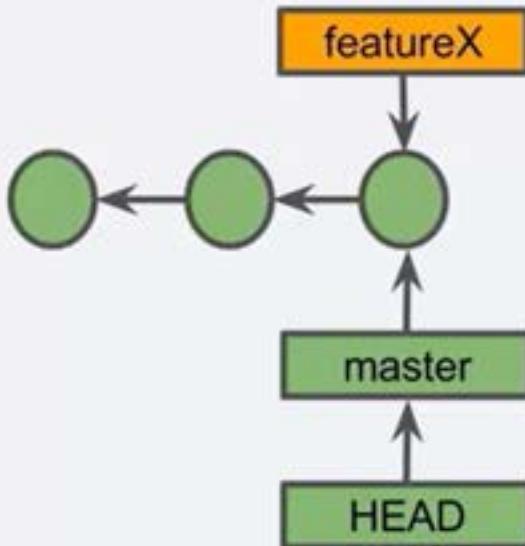
THE NEXT COMMIT ON MASTER



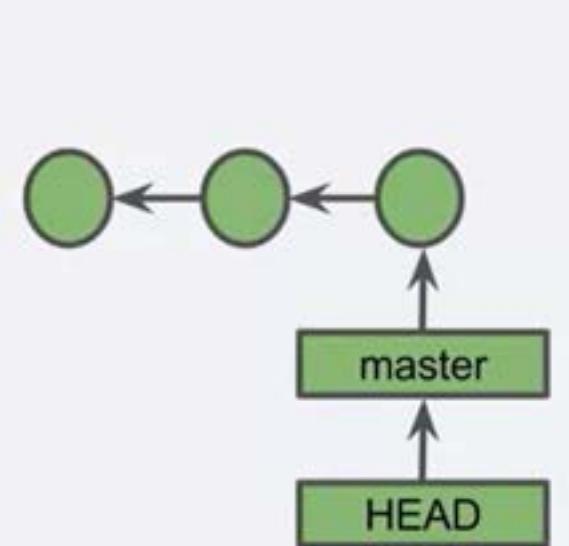
DETACHED HEAD



DELETING A BRANCH LABEL

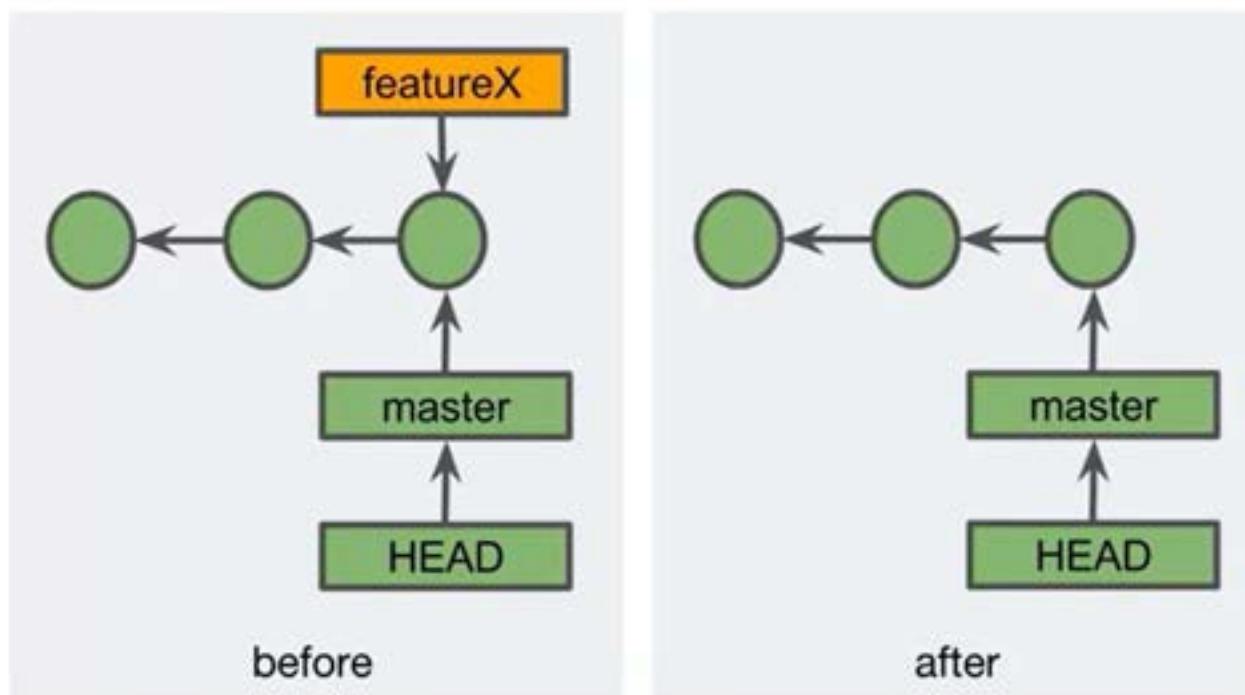


before



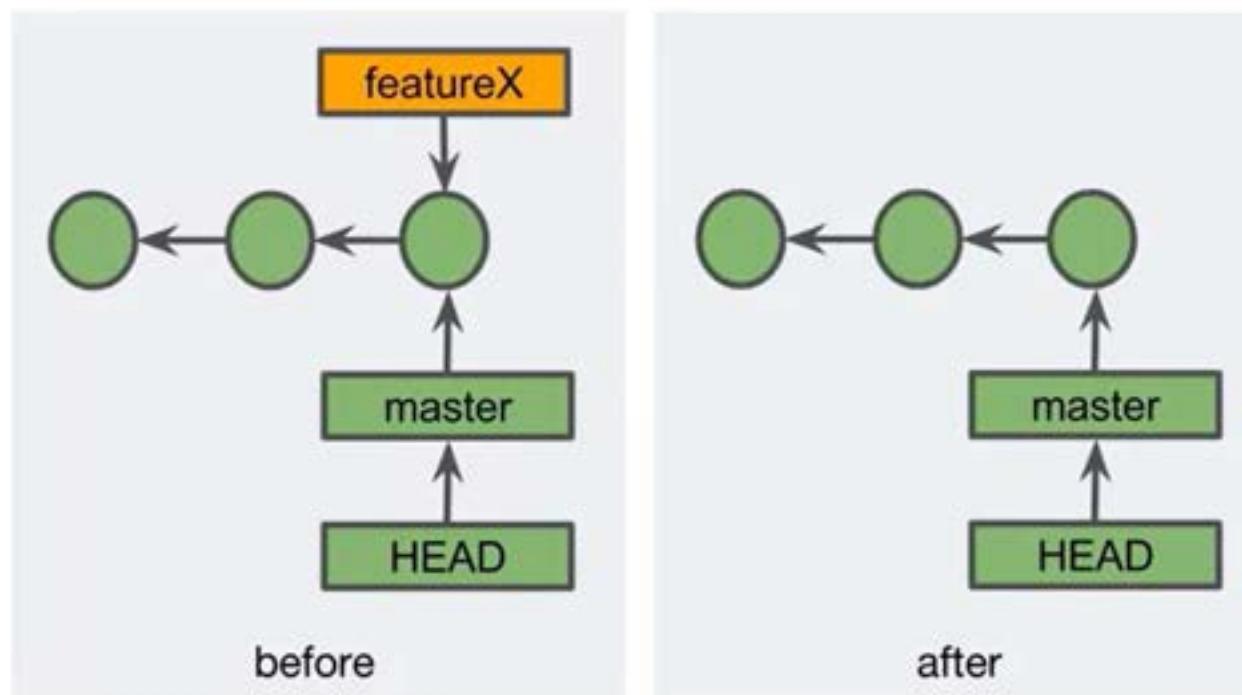
after

DELETING A BRANCH LABEL WITH `git branch -d <branch>`



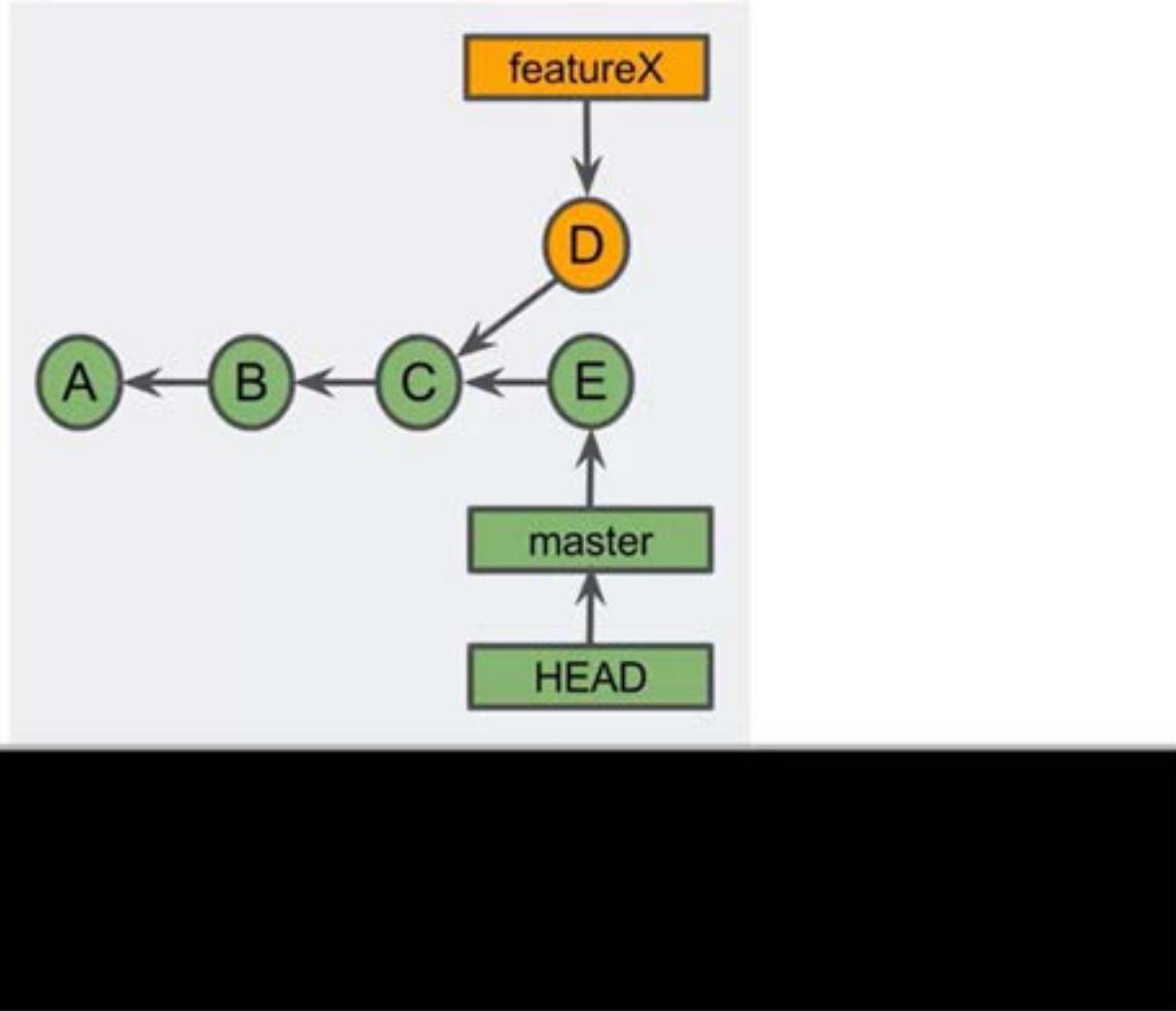
```
$ git branch -d featureX
Deleted branch featureX (was 010226b)
$ git branch
* master
```

DELETING A BRANCH LABEL WITH `git branch -d <branch>`

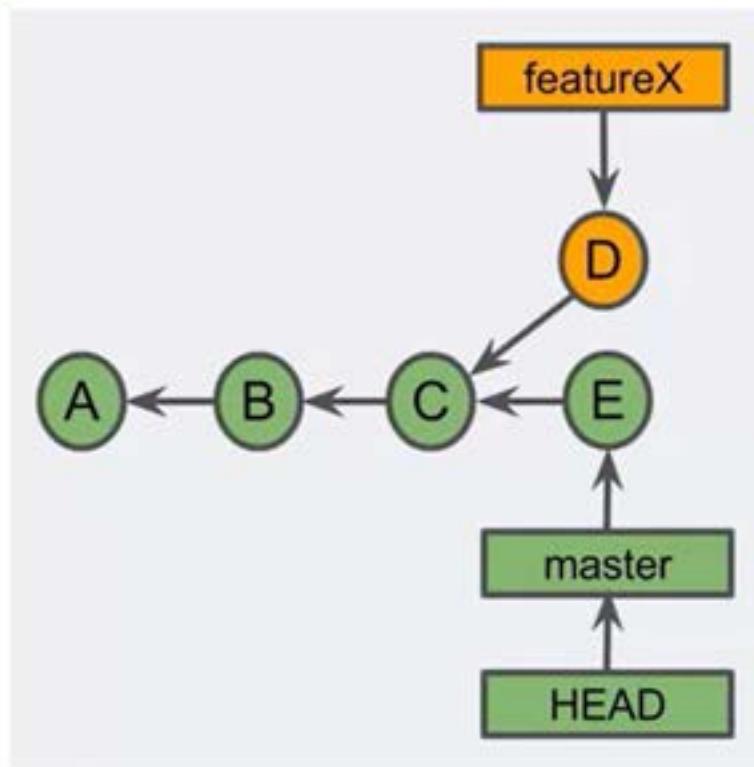


```
$ git branch -d featureX
Deleted branch featureX (was 010226b)
$ git branch
* master
```

DANGLING COMMITS

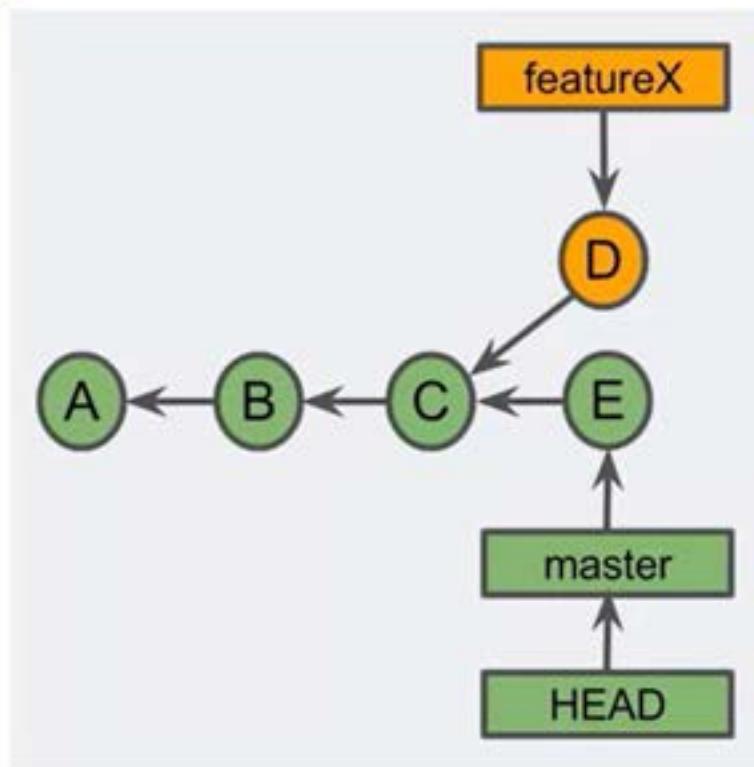


DANGLING COMMITS



```
$ git branch -d featureX  
error: The branch 'featureX' is not fully merged.  
If you are sure you want to delete it, run 'git branch -D featureX'.  
$ git branch -D featureX  
Deleted branch featureX (was 434dfa0)
```

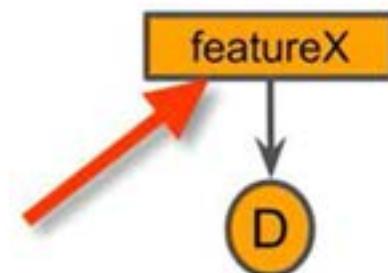
DANGLING COMMITS



```
$ git branch -d featureX  
error: The branch 'featureX' is not fully merged.  
If you are sure you want to delete it, run 'git branch -D featureX'.  
$ git branch -D featureX  
Deleted branch featureX (was 434dfa0)
```

UNDOING AN ACCIDENTAL BRANCH DELETE WITH `git reflog`

`git reflog` returns a *local* list
of recent **HEAD** commits



```
$ git branch -D featureX
Deleted branch featureX (was 434dfa0)

$ git reflog
942c36f (HEAD -> master) HEAD@{0}: checkout: moving from featureX to master
434dfa0 HEAD@{1}: commit: added featureX
942c36f (HEAD -> master) HEAD@{2}: checkout: moving from master to featureX
...
$ git checkout -b featureX 434dfa0
Switched to a new branch 'featureX'
```

REVIEW

- A branch is a set of commits that trace back to the project's first commit
- Creating a branch creates a branch label
- Checkout involves updating HEAD and updating the working tree
- A detached HEAD reference points directly to a commit
- Fix a detached HEAD by creating a branch
- Deleting a branch deletes a branch label
- Dangling commits will eventually be garbage collected

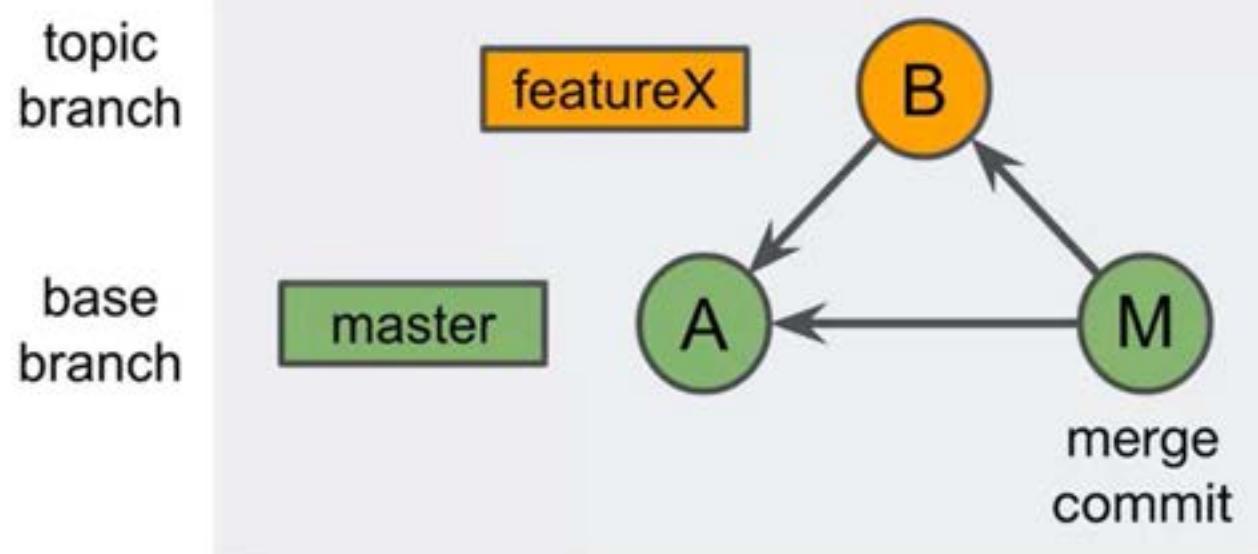
Topics

Merging overview

Fast-forward merges

Merge commits

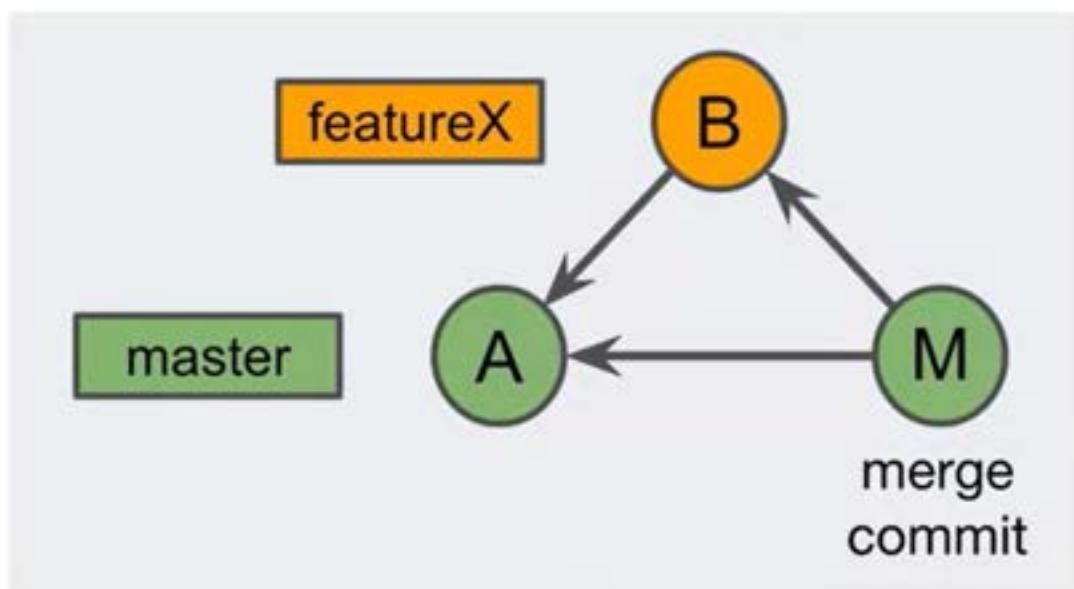
MERGING



MERGING

Main types of merges:

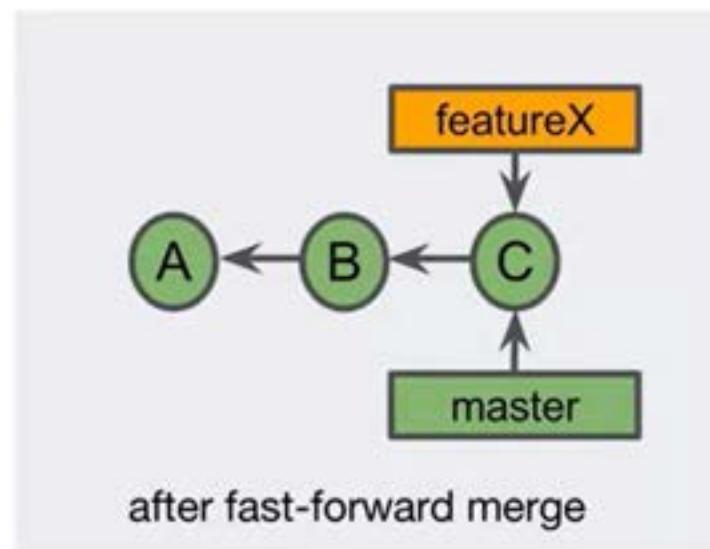
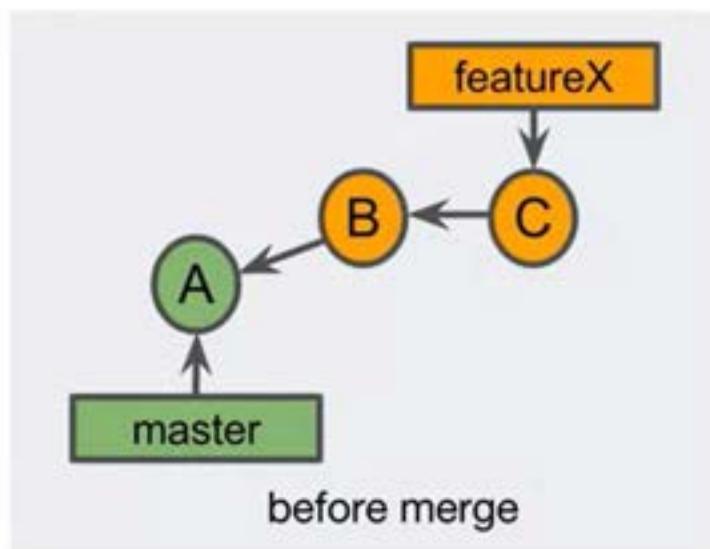
1. Fast-forward merge
2. Merge commit
3. Squash merge*
4. Rebase*



* We will discuss squash merges and rebasing later- they rewrite the commit history

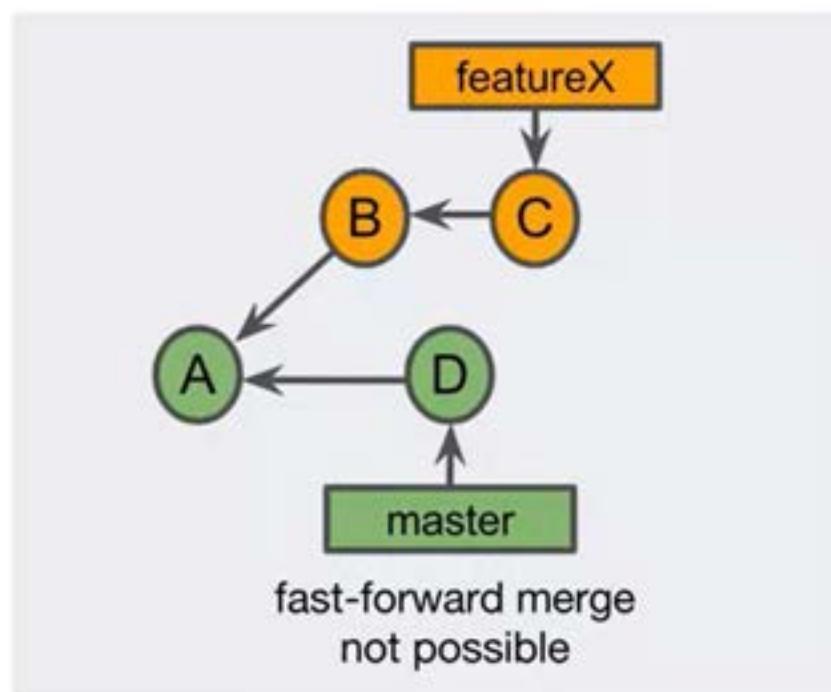
FAST-FORWARD (FF) MERGE

Moves the base branch label to the tip of the topic branch



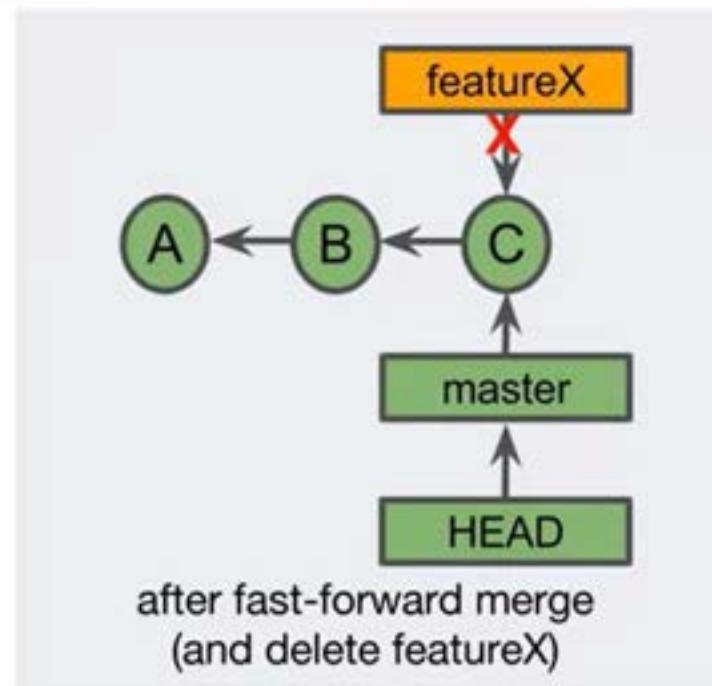
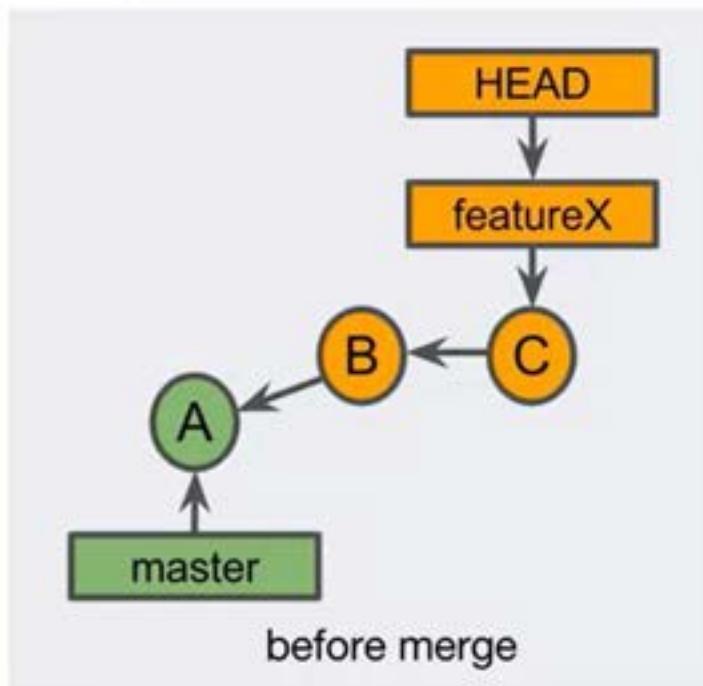
CONDITIONS FOR A FAST-FORWARD MERGE

Possible if no other commits have been made to the base branch since branching



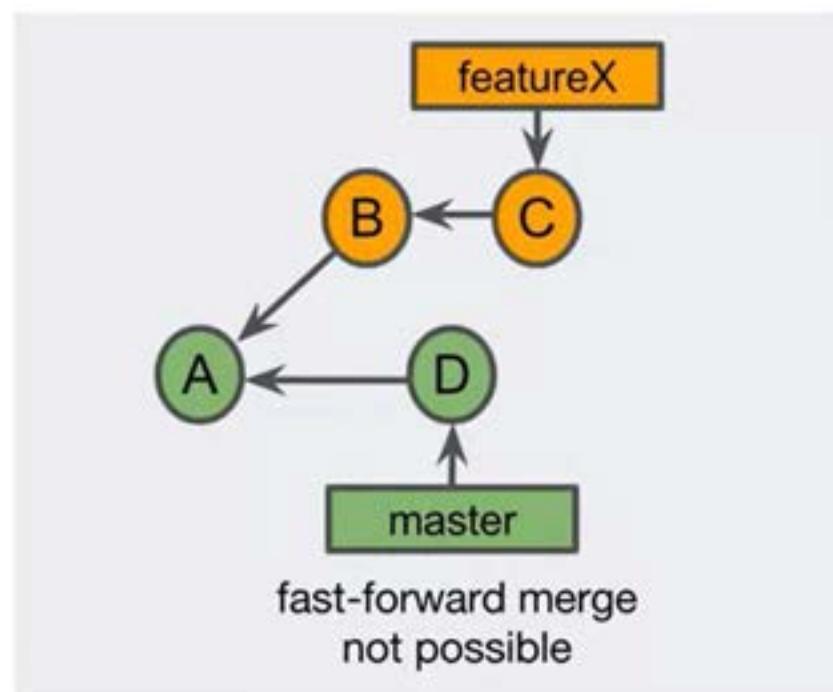
PERFORMING A FAST-FORWARD MERGE

1. Checkout master
2. Merge featureX
 - a. Attempting a fast forward merge is the default
3. Delete the featureX branch label



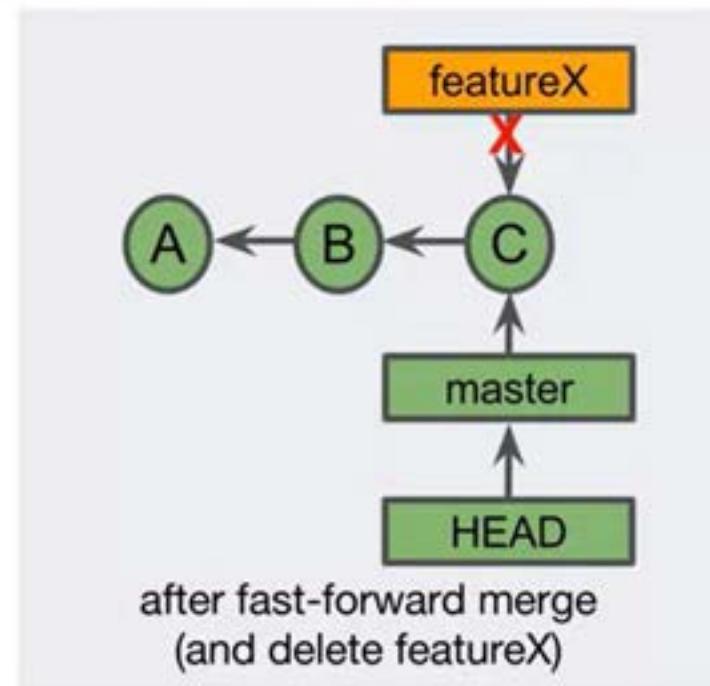
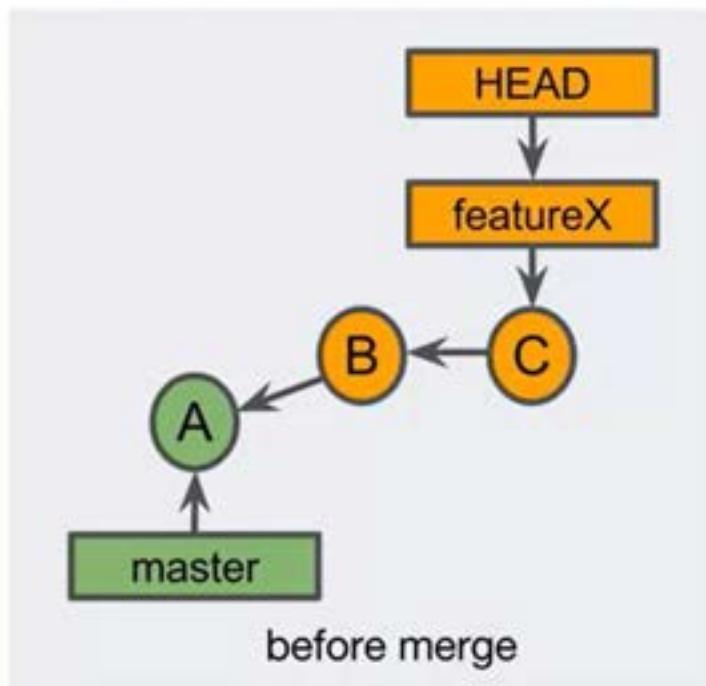
CONDITIONS FOR A FAST-FORWARD MERGE

Possible if no other commits have been made to the base branch since branching



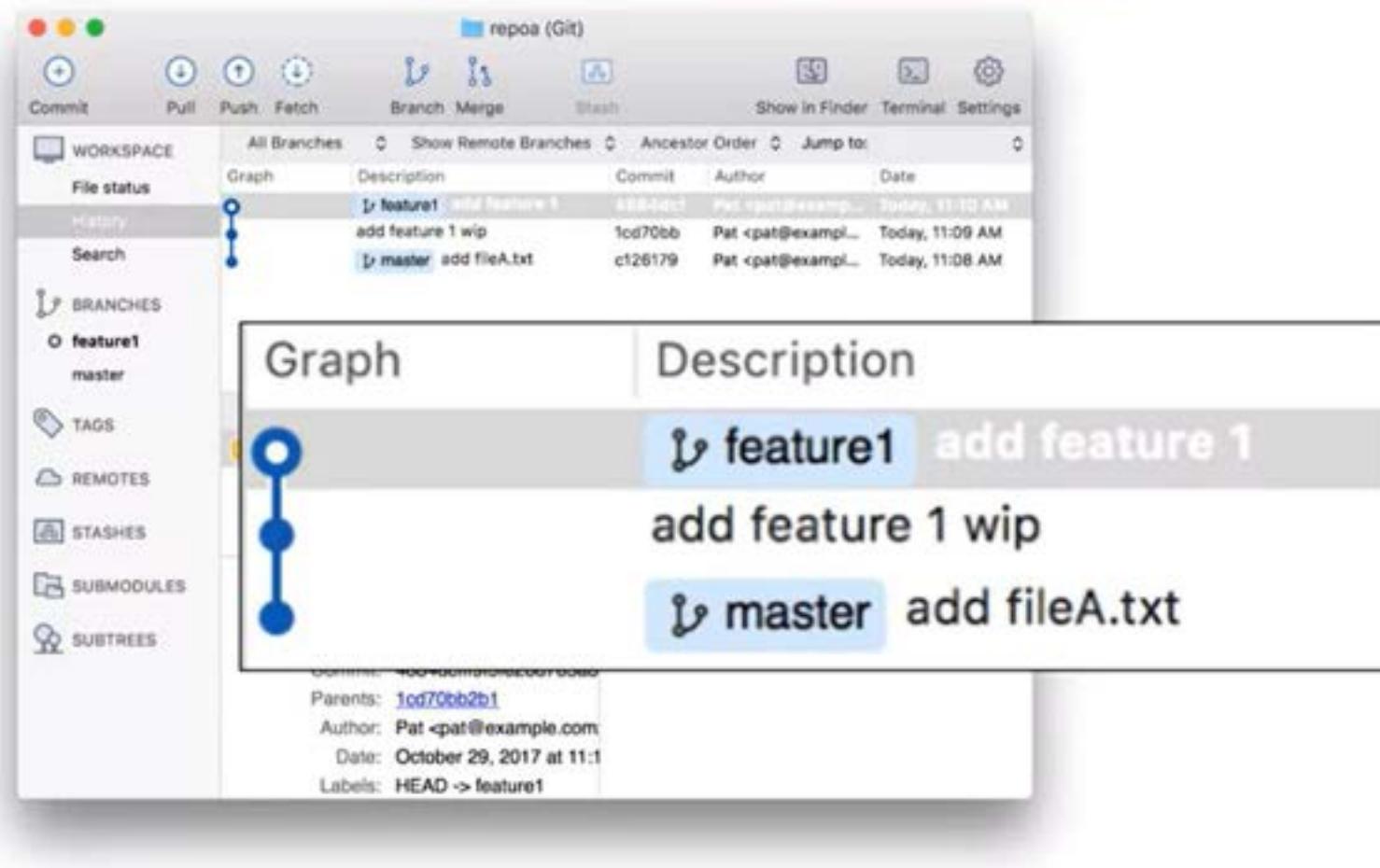
PERFORMING A FAST-FORWARD MERGE

1. Checkout master
2. Merge featureX
 - a. Attempting a fast forward merge is the default
3. Delete the featureX branch label



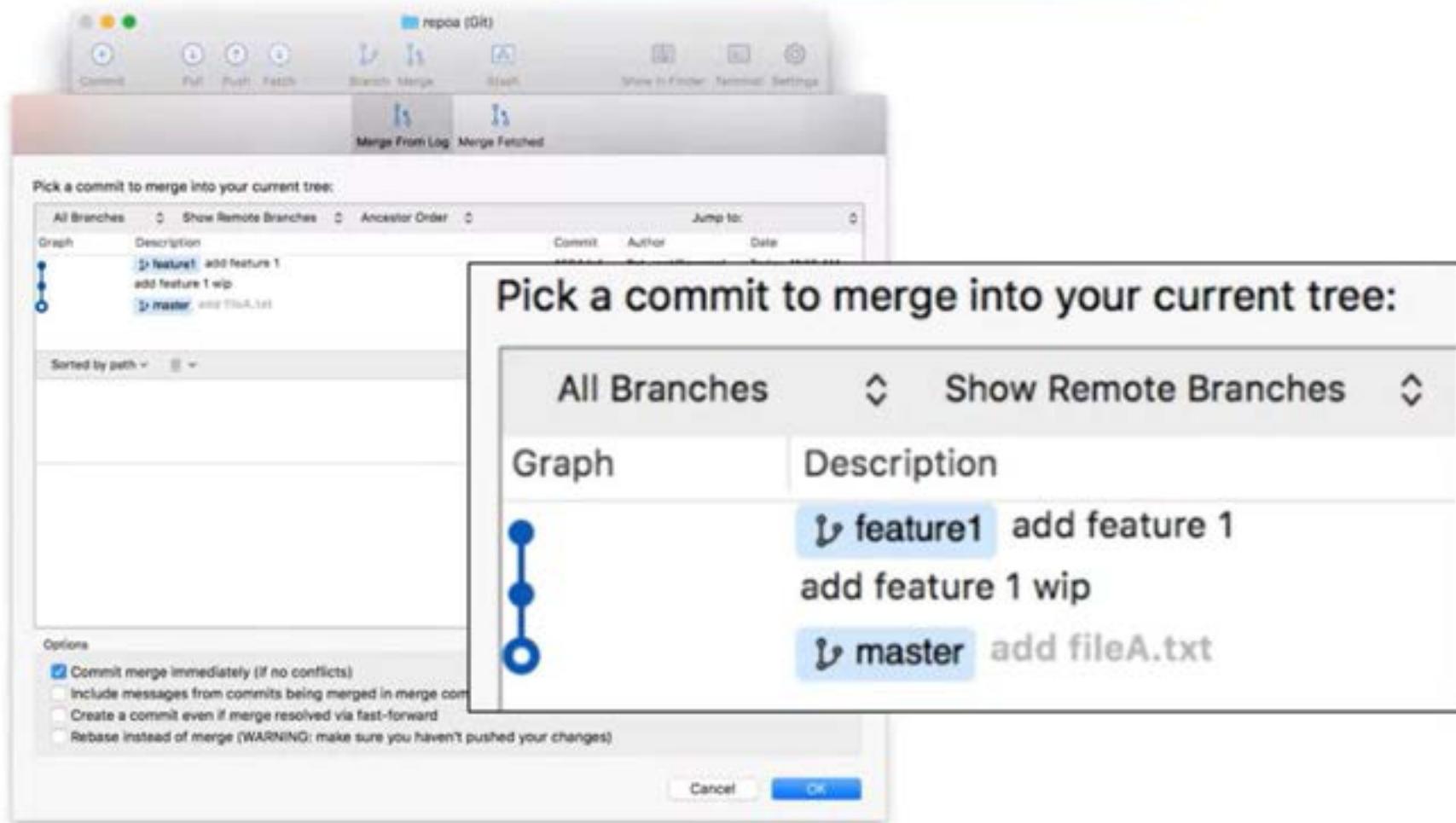
FAST-FORWARD MERGE

1. Checkout master
2. Merge featureX
3. Delete the featureX branch



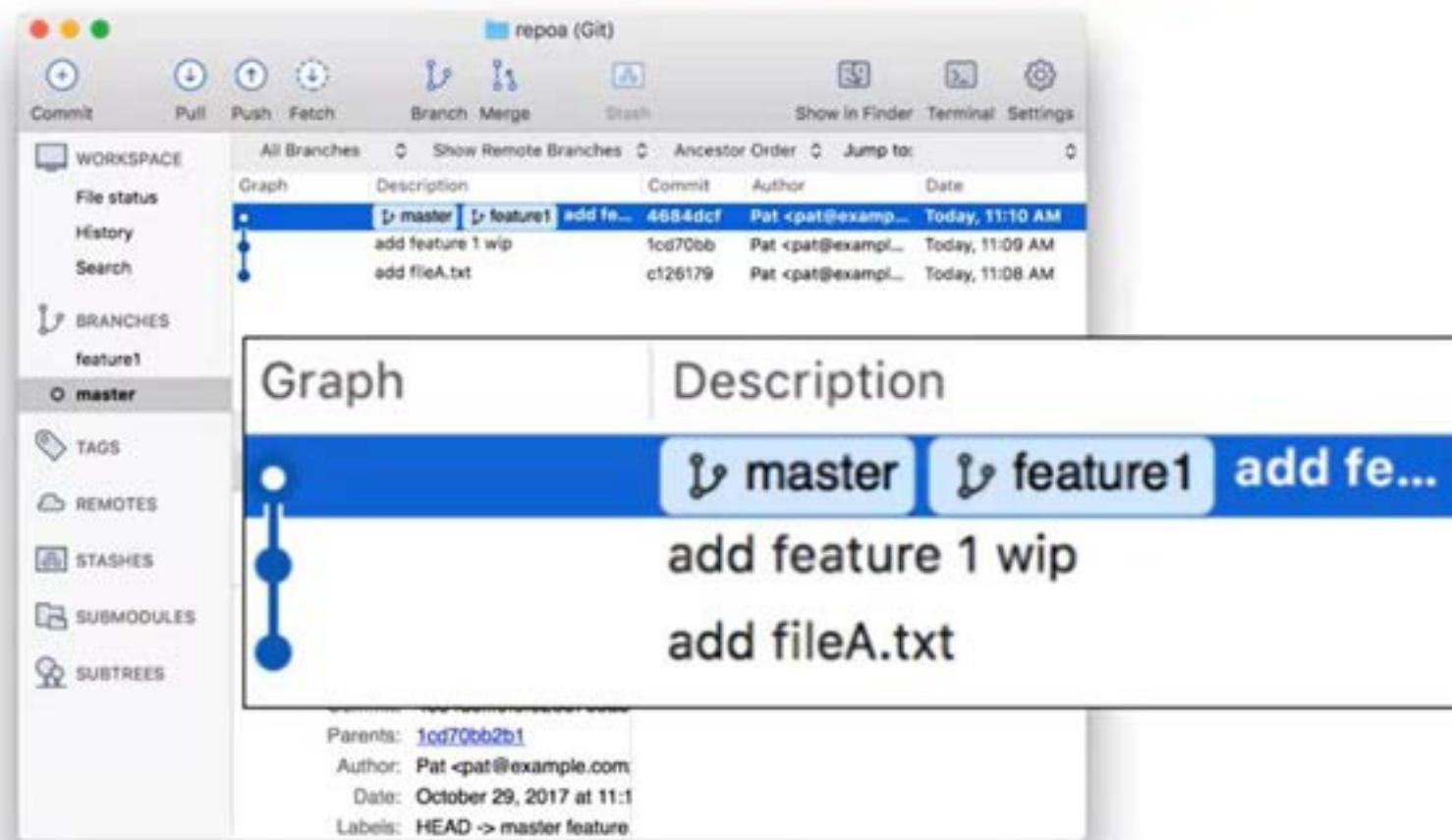
FAST-FORWARD MERGE

1. Checkout master
2. Merge featureX
3. Delete the featureX branch



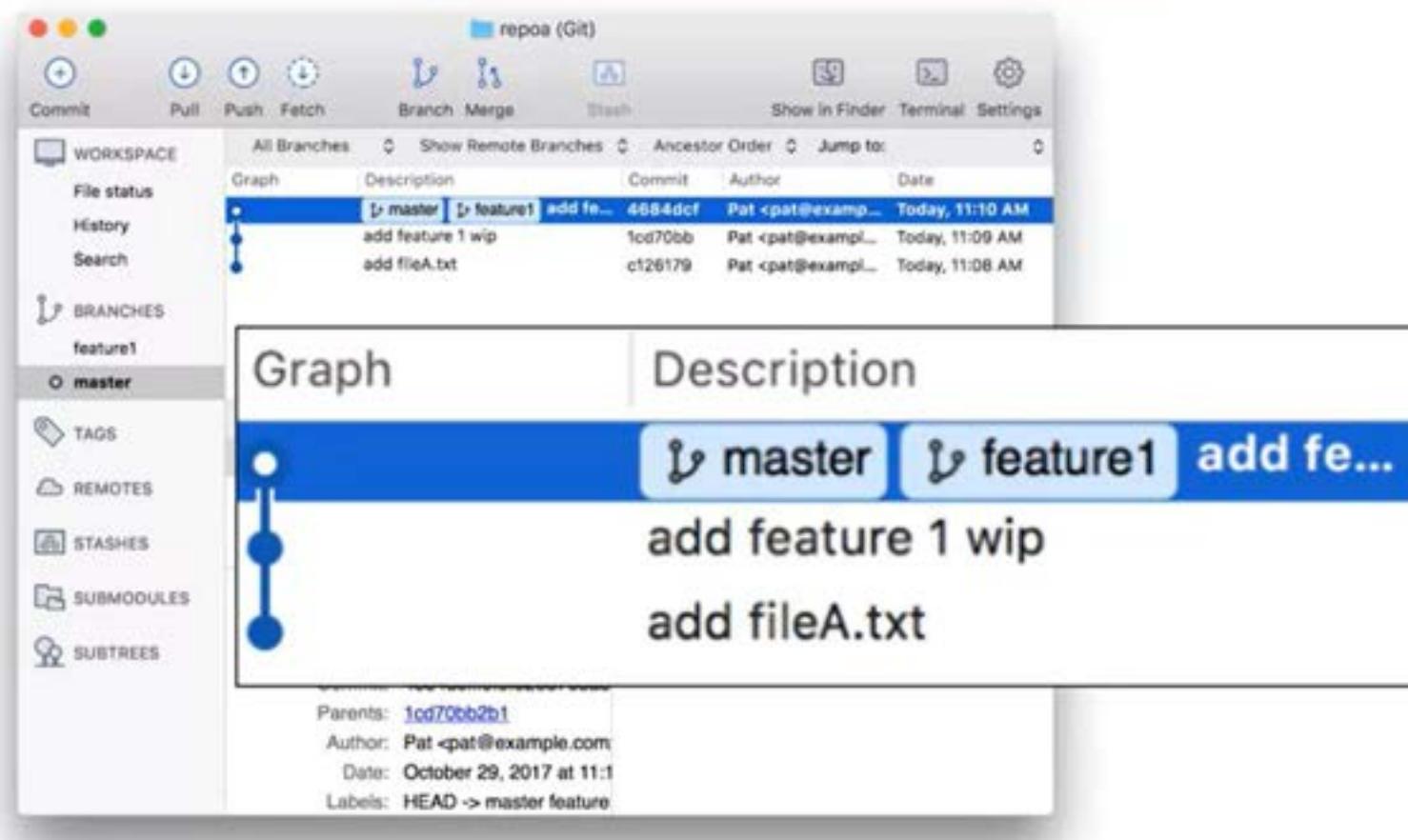
FAST-FORWARD MERGE

1. Checkout master
2. Merge featureX
3. Delete the featureX branch



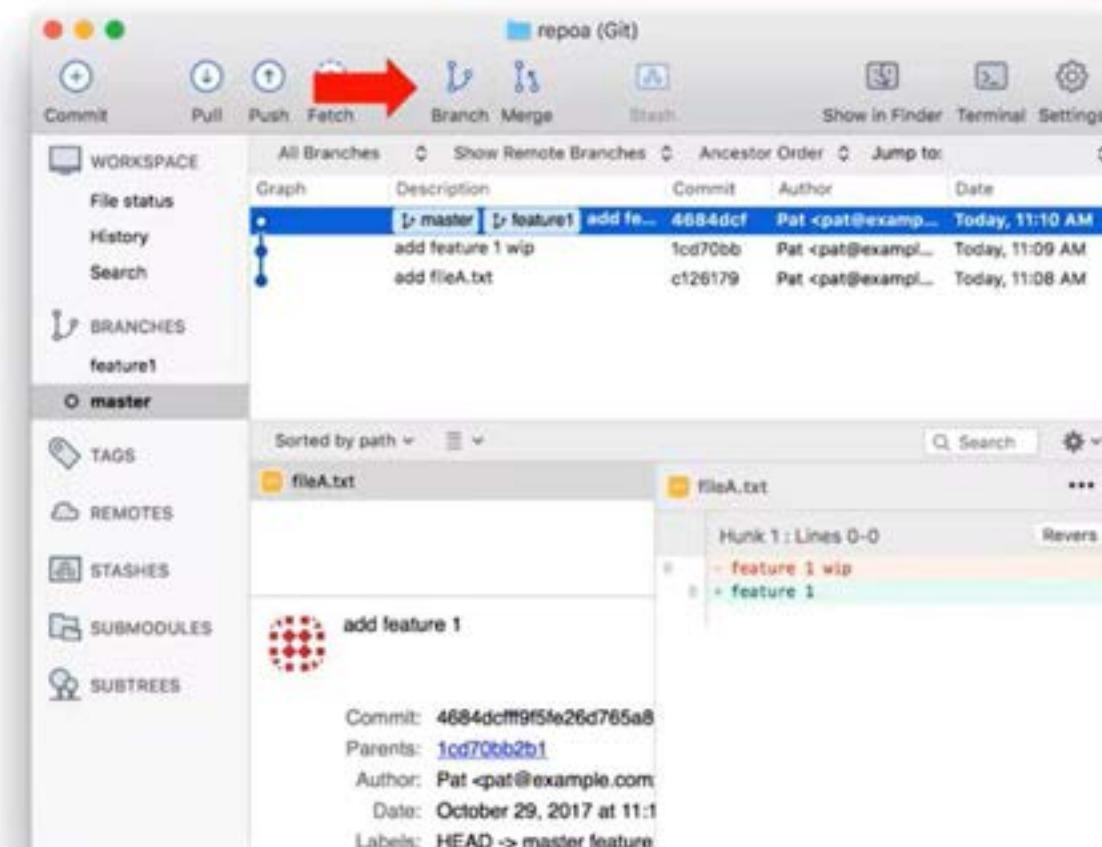
FAST-FORWARD MERGE

1. Checkout master
2. Merge featureX
3. Delete the featureX branch

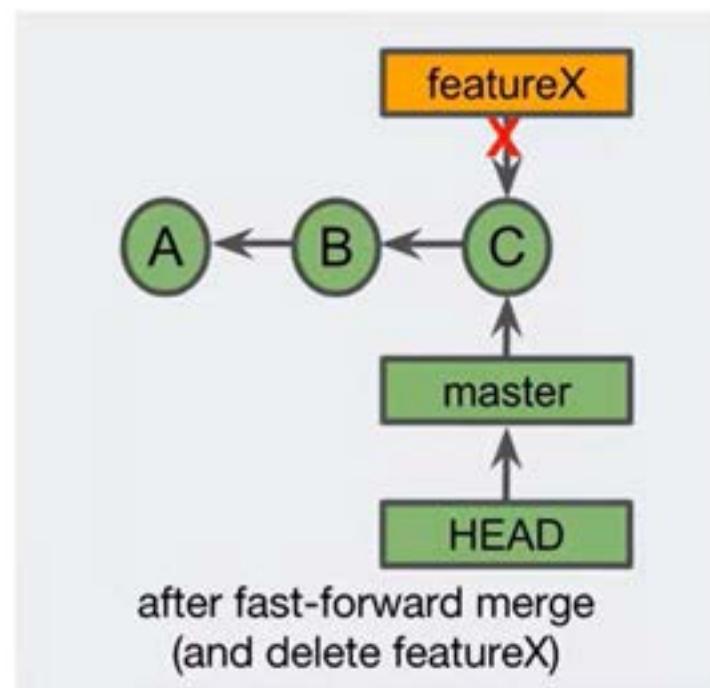
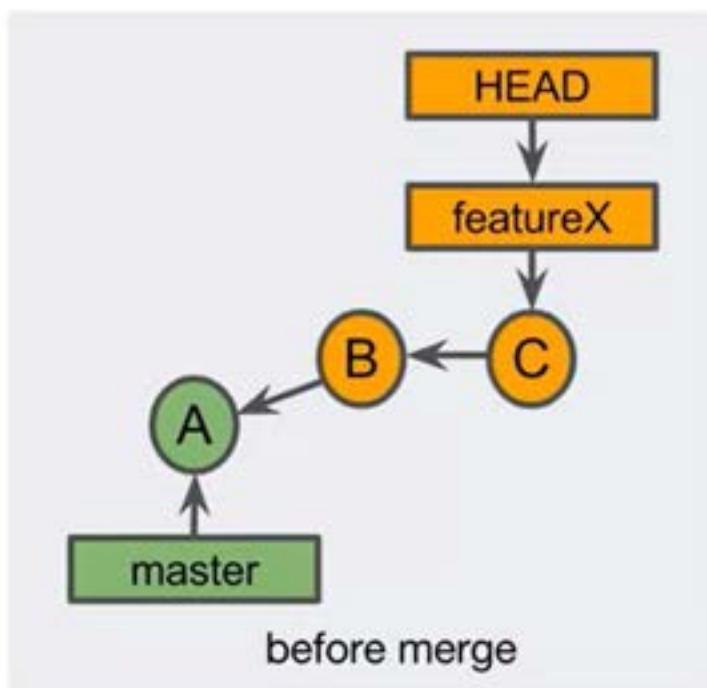


FAST-FORWARD MERGE

1. Checkout master
2. Merge featureX
3. Delete the featureX branch

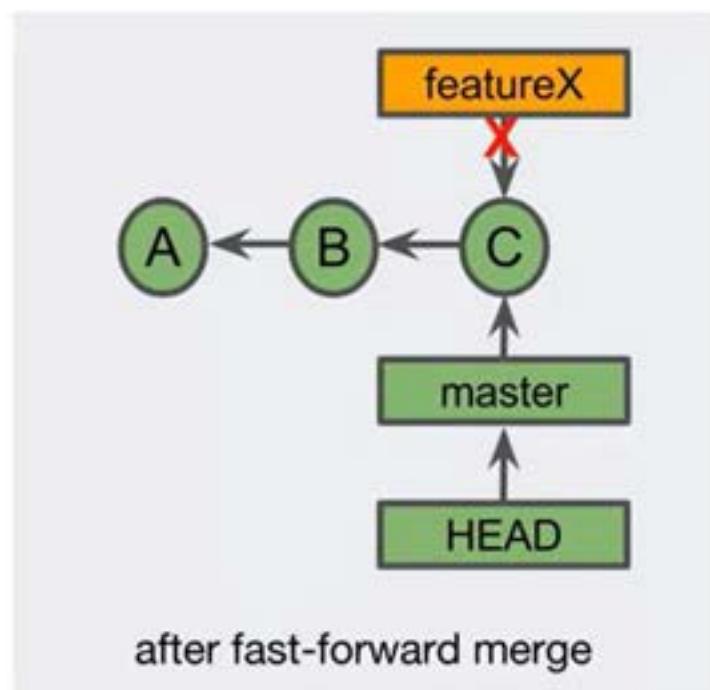
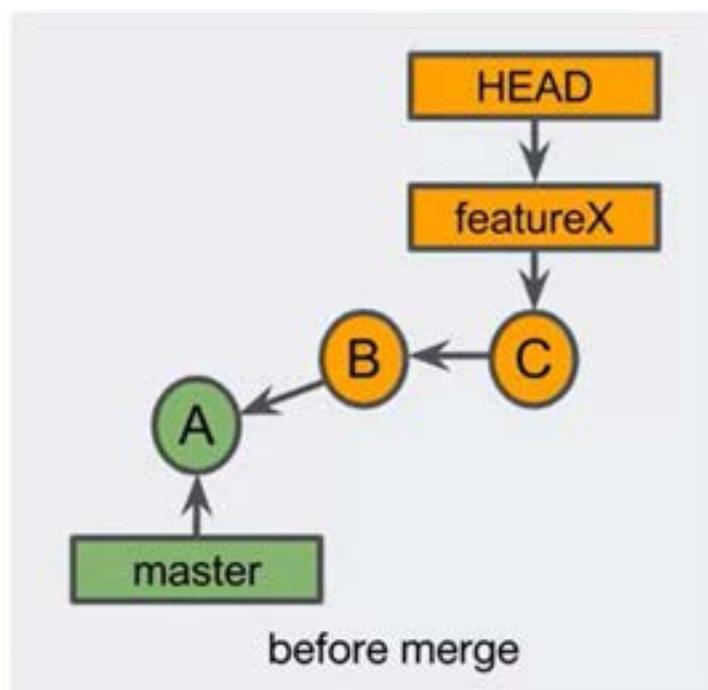


WHY DELETE THE FEATURE BRANCH LABEL?



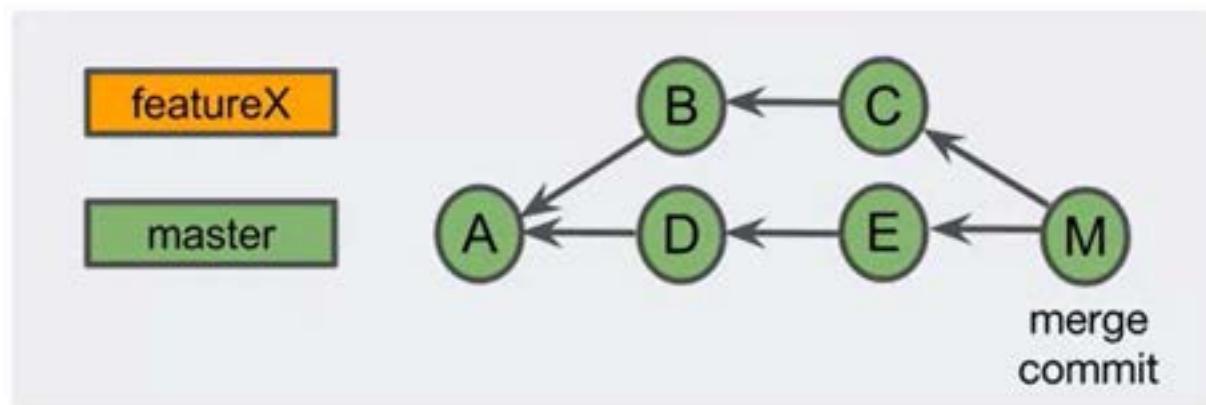
LINEAR COMMIT HISTORY

The resulting commit history is linear



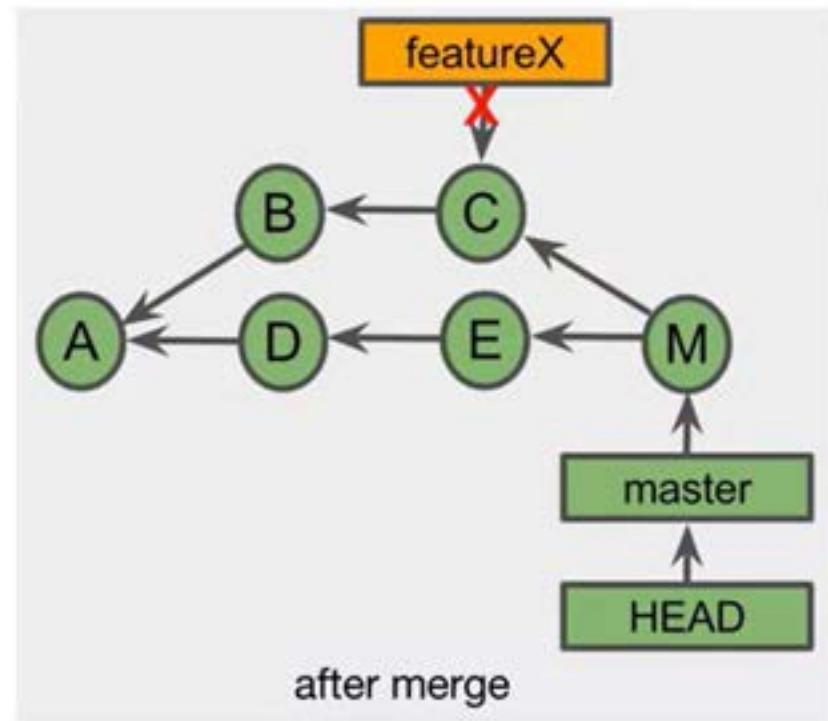
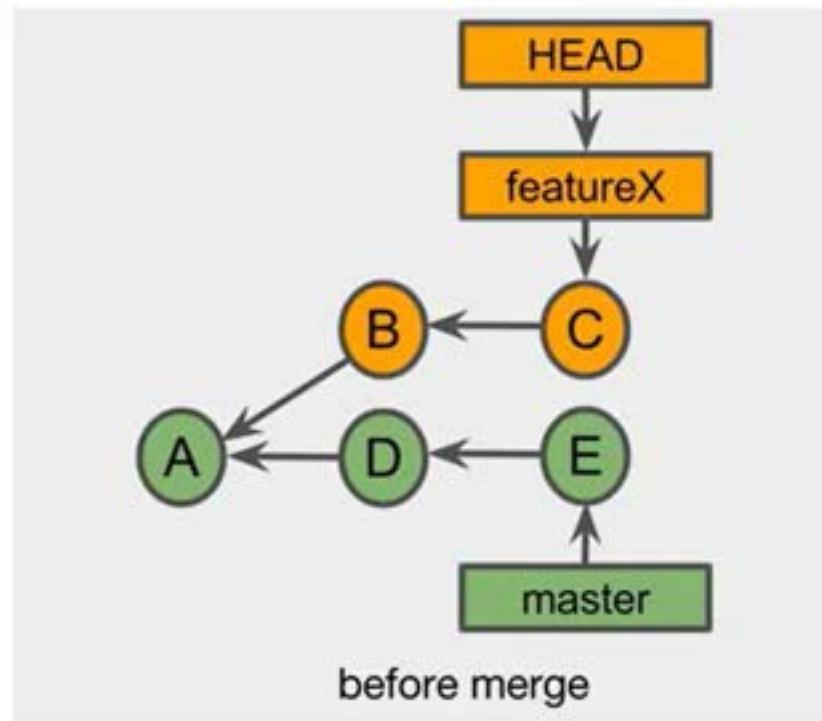
MERGE COMMIT

1. Combines the commits at the tips of the merged branches
2. Places the result in the merge commit



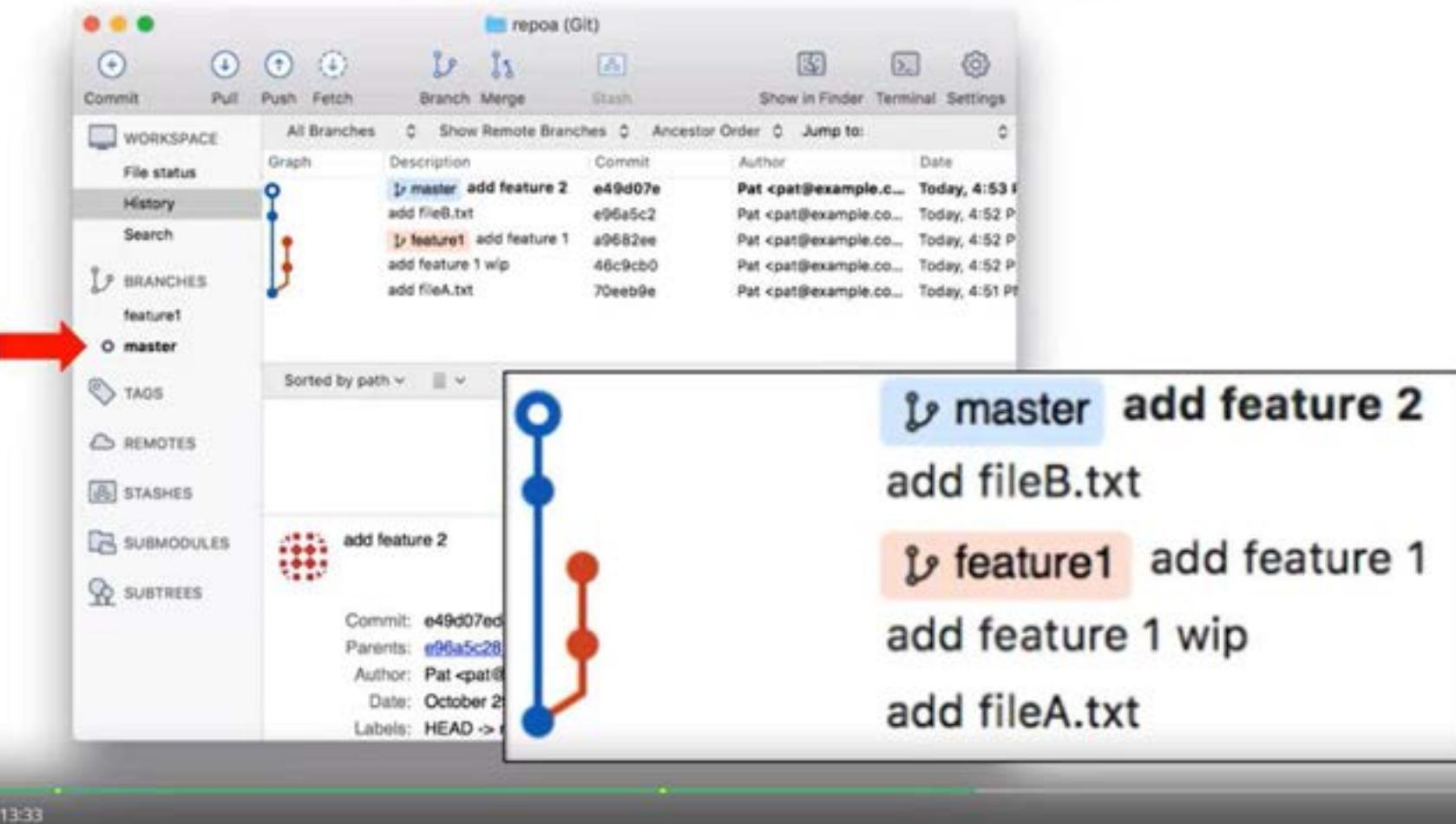
PERFORMING A MERGE COMMIT

1. Checkout master
2. Merge featureX
3. Delete the featureX branch



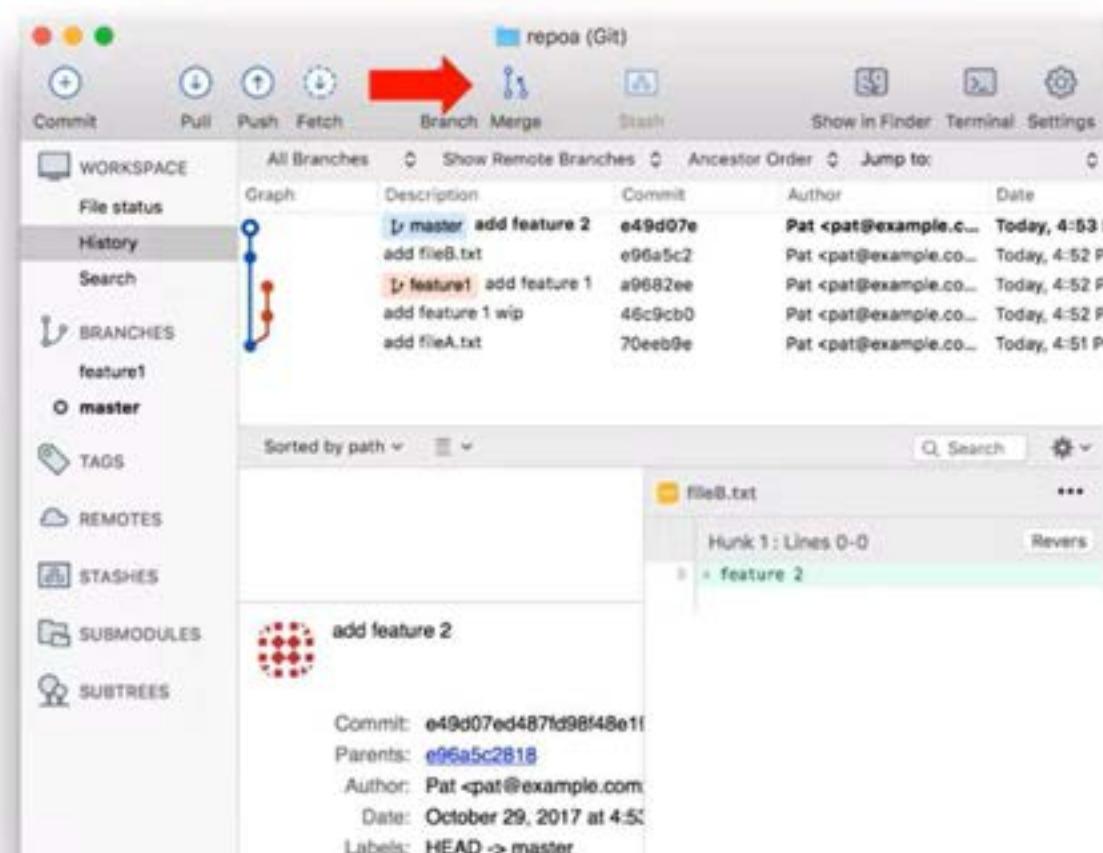
MERGE COMMIT

1. Checkout master
2. Merge featureX
3. Delete the featureX branch



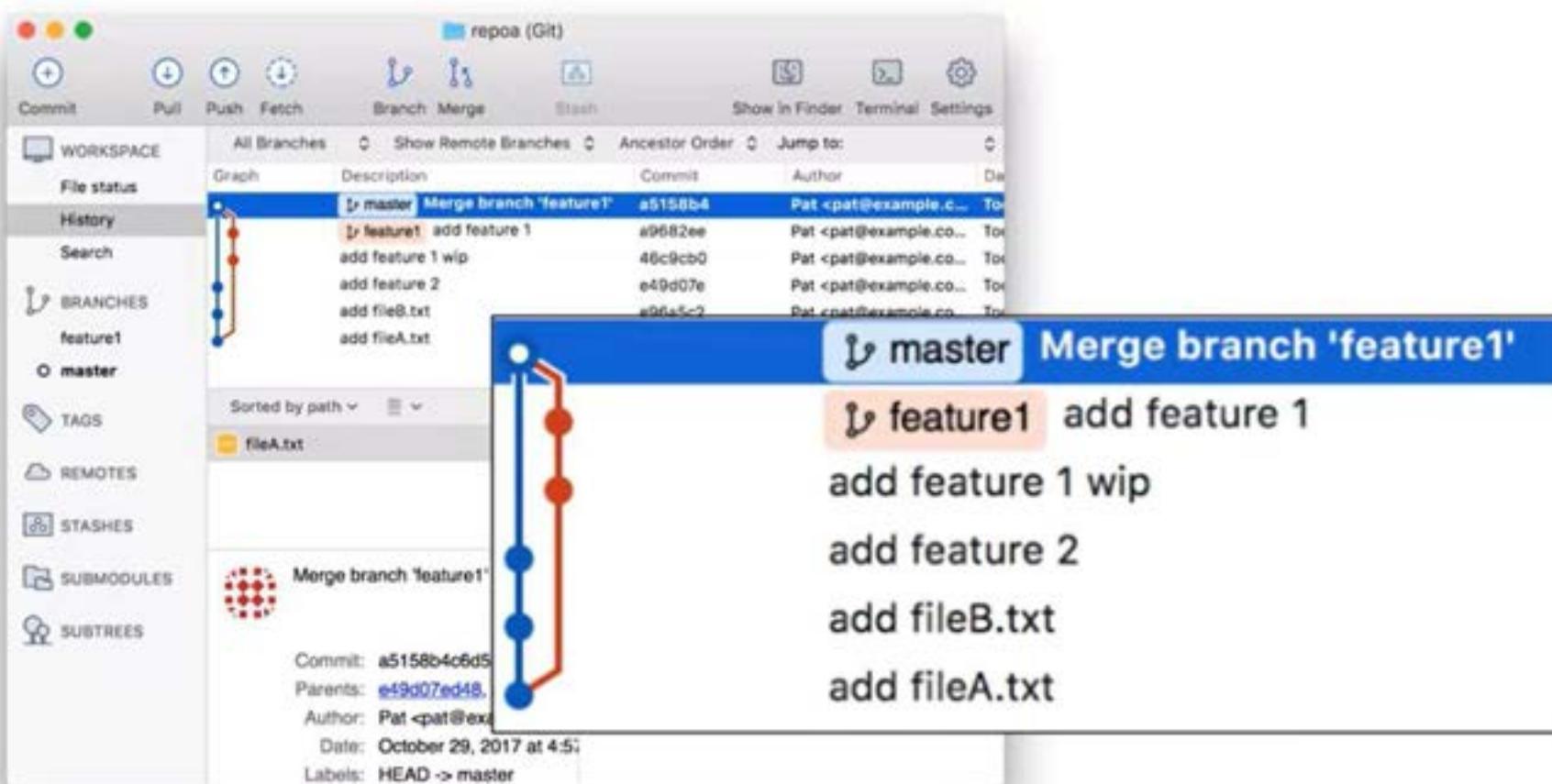
MERGE COMMIT

1. Checkout master
2. Merge featureX
3. Delete the featureX branch



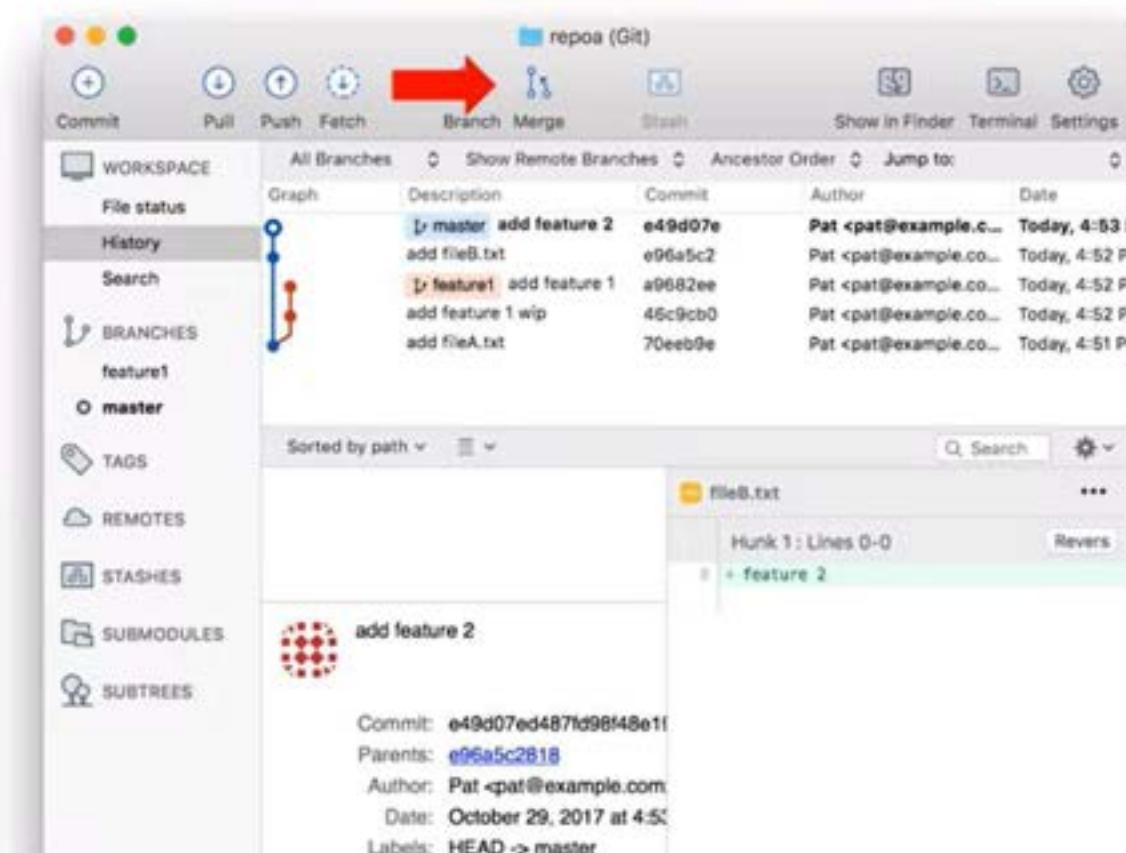
MERGE COMMIT

1. Checkout master
2. Merge featureX
3. Delete the featureX branch



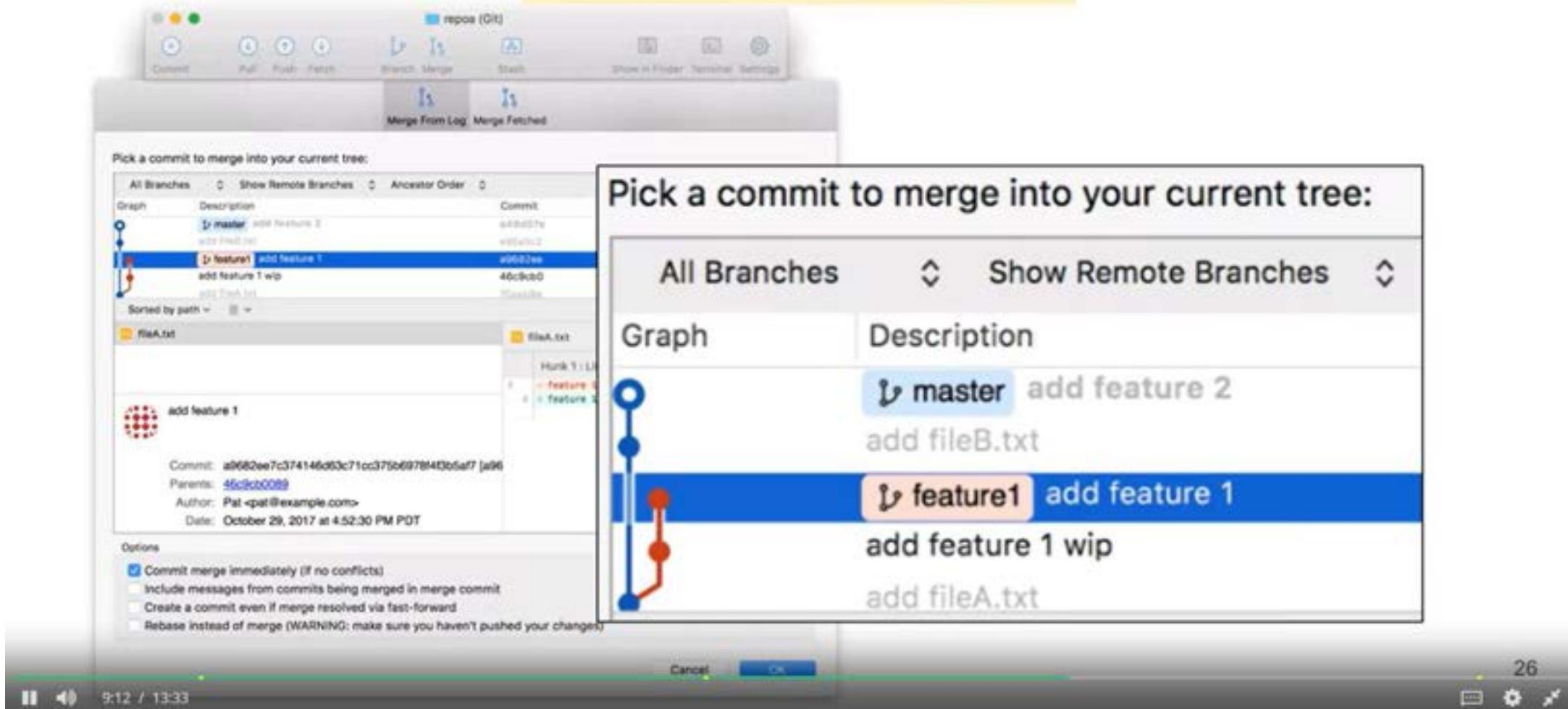
MERGE COMMIT

1. Checkout master
2. Merge featureX
3. Delete the featureX branch



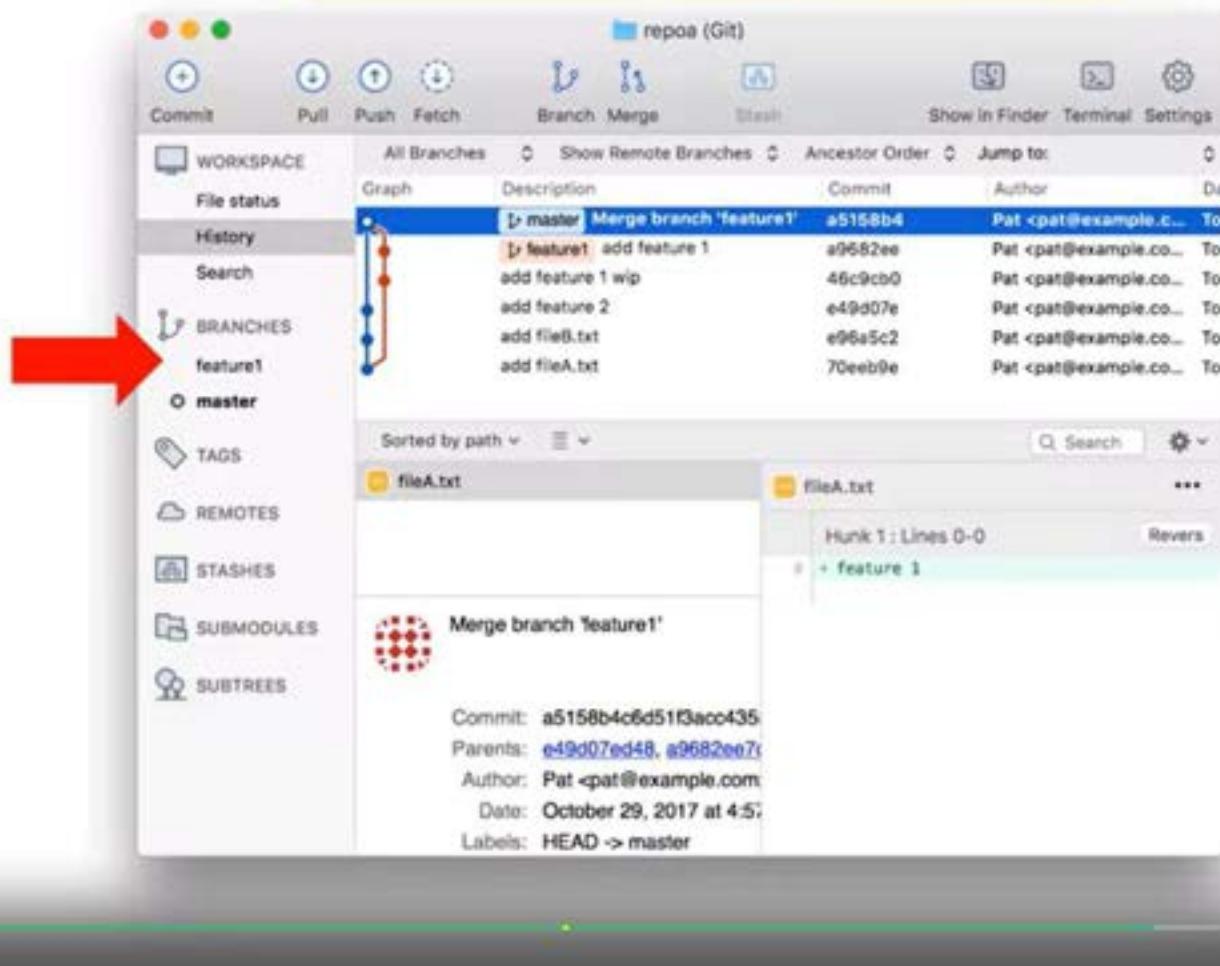
MERGE COMMIT

1. Checkout **master**
2. **Merge featureX**
3. Delete the featureX branch



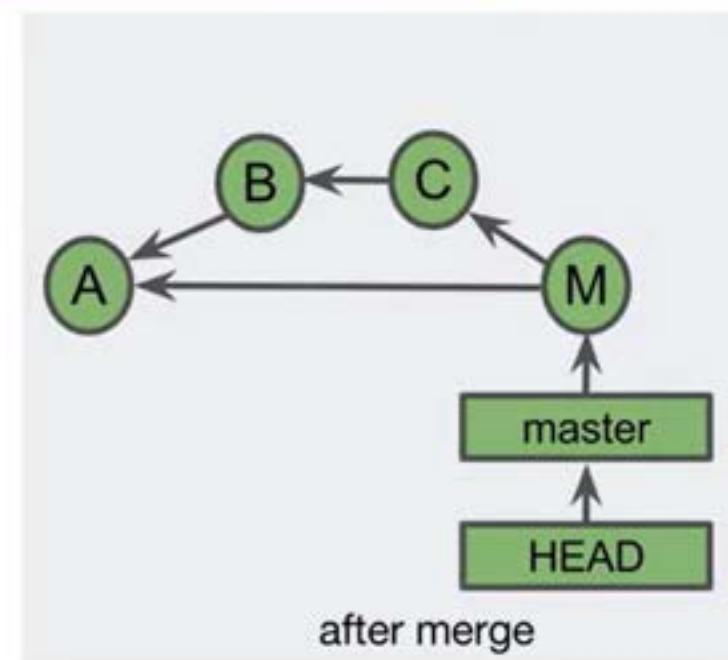
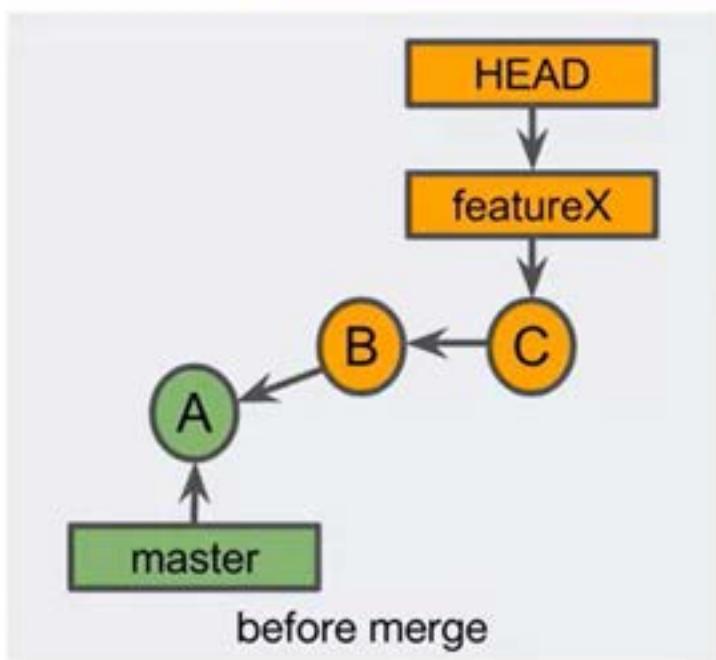
MERGE COMMIT

1. Checkout master
2. Merge featureX
3. Delete the featureX branch

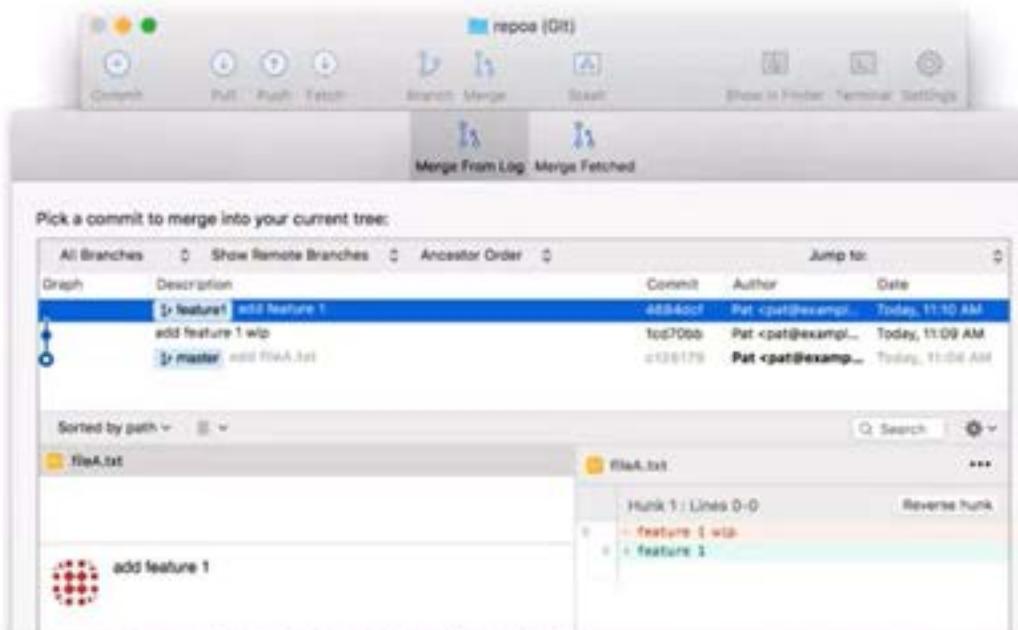


MERGE COMMIT (NO FAST-FORWARD)

1. Checkout master
2. Merge featureX
 - a. Select **Create a commit even if merge resolved via fast-forward**
3. Delete the featureX branch



MERGE COMMIT (NO FAST-FORWARD)



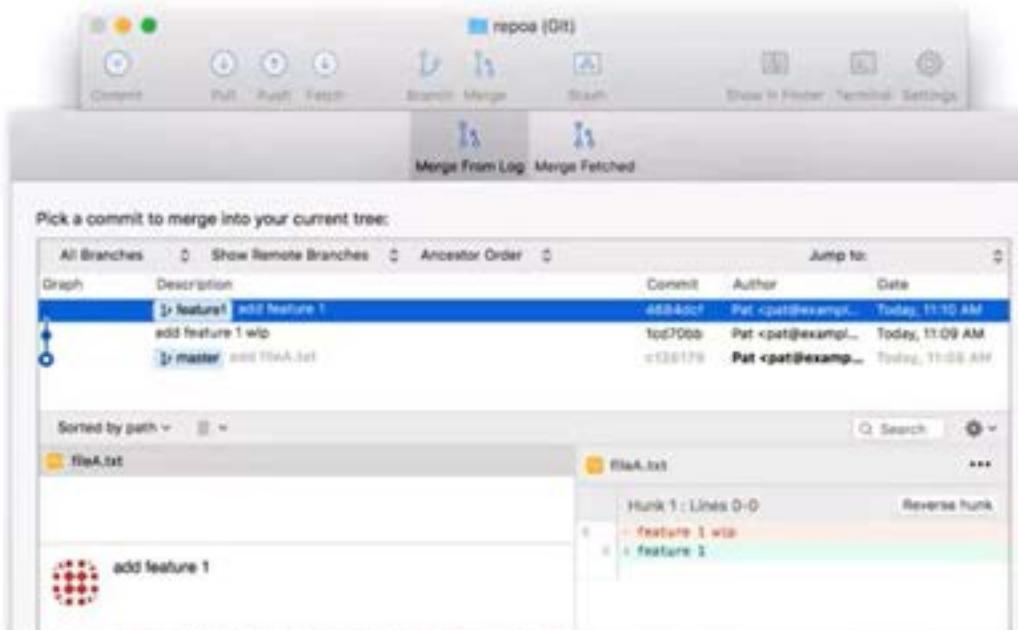
Create a commit even if merge resolved via fast-forward

- Options
- Commit merge immediately (if no conflicts)
Include messages from commits being merged in merge commit
 - Create a commit even if merge resolved via fast-forward
Rebase instead of merge (WARNING: make sure you haven't pushed your changes)

Cancel

OK

MERGE COMMIT (NO FAST-FORWARD)



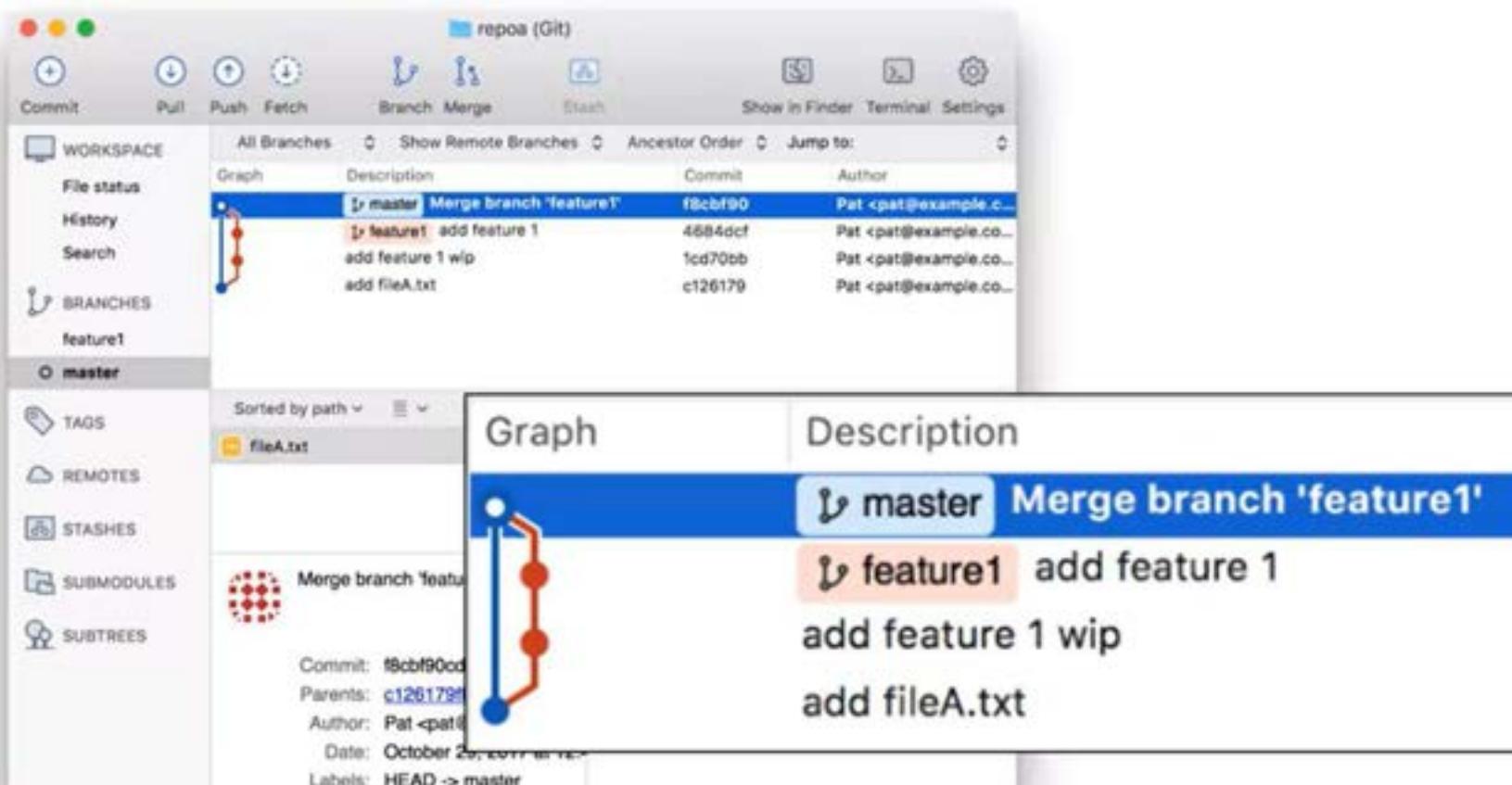
Create a commit even if merge resolved via fast-forward

- Options
- Commit merge immediately (if no conflicts)
Include messages from commits being merged in merge commit
 - Create a commit even if merge resolved via fast-forward
Rebase instead of merge (WARNING: make sure you haven't pushed your changes!)

Cancel

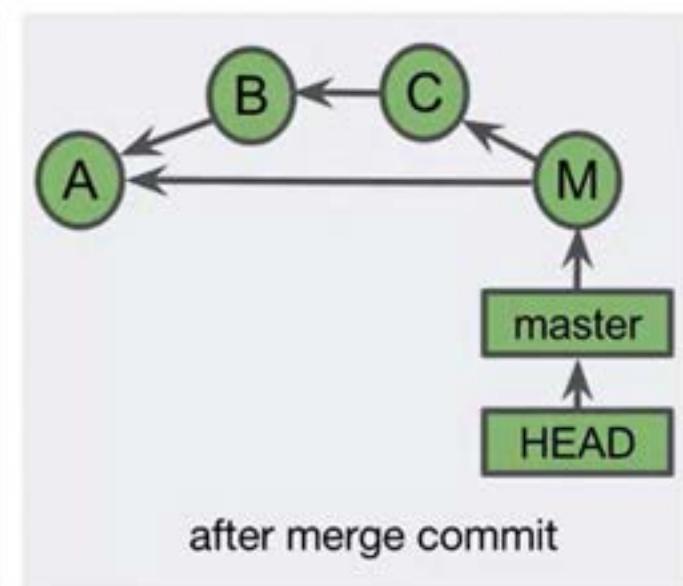
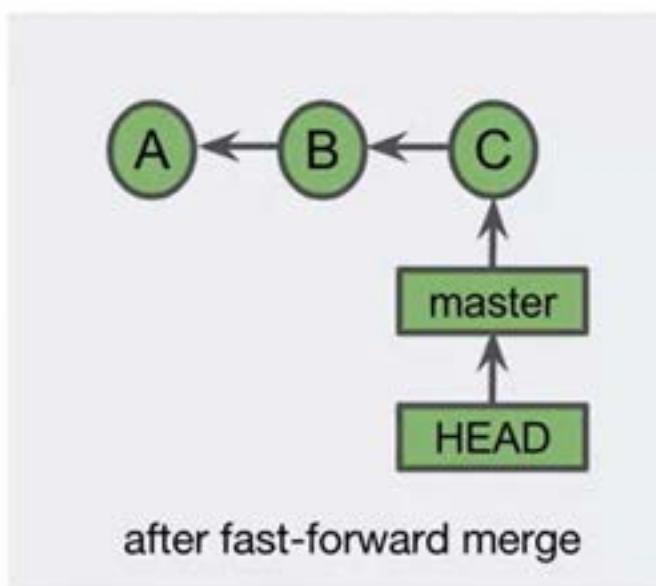
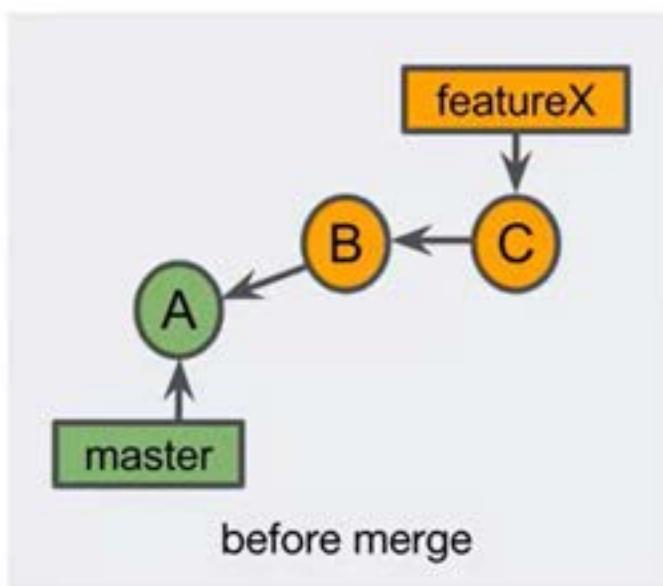
OK

MERGE COMMIT (NO FAST-FORWARD)



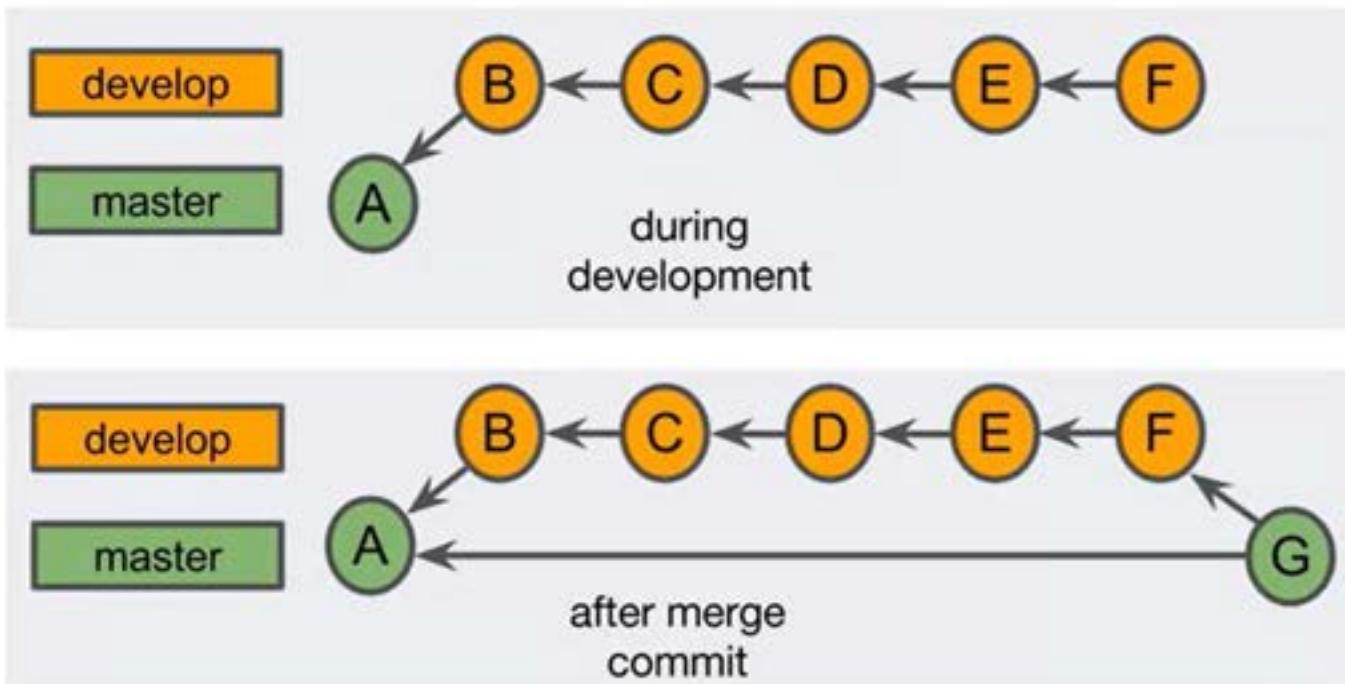
TEAM COMMIT HISTORY POLICIES

- Require a linear history
- Require merge commits
- Let the person merging decide



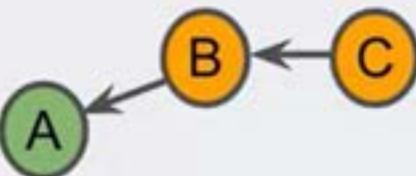
MERGING LONG-RUNNING BRANCHES

These merges are typically easy because they are fast-forwardable



REVIEW

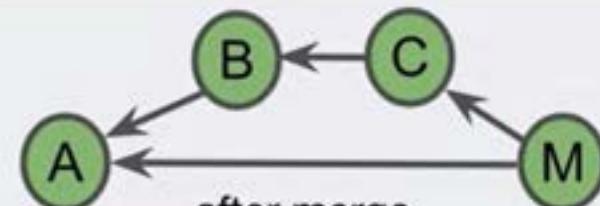
- Merging combines the work of multiple branches
- A fast-forward merge moves the base branch label to the tip of the topic branch
- A merge is fast-forwardable if no other commits have been made to the base branch since branching
- A merge commit is the result of combining the work of multiple commits
- A merge commit has multiple parents



before merge



after fast-forward
merge



after merge
commit

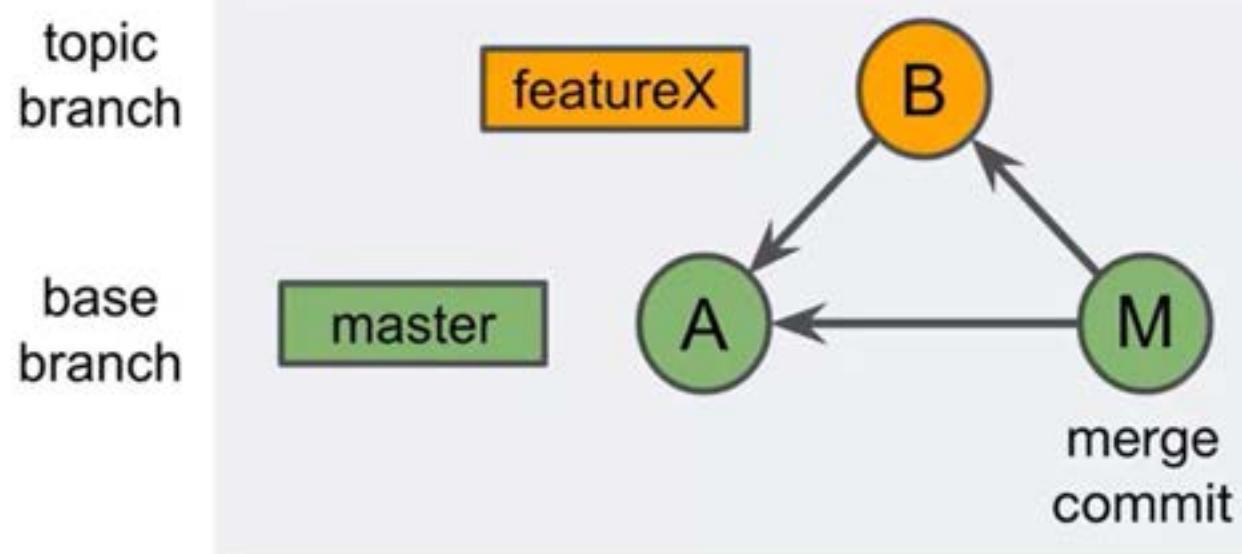
Topics

Merging overview

Fast-forward merges

Merge commits

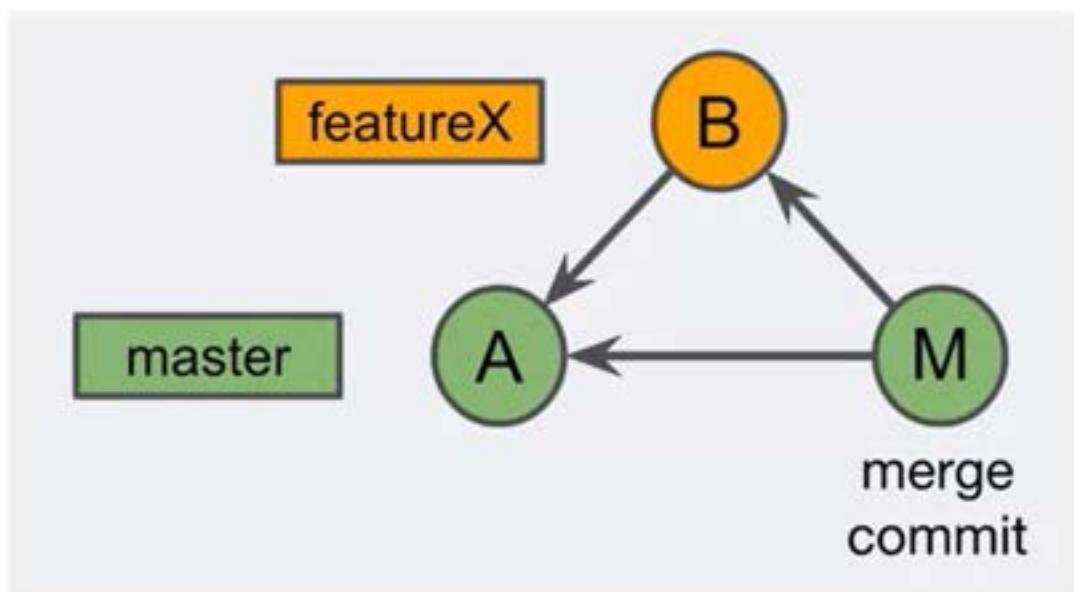
MERGING



MERGING

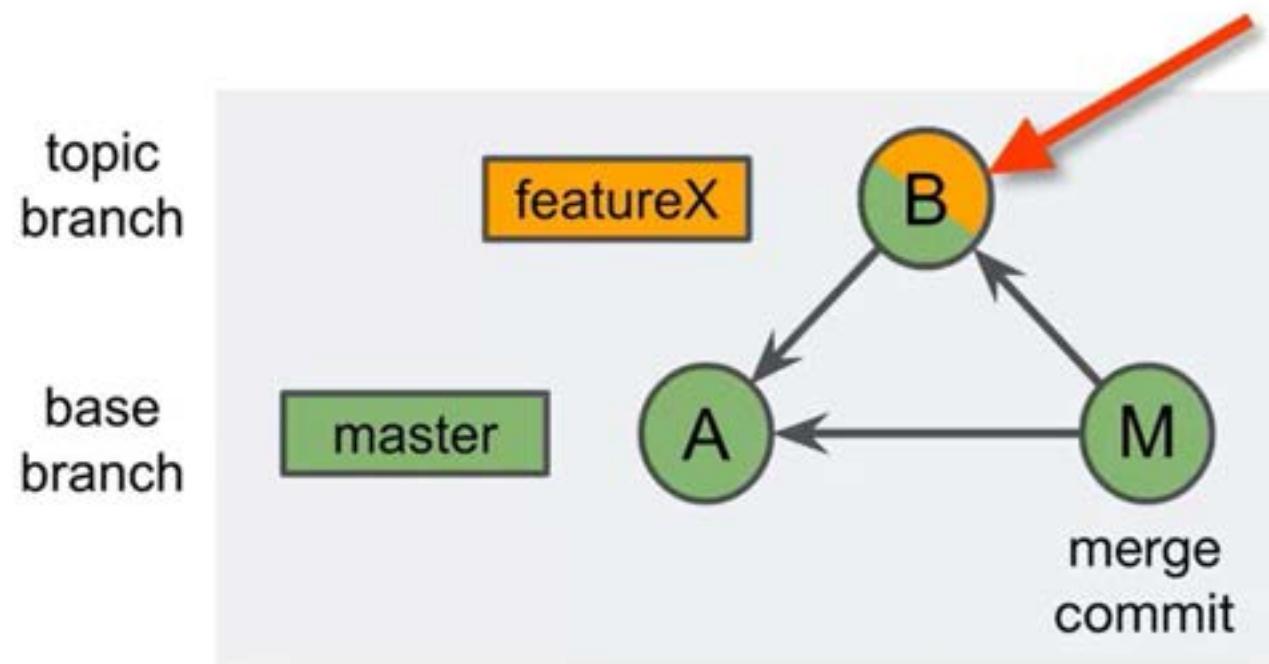
Main types of merges:

1. Fast-forward merge
2. Merge commit
3. Squash merge*
4. Rebase*



* We will discuss squash merges and rebasing later- they rewrite the commit history

MERGING



Topics

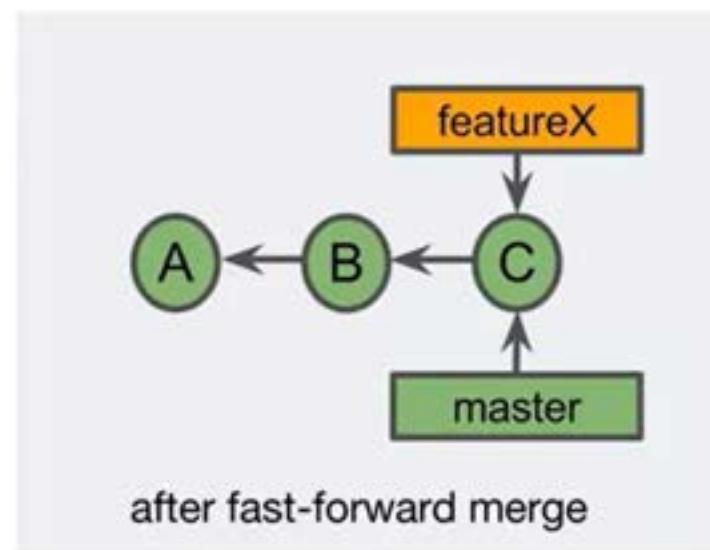
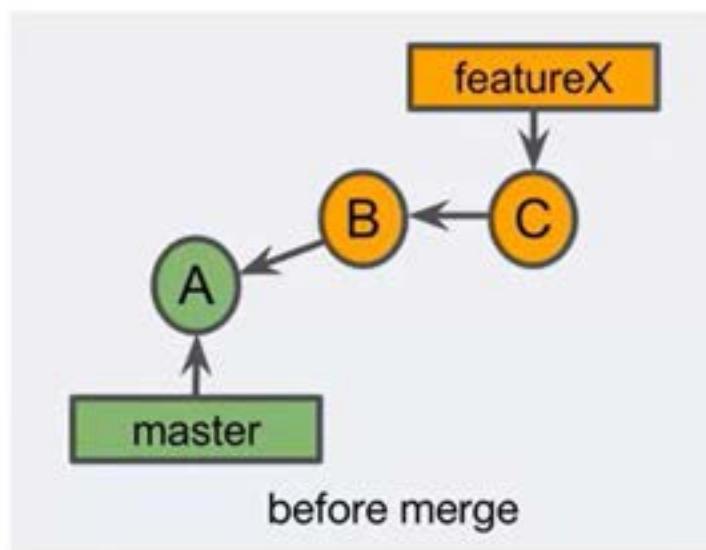
Merging overview

Fast-forward merges

Merge commits

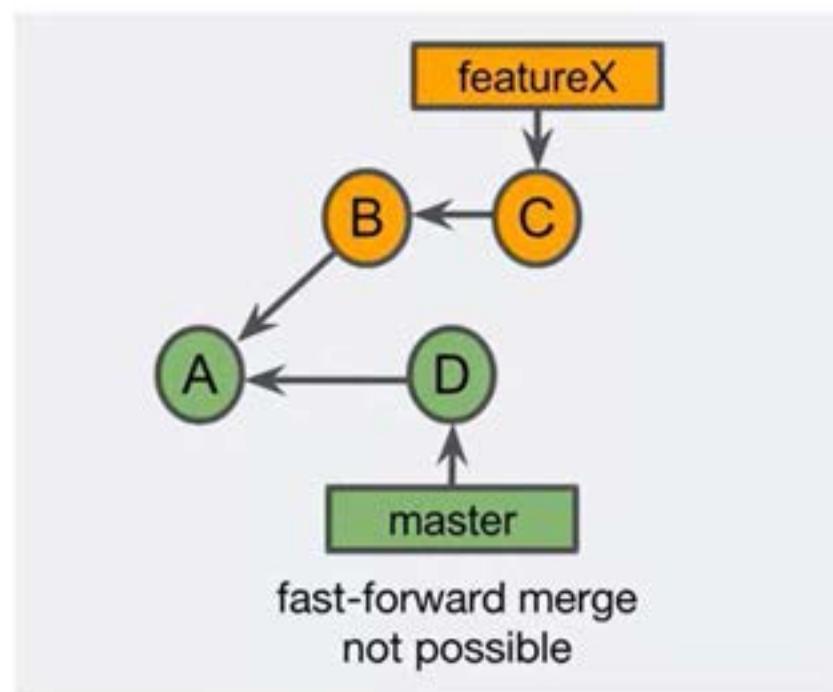
FAST-FORWARD (FF) MERGE

Moves the base branch label to the tip of the topic branch



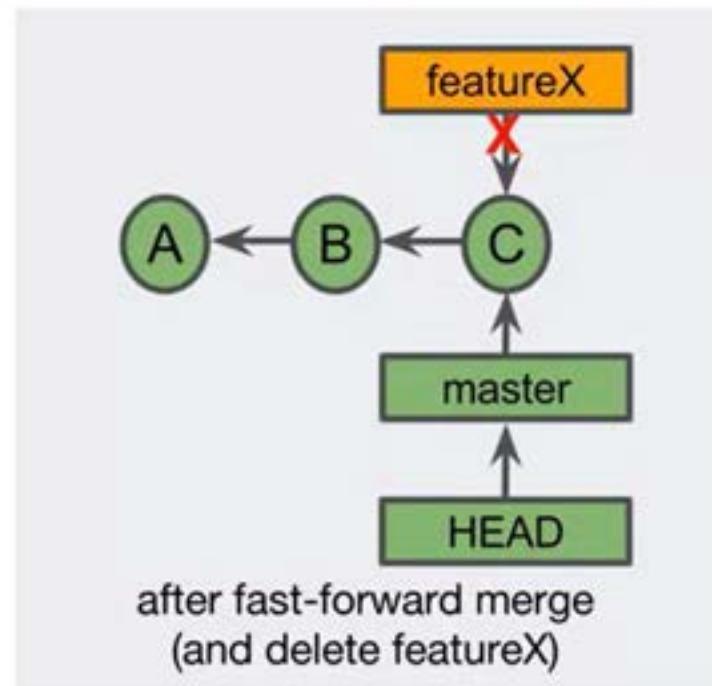
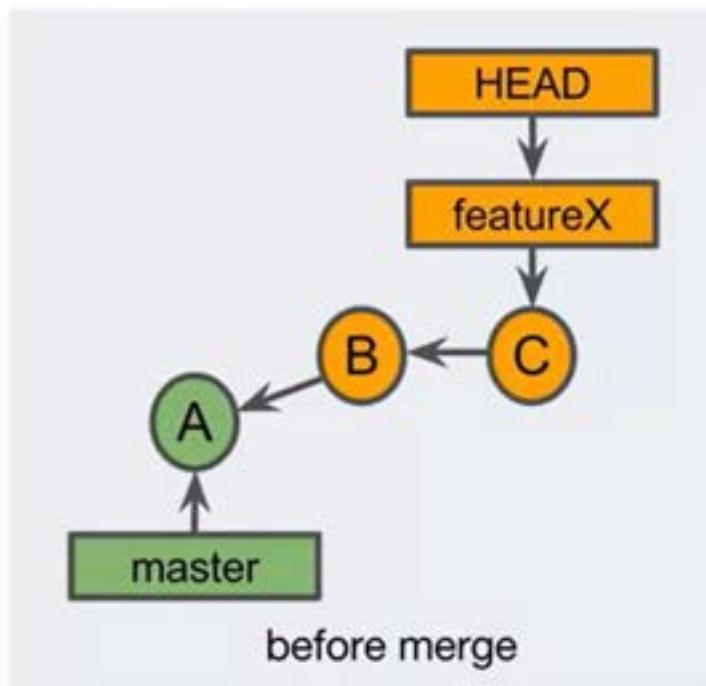
CONDITIONS FOR A FAST-FORWARD MERGE

Possible if no other commits have been made to the base branch since branching



PERFORMING A FAST-FORWARD MERGE

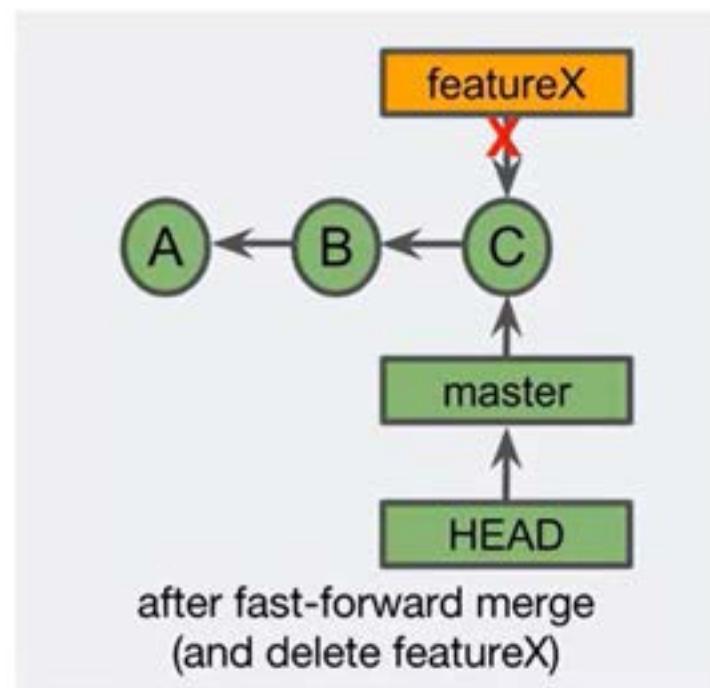
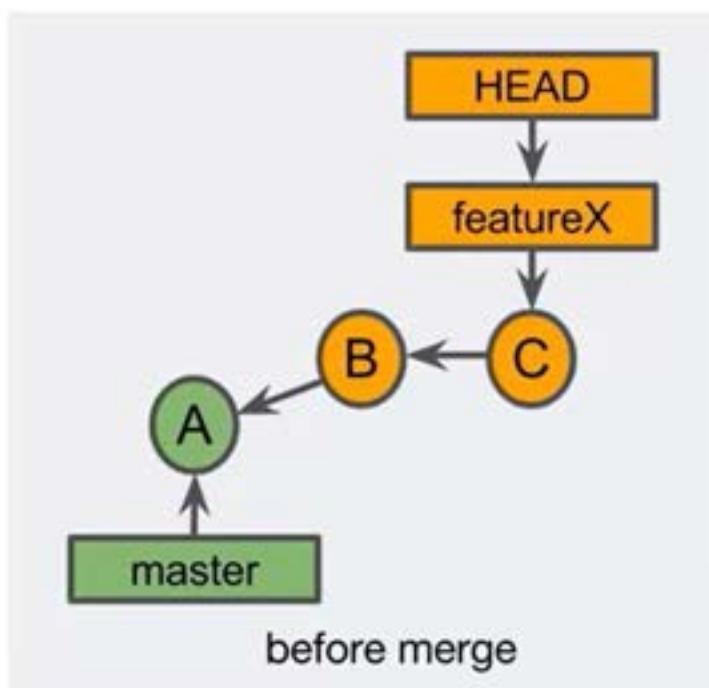
1. git checkout master
2. git merge featureX
 - attempting a fast forward merge is the default
3. git branch -d featureX



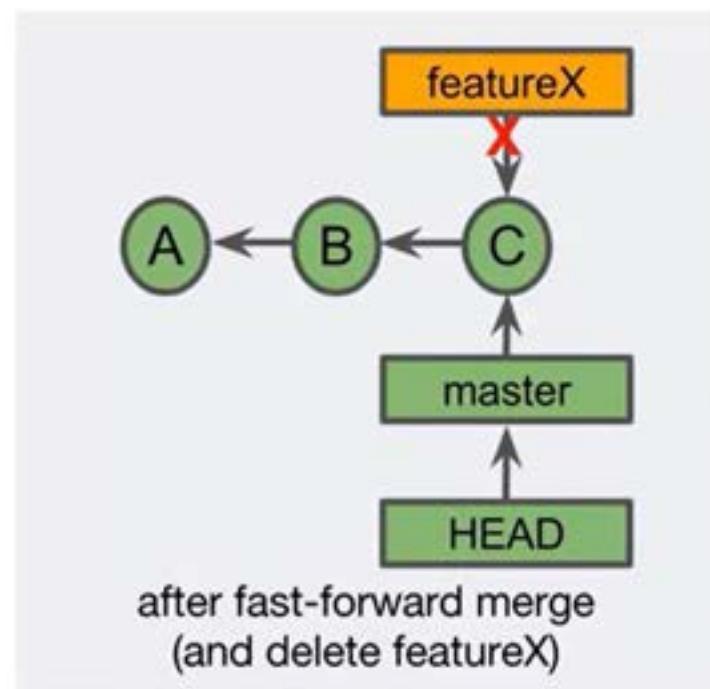
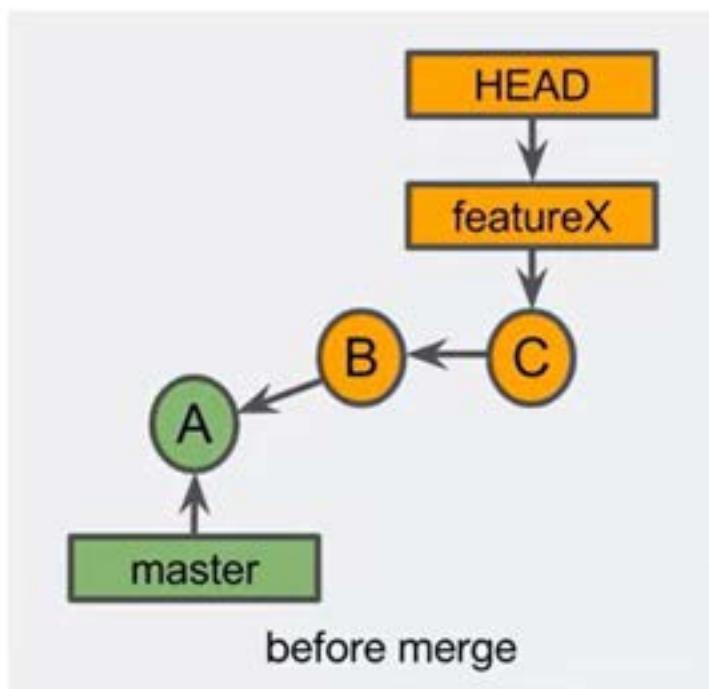
PERFORMING A FAST FORWARD MERGE

```
$ git log --oneline --graph --all
* 5046c75 (HEAD -> feature2) added feature 2
* fbadd75 (master) added feature 1
$ git checkout master
Switched to branch 'master'
$ git merge feature2
Updating fbadd75..5046c75
Fast-forward
  fileA.txt | 1 +
  1 file changed, 1 insertion(+)
$ git log --oneline --graph --all
* 5046c75 (HEAD -> master, feature2) added feature 2
* fbadd75 added feature 1
$ git branch -d feature2
Deleted branch feature2 (was 5046c75).
$ git log --oneline --graph --all
* 5046c75 (HEAD -> master) added feature 2
* fbadd75 added feature 1
```

WHY DELETE THE FEATURE BRANCH LABEL?

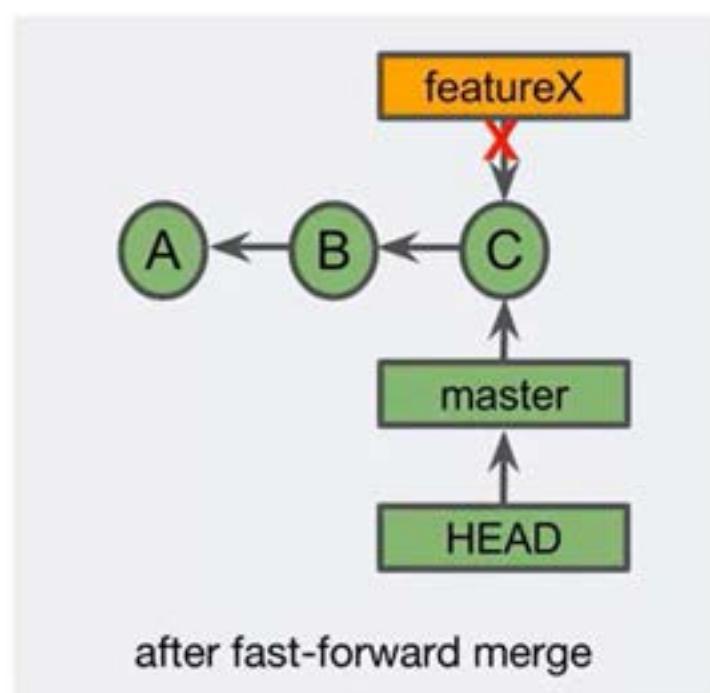
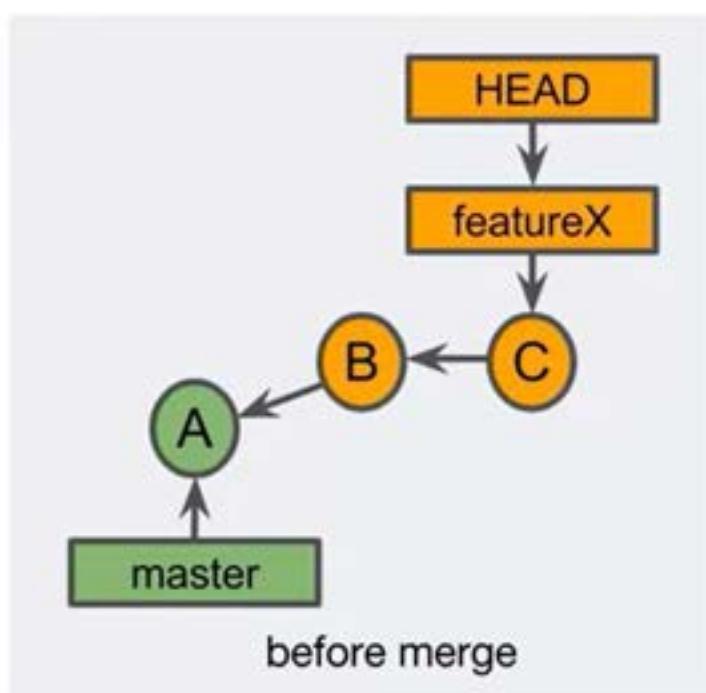


WHY DELETE THE FEATURE BRANCH LABEL?



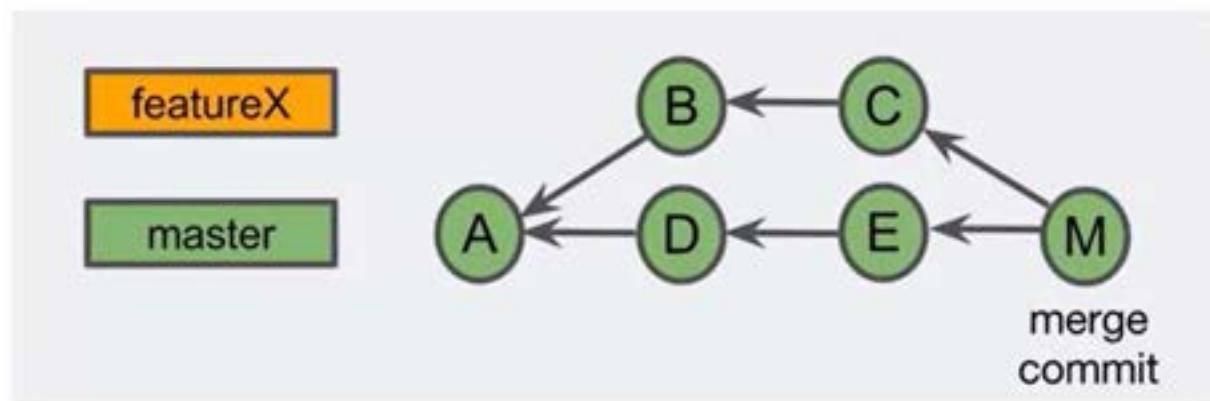
LINEAR COMMIT HISTORY

The resulting commit history is linear



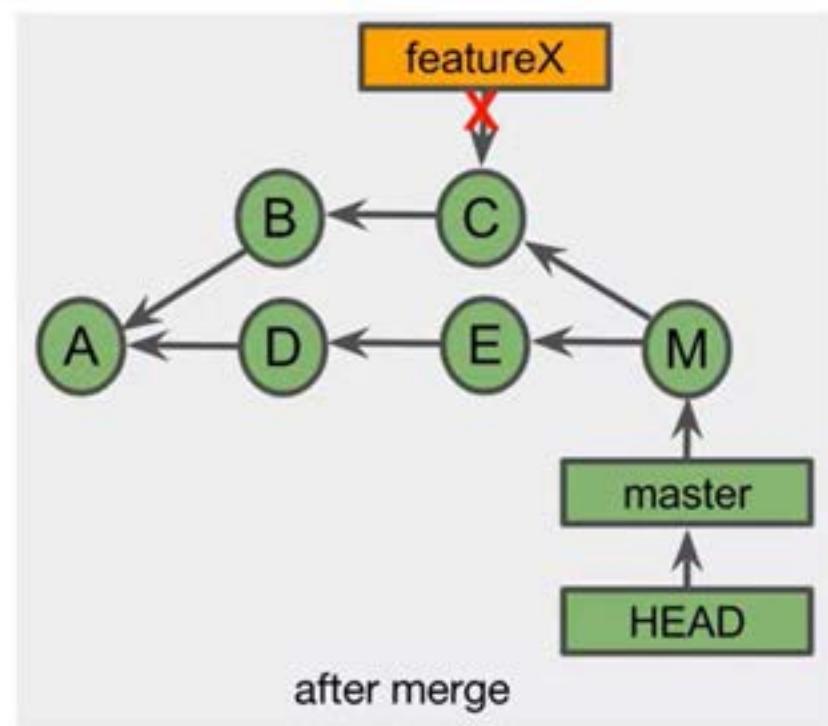
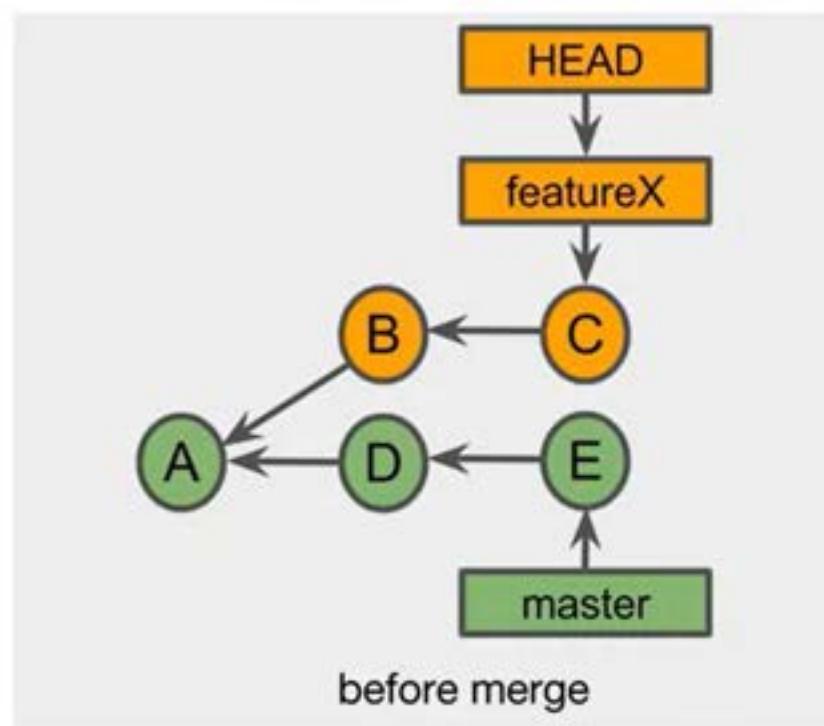
MERGE COMMIT

1. Combines the commits at the tips of the merged branches
2. Places the result in the merge commit



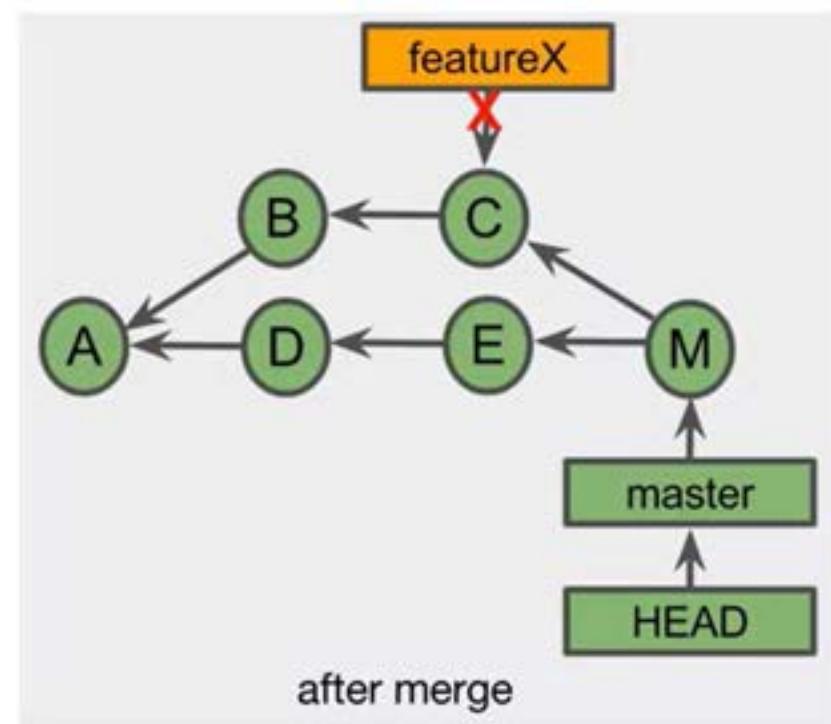
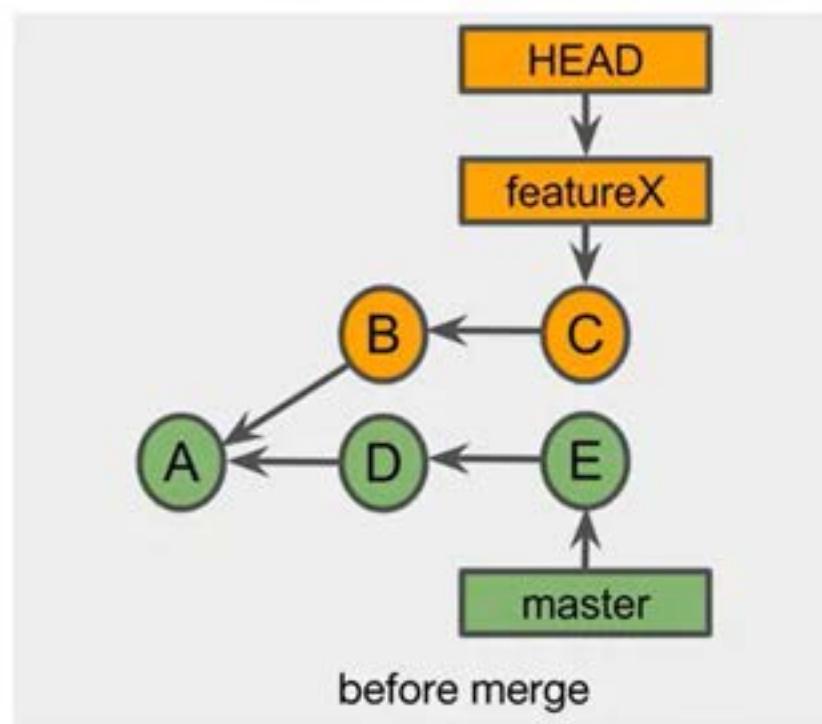
PERFORMING A MERGE COMMIT

1. git checkout master
2. git merge featureX
 - o accept or modify the **merge message**
3. git branch -d featureX



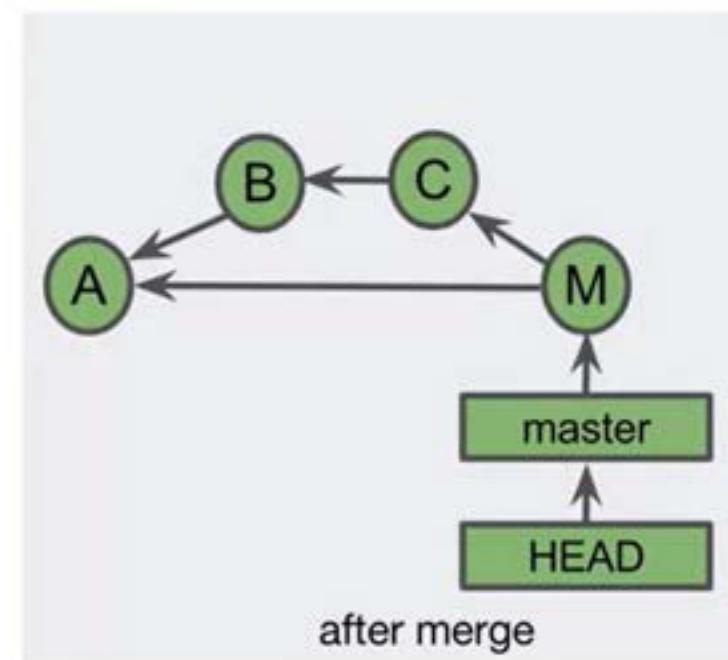
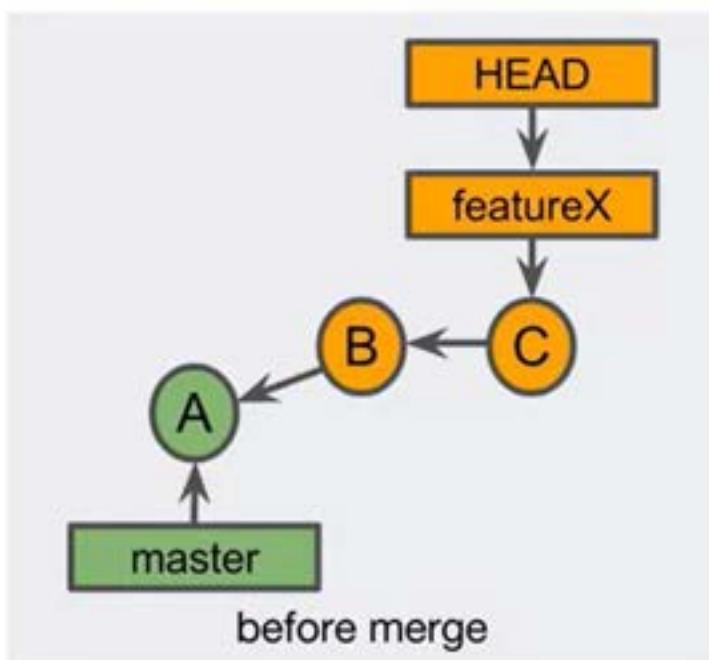
PERFORMING A MERGE COMMIT

1. git checkout master
2. git merge featureX
 - o accept or modify the **merge message**
3. git branch -d featureX



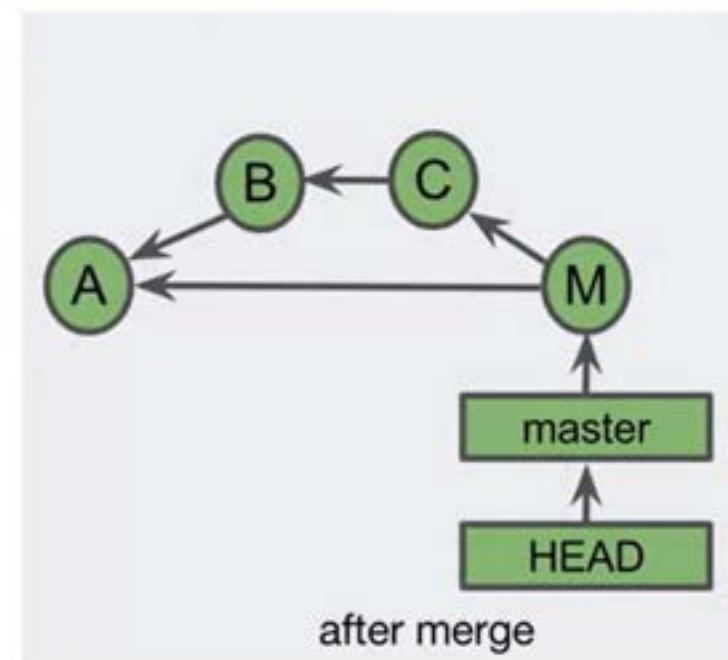
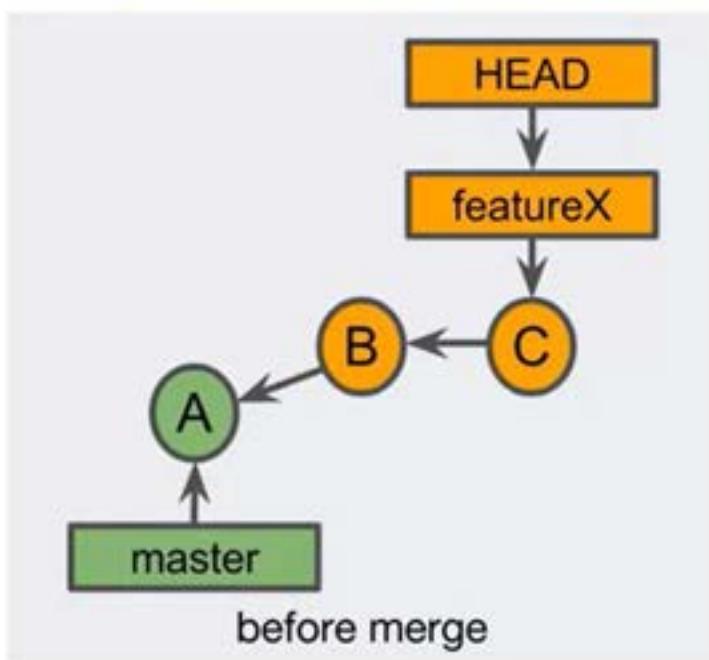
PERFORMING A MERGE COMMIT (NO FAST-FORWARD)

1. git checkout master
2. git merge --no-ff featureX
 - o accept or modify the merge message
3. git branch -d featureX



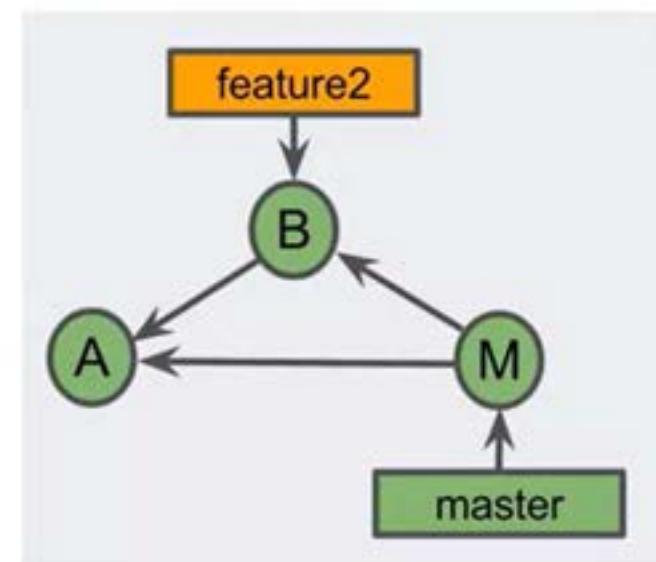
PERFORMING A MERGE COMMIT (NO FAST-FORWARD)

1. git checkout master
2. git merge --no-ff featureX
 - o accept or modify the merge message
3. git branch -d featureX



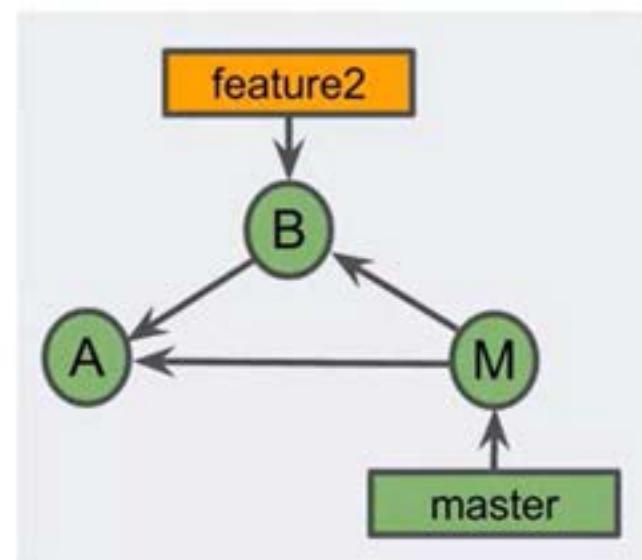
PERFORMING A MERGE WITH A MERGE COMMIT

```
$ git log --oneline --graph --all
* 804772b (HEAD -> feature2) added feature 2
* 9232ald (master) added feature 1
$ git checkout master
Switched to branch 'master'
$ git merge --no-ff feature2
# opens editor to edit merge message
Merge made by the 'recursive' strategy.
 fileA.txt | 1 +
 1 file changed, 1 insertion(+)
$ git log --oneline --graph --all
* efaa6ff (HEAD -> master) Merge branch 'feature2'
|\ \
| * 804772b (feature2) added feature 2
|/
* 9232ald added feature 1
```



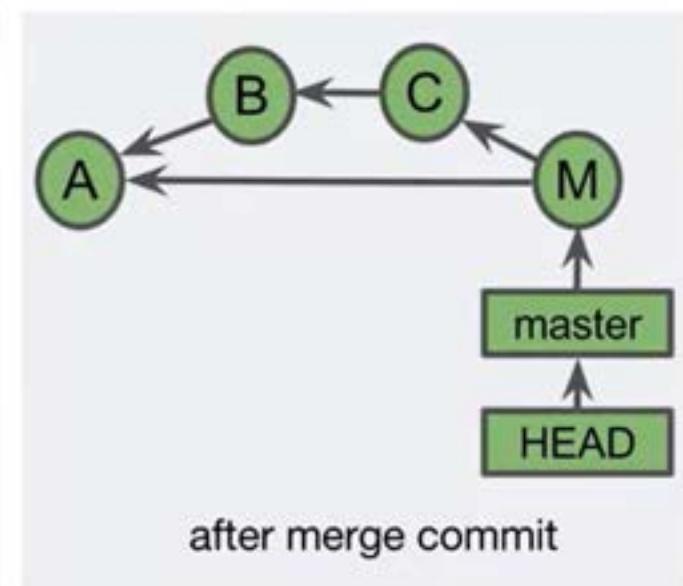
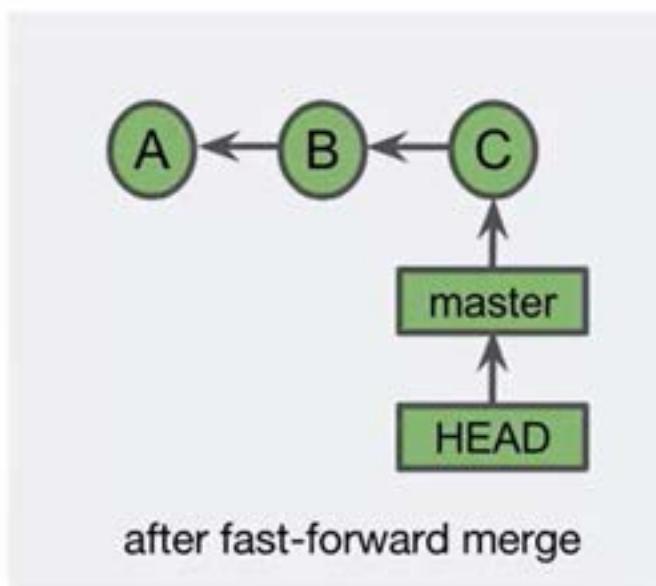
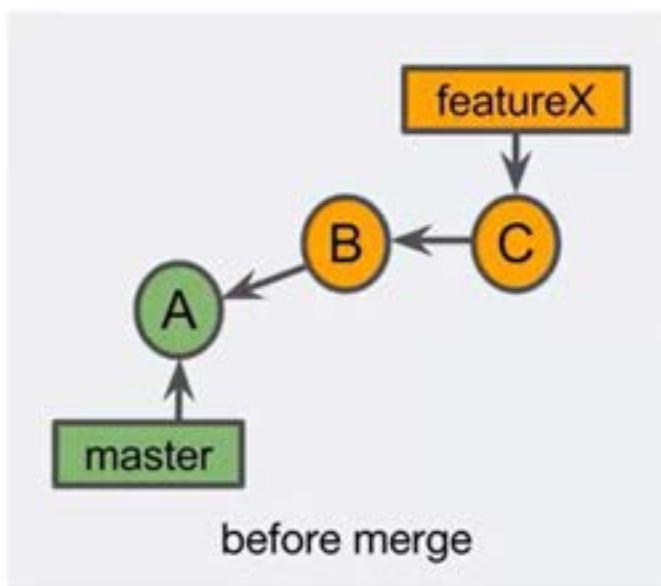
PERFORMING A MERGE WITH A MERGE COMMIT

```
$ git log --oneline --graph --all
* 804772b (HEAD -> feature2) added feature 2
* 9232ald (master) added feature 1
$ git checkout master
Switched to branch 'master'
$ git merge --no-ff feature2
# opens editor to edit merge message
Merge made by the 'recursive' strategy.
 fileA.txt | 1 +
 1 file changed, 1 insertion(+)
$ git log --oneline --graph --all
* efaa6ff (HEAD -> master) Merge branch 'feature2'
|\ \
| * 804772b (feature2) added feature 2
|/
* 9232ald added feature 1
$ git branch -d feature2
Deleted branch feature2 (was 804772b).
```



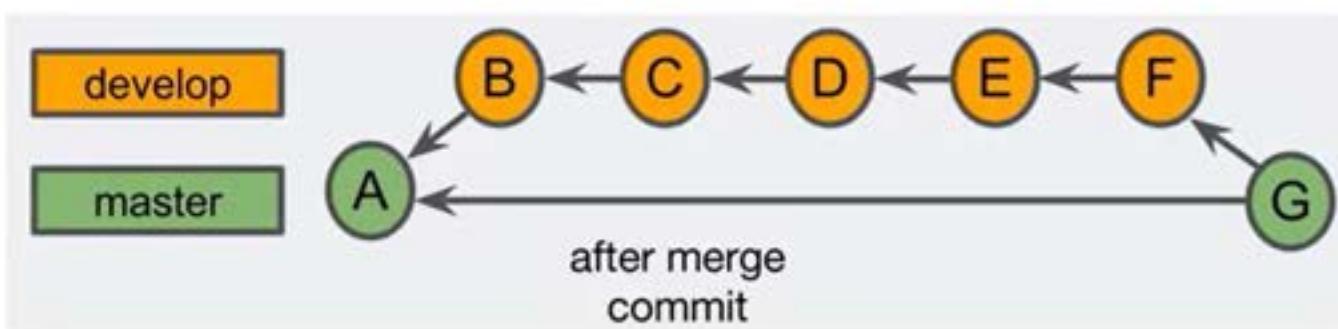
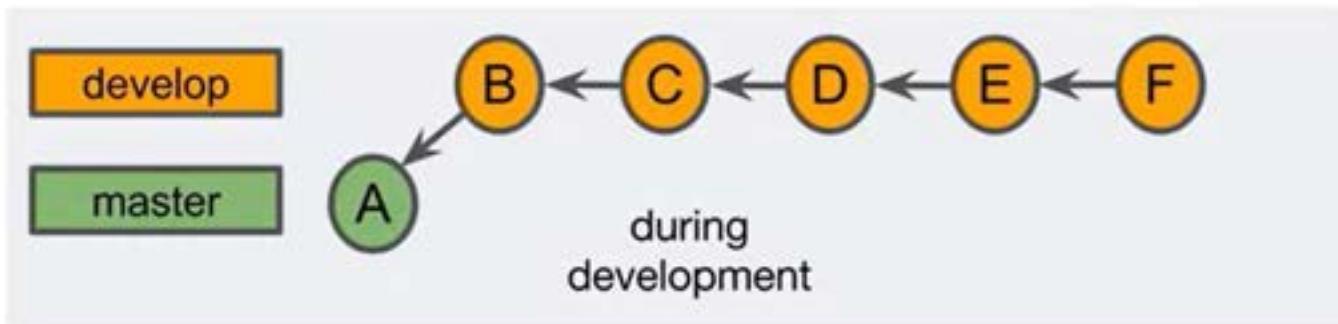
TEAM COMMIT HISTORY POLICIES

- Require a linear history
- Require merge commits
- Let the person merging decide



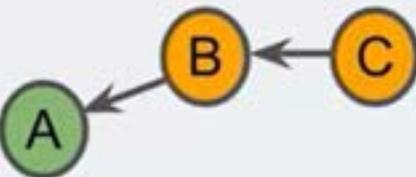
MERGING LONG-RUNNING BRANCHES

These merges are typically easy because they are fast-forwardable

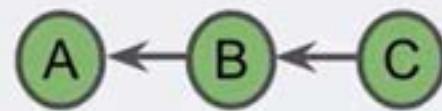


REVIEW

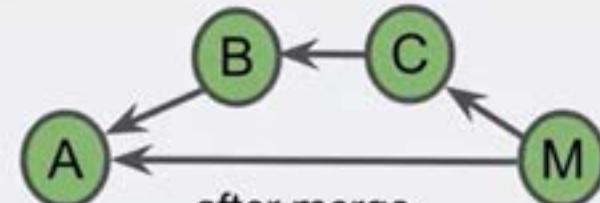
- Merging combines the work of multiple branches
- A fast-forward merge moves the base branch label to the tip of the topic branch
- A merge is fast-forwardable if no other commits have been made to the base branch since branching
- A merge commit is the result of combining the work of multiple commits
- A merge commit has multiple parents



before merge



after fast-forward
merge



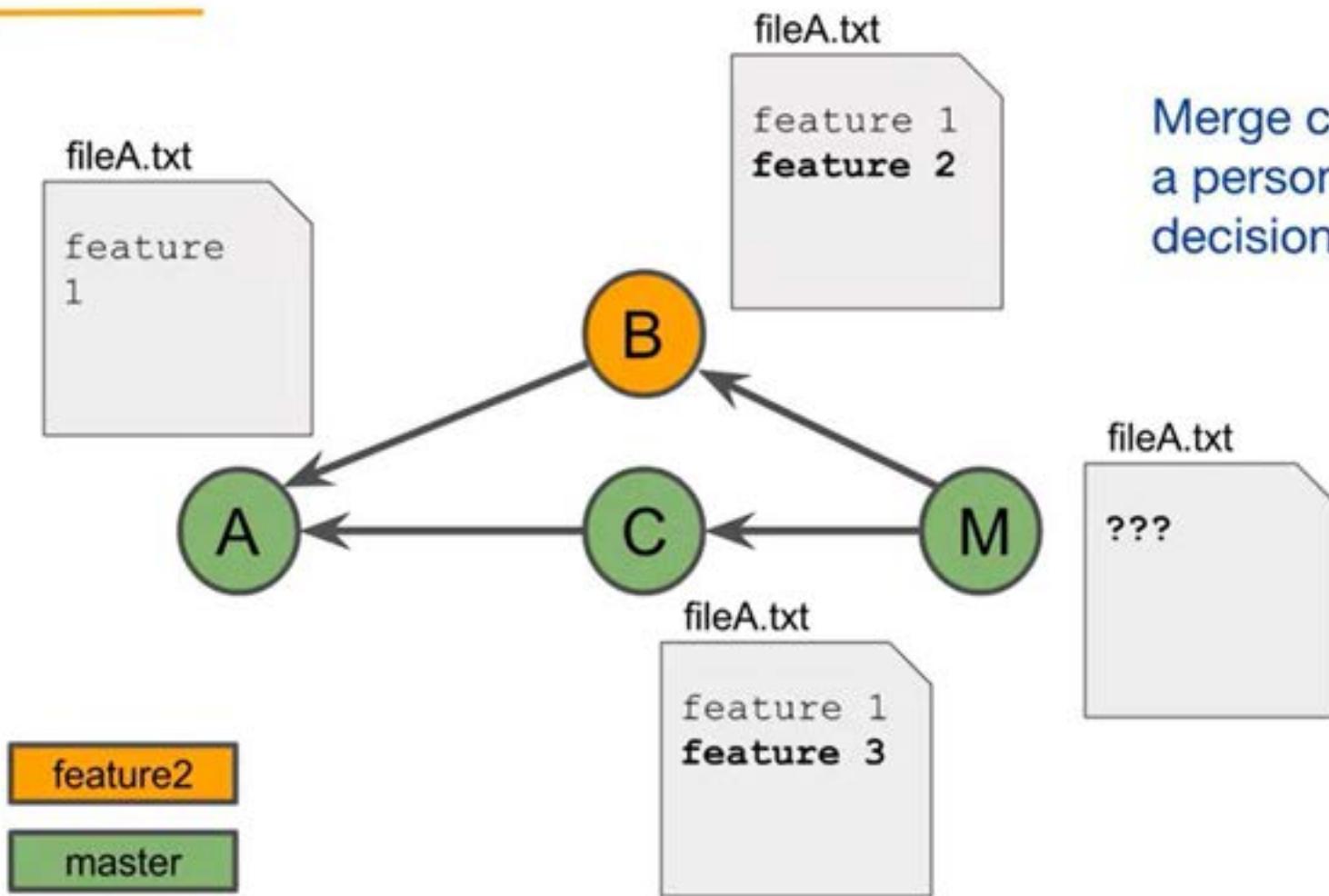
after merge
commit

Resolving Merge Conflicts

Using Sourcetree

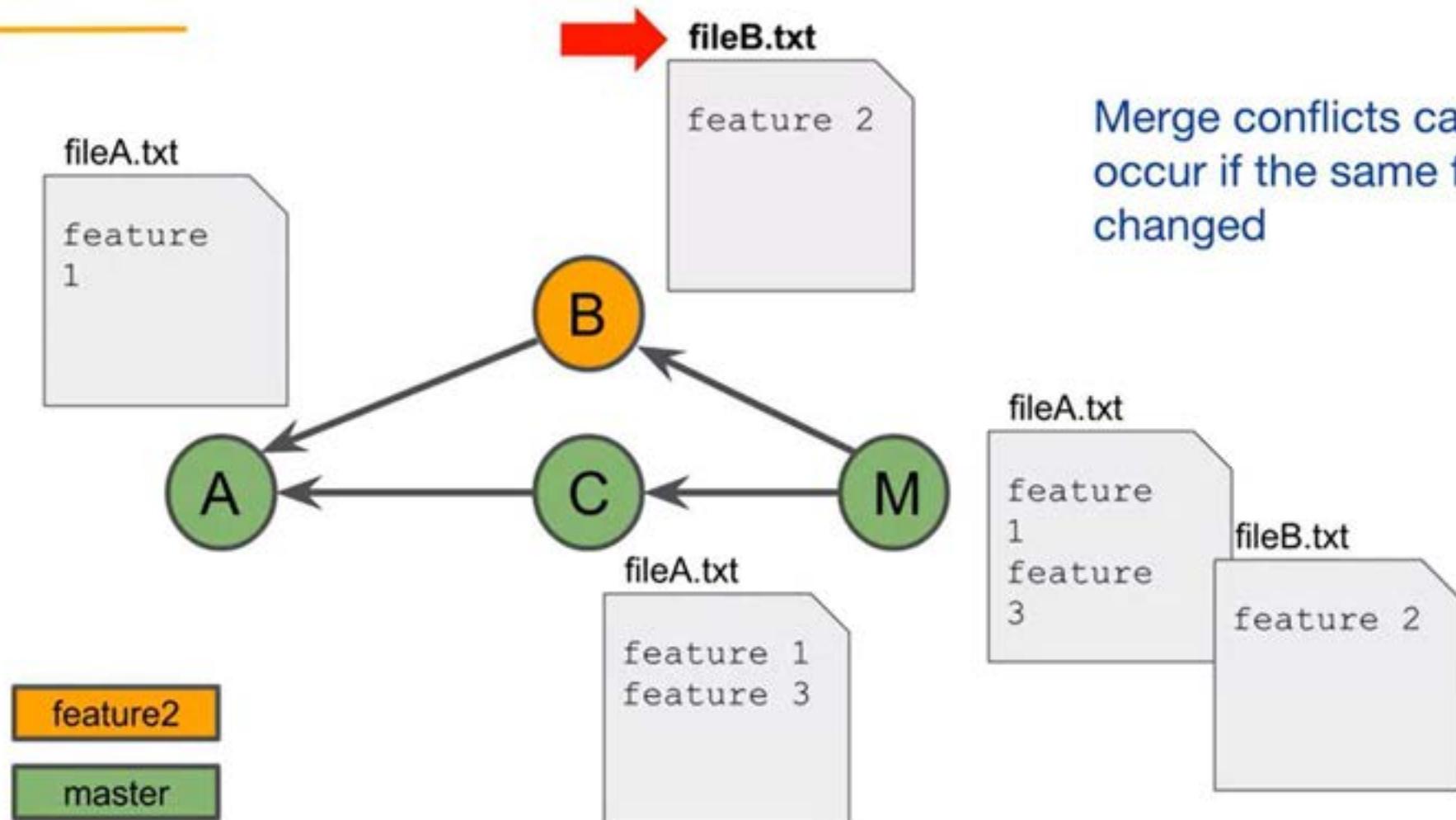
Copyright © 2018 Atlassian

MERGE CONFLICTS

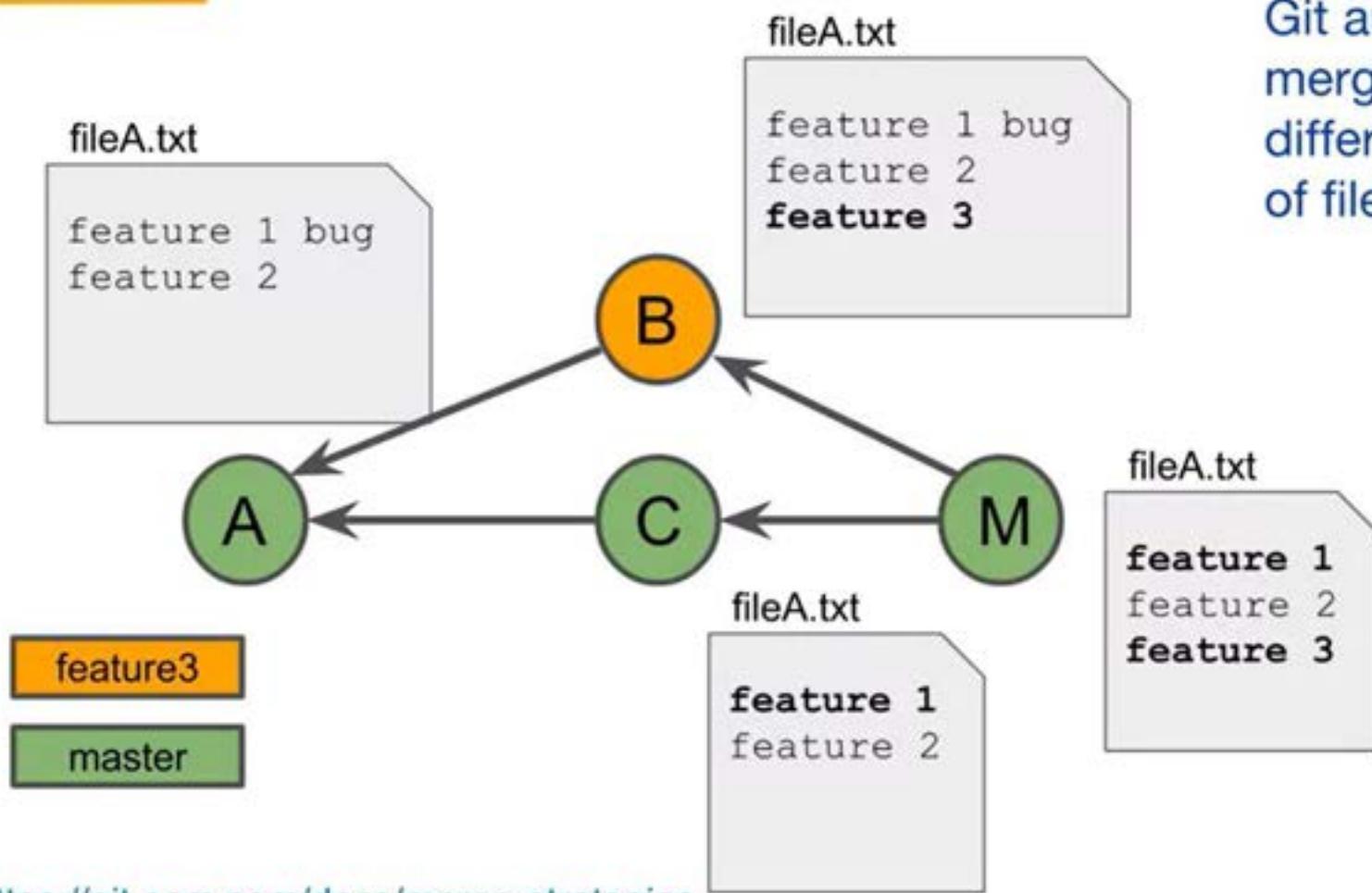


Merge conflicts occur when a person needs to make a decision

NOT A MERGE CONFLICT- DIFFERENT FILES



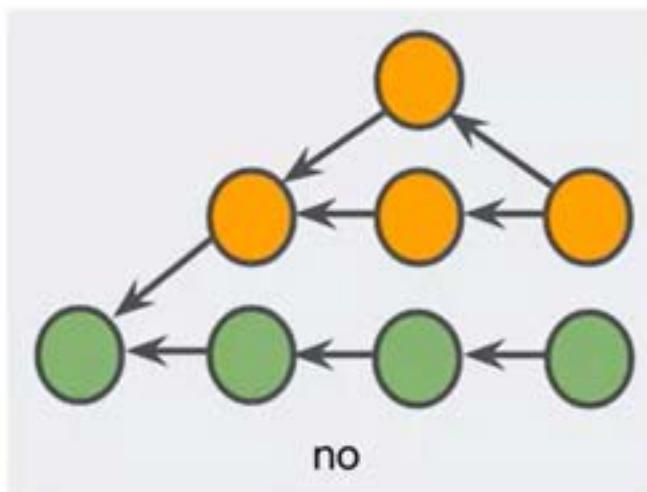
NOT A MERGE CONFLICT- DIFFERENT HUNKS



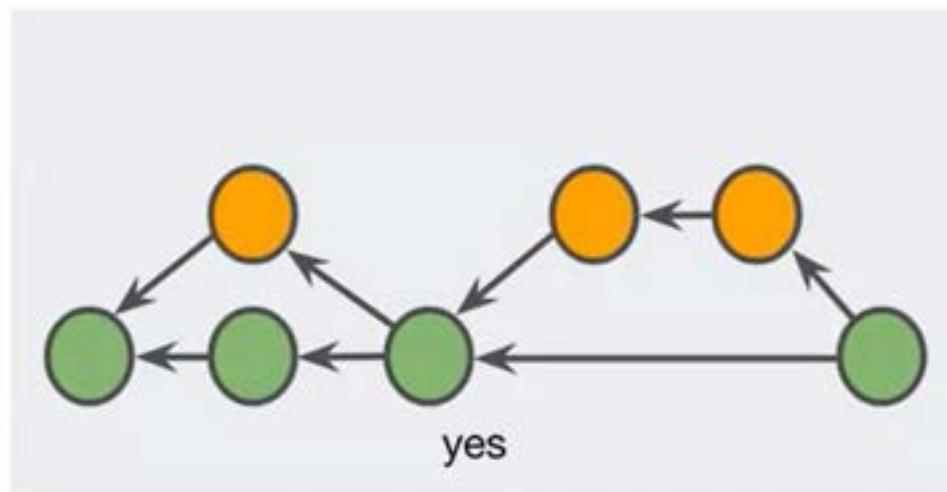
Git automatically merges changes to different parts (hunks) of files

AVOIDING MERGE CONFLICTS

- Git merges are usually quite easy
- Small, frequent merges are the easiest



no



yes

Topics

Merge conflict overview

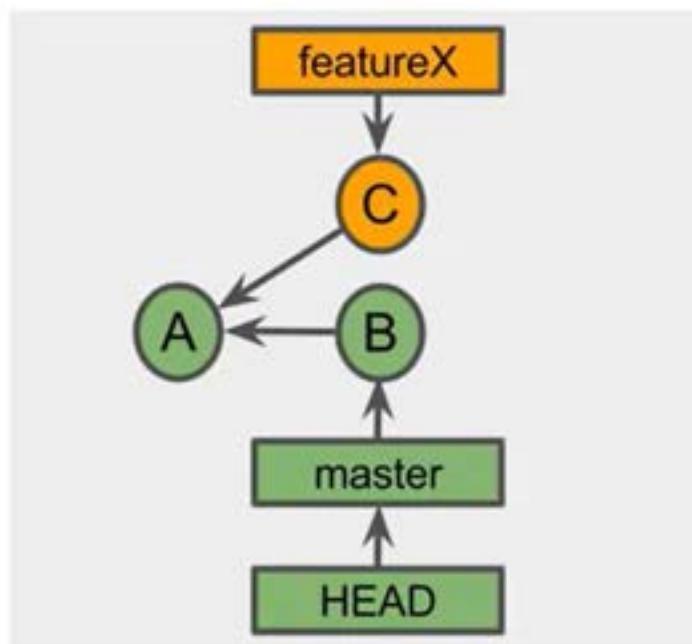
Resolving a merge conflict

Aborting a merge attempt

RESOLVING A MERGE CONFLICT

Involves three commits:

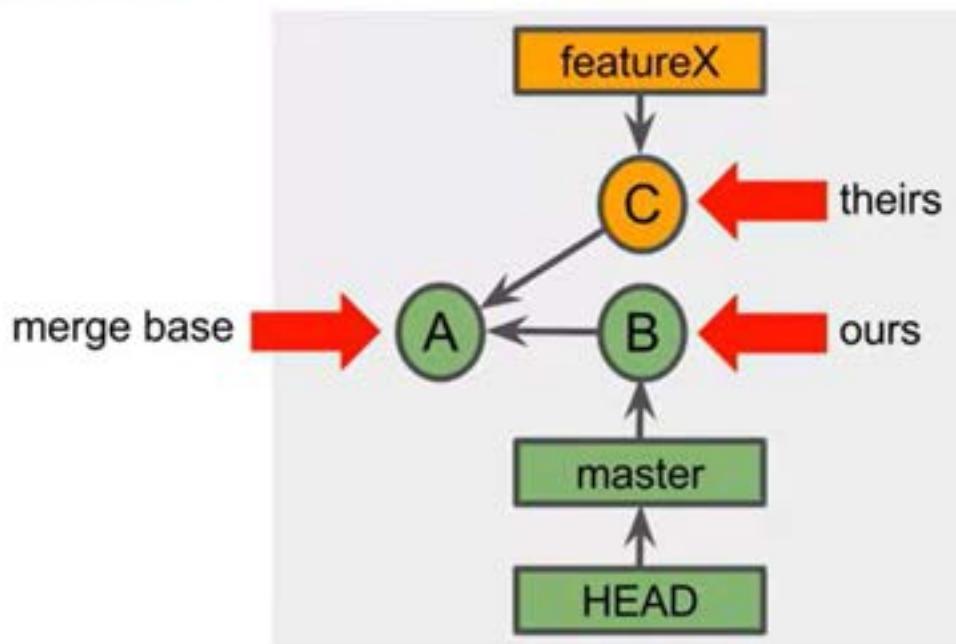
- 1.
- 2.
- 3.



RESOLVING A MERGE CONFLICT

Involves three commits:

1. The tip of the current branch (B)- "ours" or "mine"
2. The tip of the branch to be merged (C)- "theirs"
3. A common ancestor (A)- "merge base"

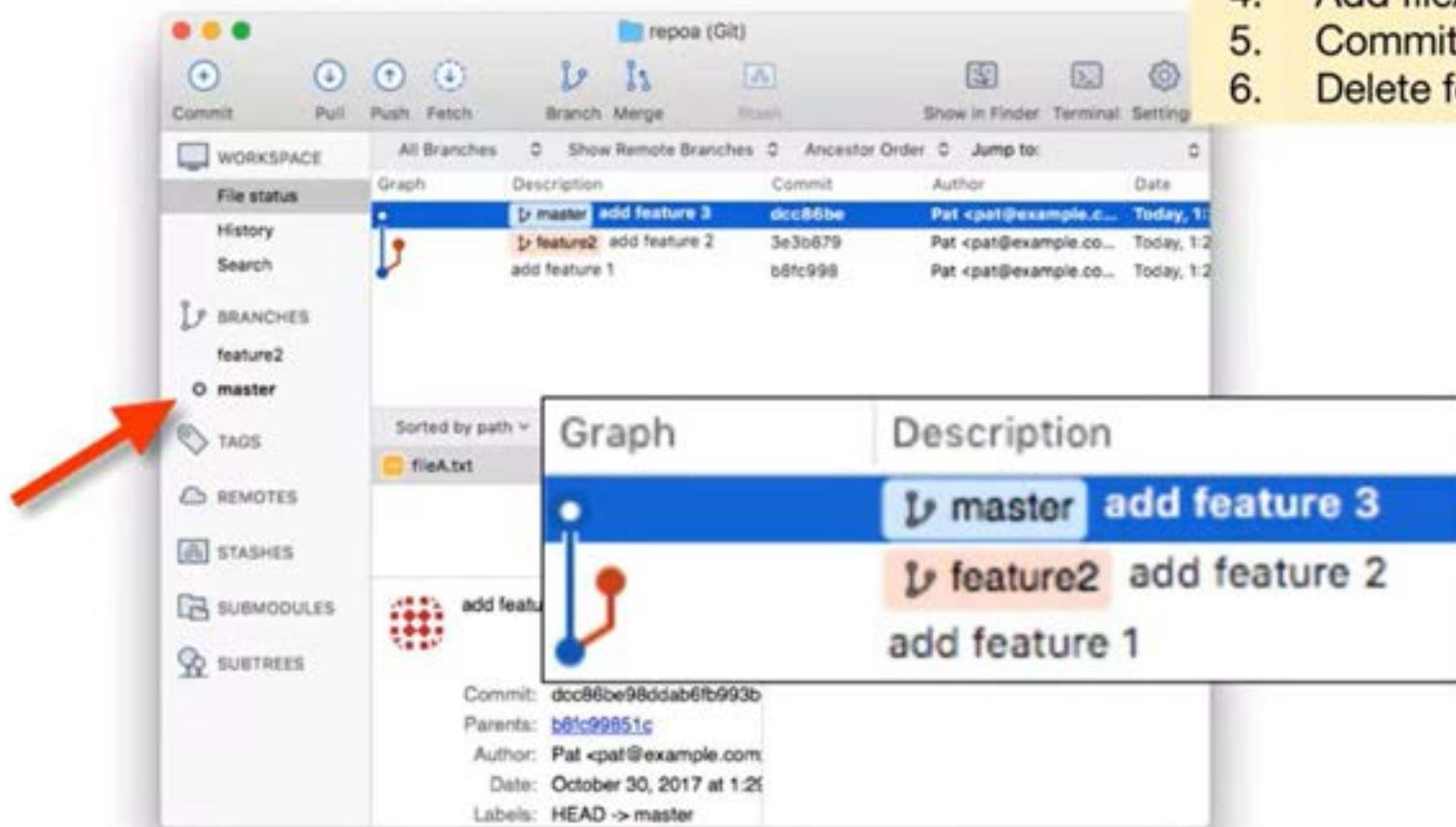


BASIC STEPS TO RESOLVE A MERGE CONFLICT

1. Checkout master
2. Merge featureX
 - a. CONFLICT- Both modified fileA.txt
3. Fix fileA.txt
4. Stage fileA.txt
5. Commit the merge commit
6. Delete the featureX branch label

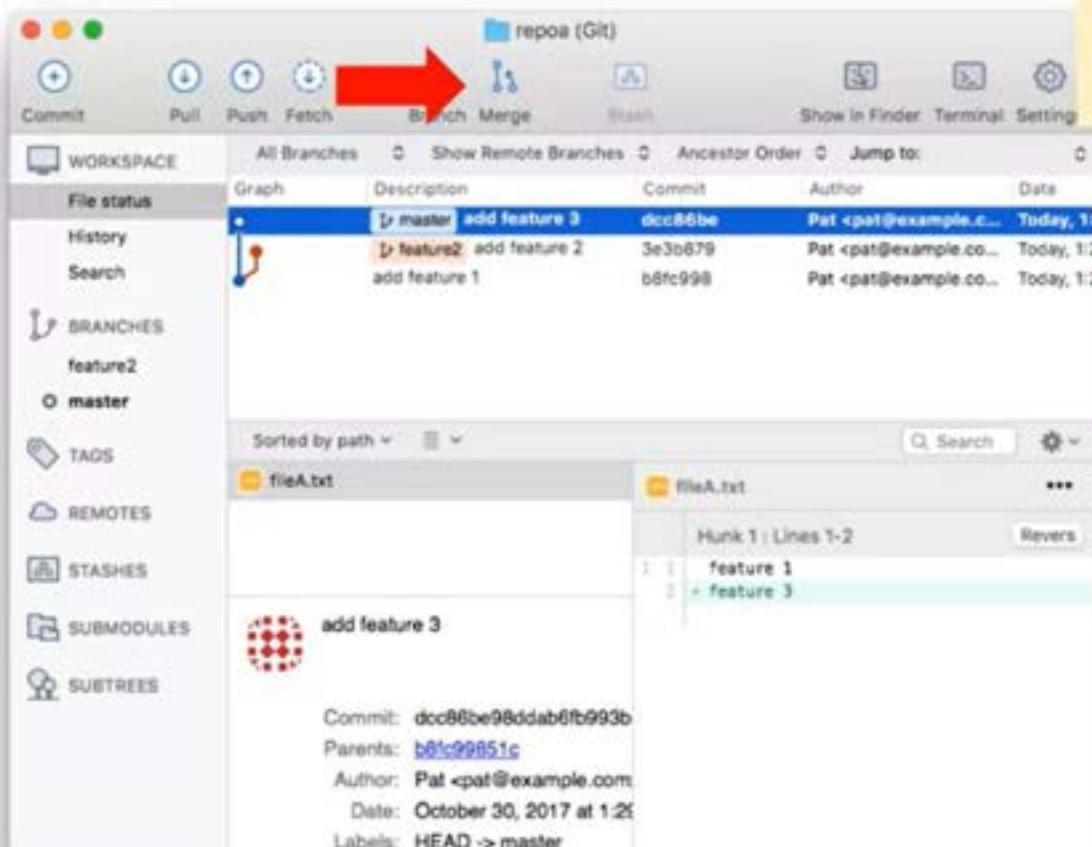
When attempting a merge, files with conflicts are **modified by Git** and placed in the working tree

MERGE CONFLICT- CHECKOUT MASTER



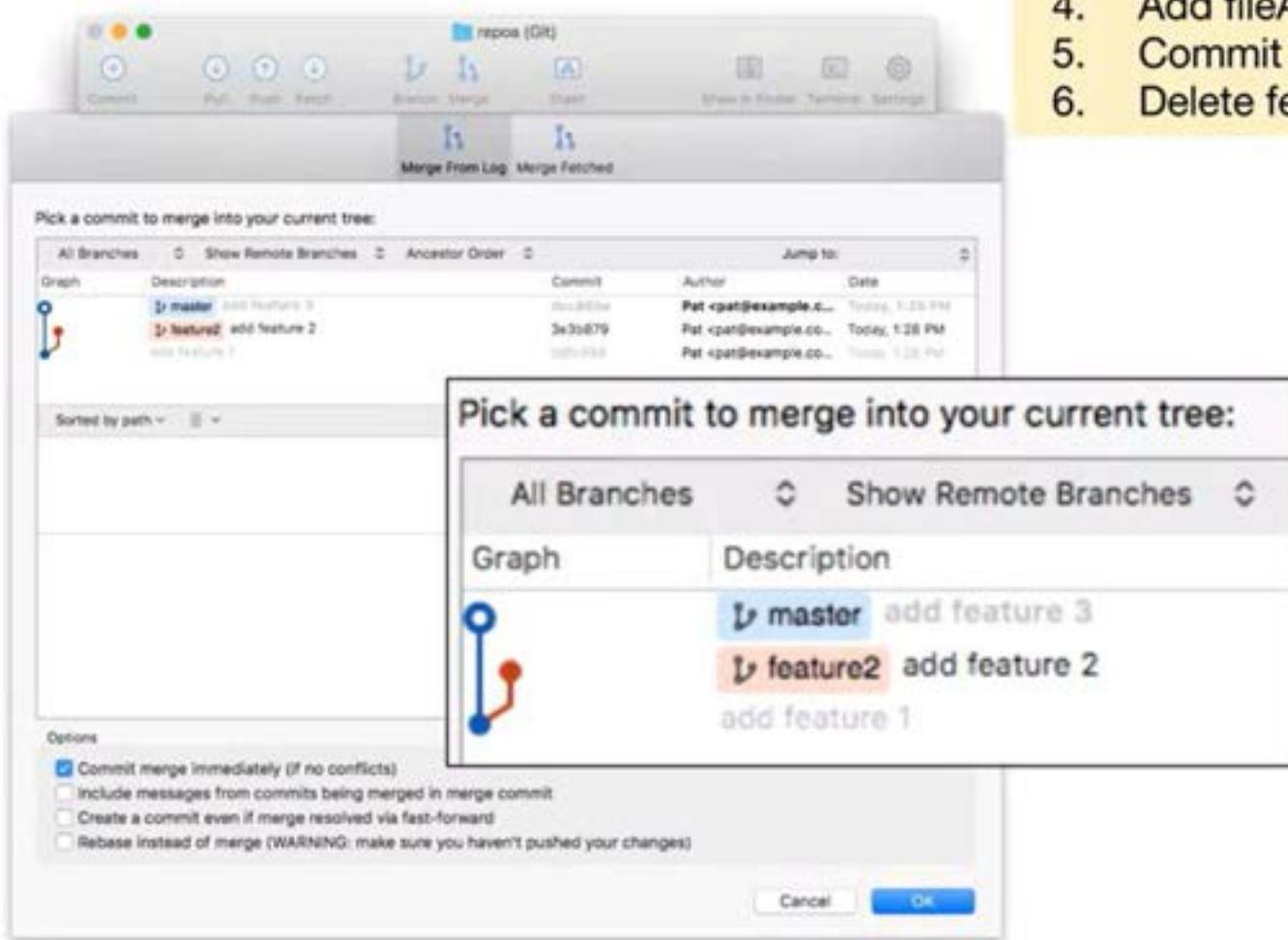
1. Checkout master
2. Merge featureX
3. Fix fileA.txt
4. Add fileA.txt
5. Commit the merge
6. Delete featureX

MERGE CONFLICT- MERGE FEATURE BRANCH



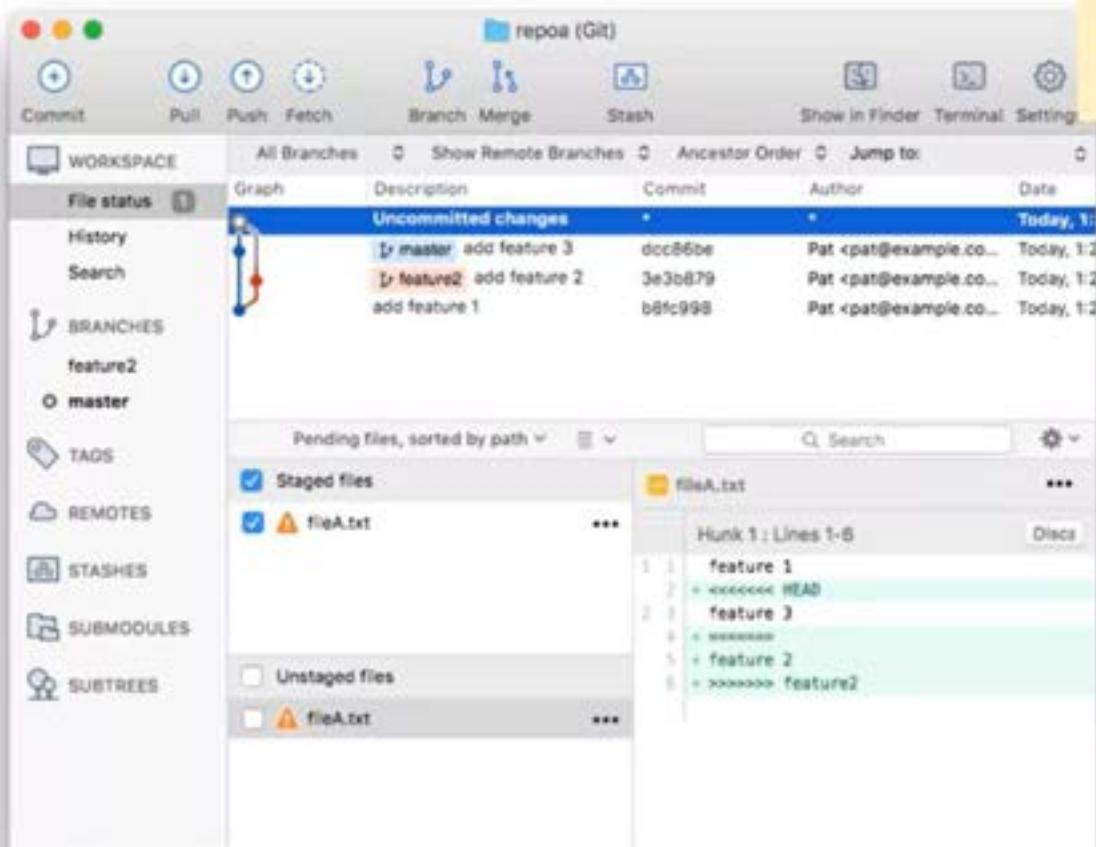
1. Checkout master
2. **Merge featureX**
3. Fix fileA.txt
4. Add fileA.txt
5. Commit the merge
6. Delete featureX

MERGE CONFLICT- MERGE FEATURE BRANCH



1. Checkout master
2. **Merge featureX**
3. Fix fileA.txt
4. Add fileA.txt
5. Commit the merge
6. Delete featureX

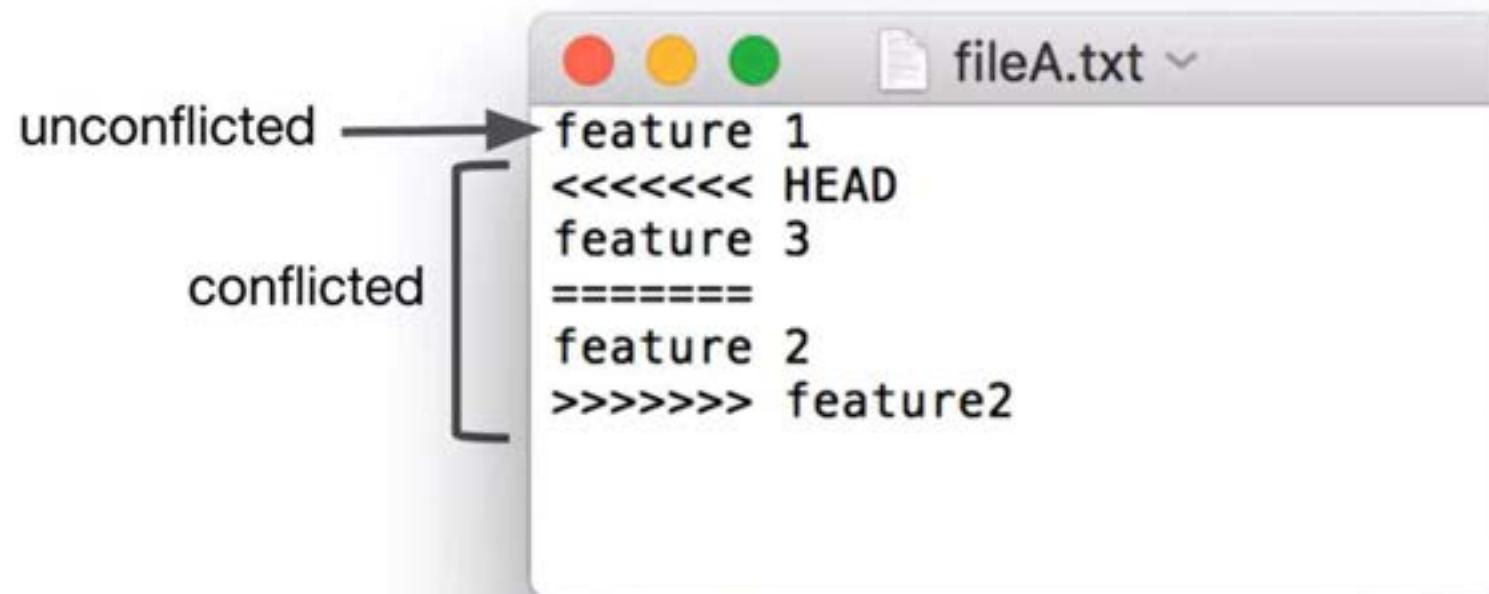
MERGE CONFLICT- VIEW MODIFIED FILE(S)



1. Checkout master
2. **Merge featureX**
3. Fix fileA.txt
4. Add fileA.txt
5. Commit the merge
6. Delete featureX

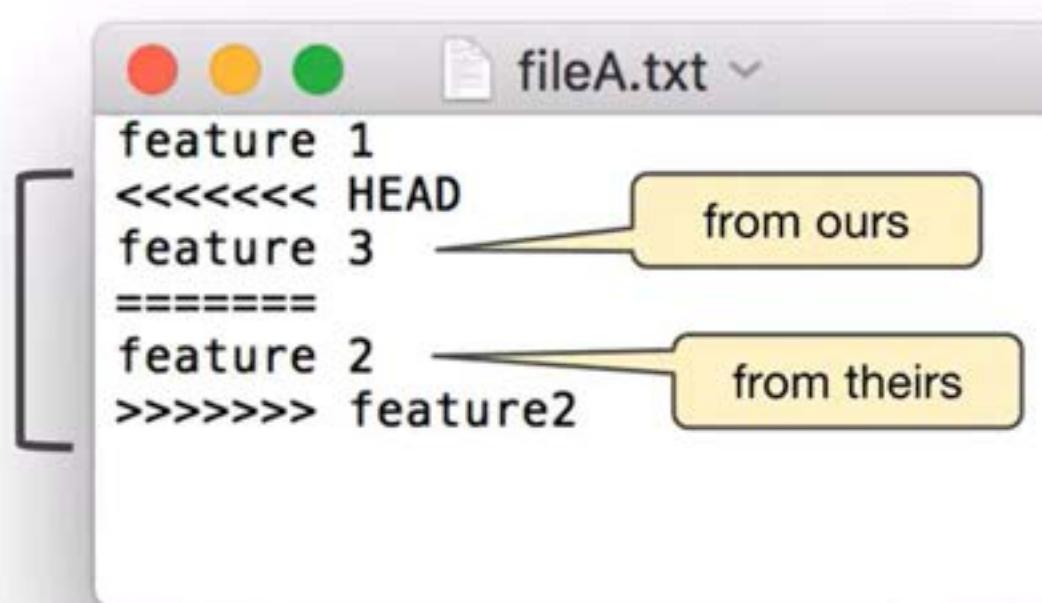
CONFLICTED HUNKS

Conflicted hunks are surrounded by conflict markers <<<<< and >>>>>



READING CONFLICT MARKERS

- Text from the HEAD commit is between <<<<< and =====
- Text from the branch to be merged is between ===== and >>>>>



FIXING A CONFLICTED FILE

1. Checkout master
2. Merge featureX
3. **Fix fileA.txt**
4. Add fileA.txt
5. Commit the merge
6. Delete featureX

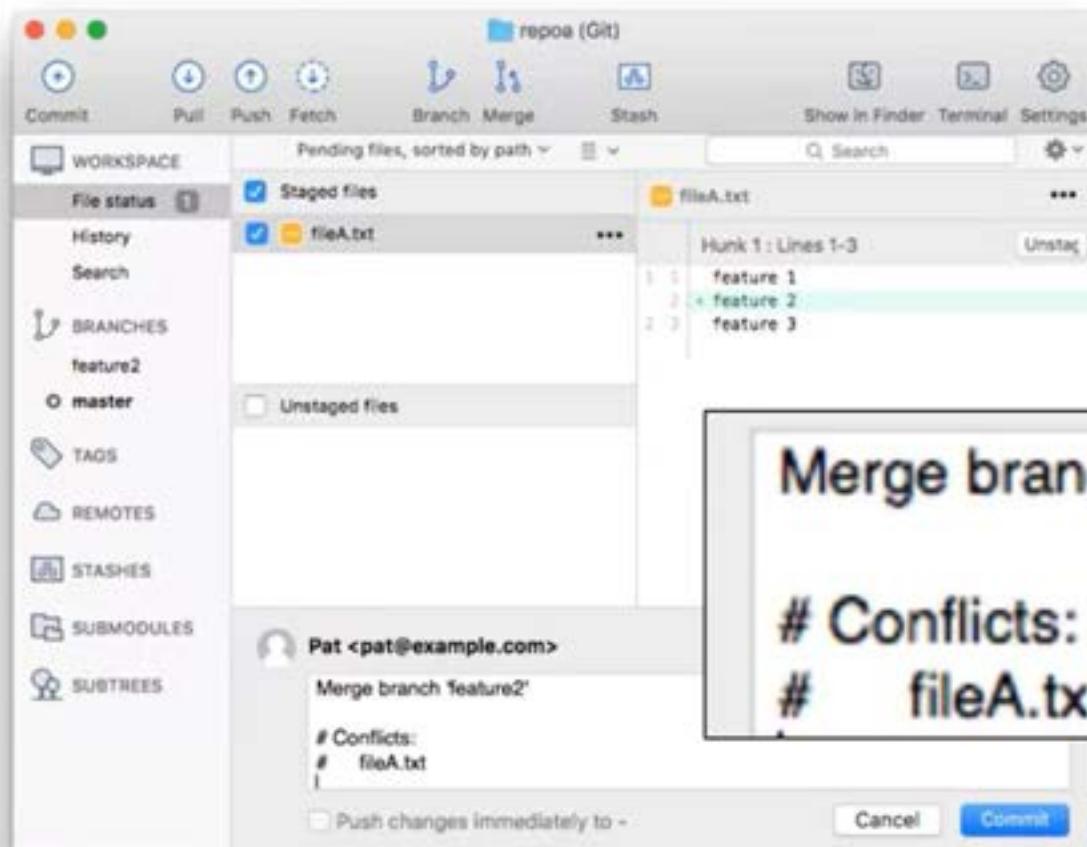
```
feature 1
<<<<< HEAD
feature 3
=====
feature 2
>>>>> feature2
```

before fixing a file

```
feature 1
feature 2
feature 3
```

after fixing a file

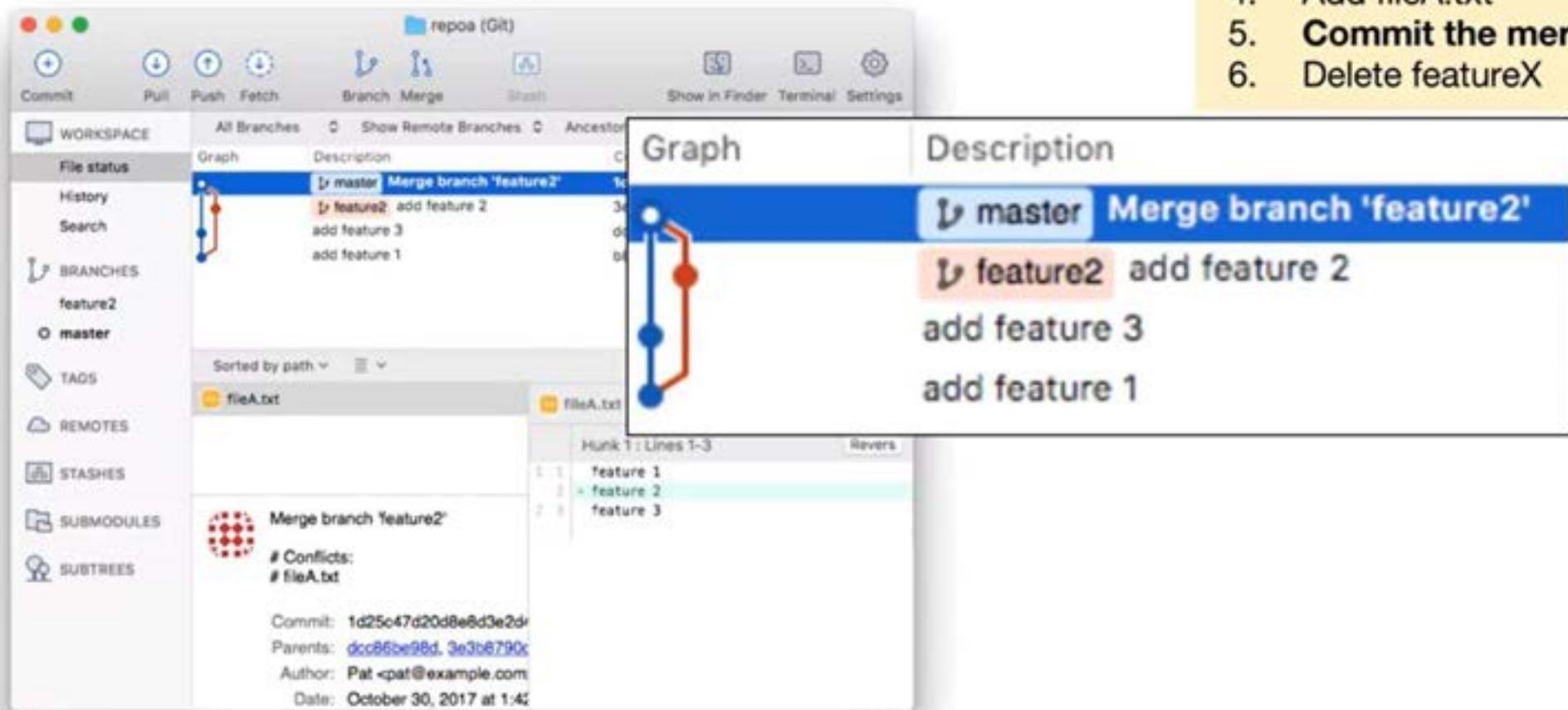
MERGE CONFLICT- SPECIFY MERGE MESSAGE



1. Checkout master
2. Merge featureX
3. Fix fileA.txt
4. Add fileA.txt
5. **Commit the merge**
6. Delete featureX

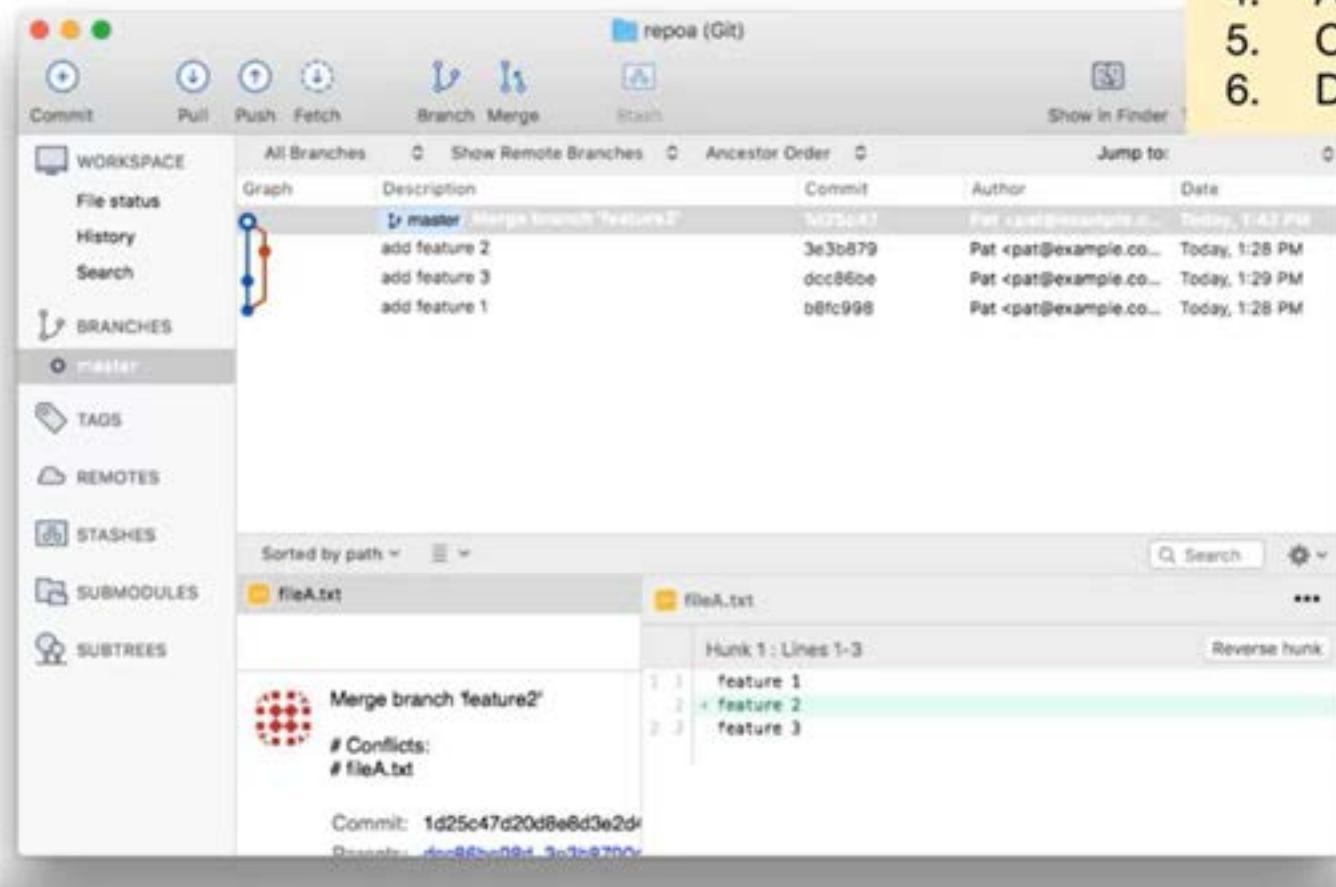
MERGE CONFLICT- COMMIT

1. Checkout master
2. Merge featureX
3. Fix fileA.txt
4. Add fileA.txt
- 5. Commit the merge**
6. Delete featureX

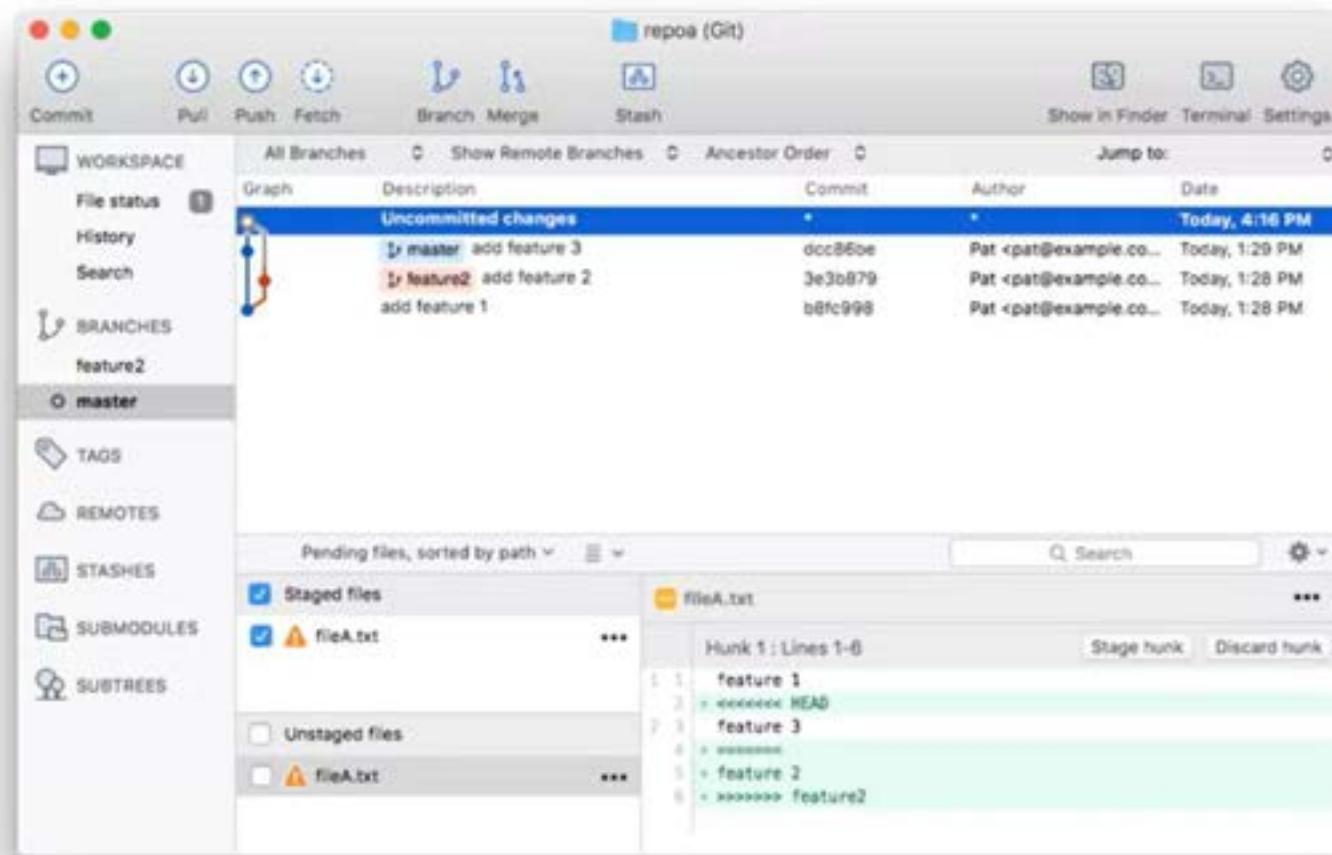


MERGE CONFLICT- FINISHED

1. Checkout master
2. Merge featureX
3. Fix fileA.txt
4. Add fileA.txt
5. Commit the merge
6. Delete featureX

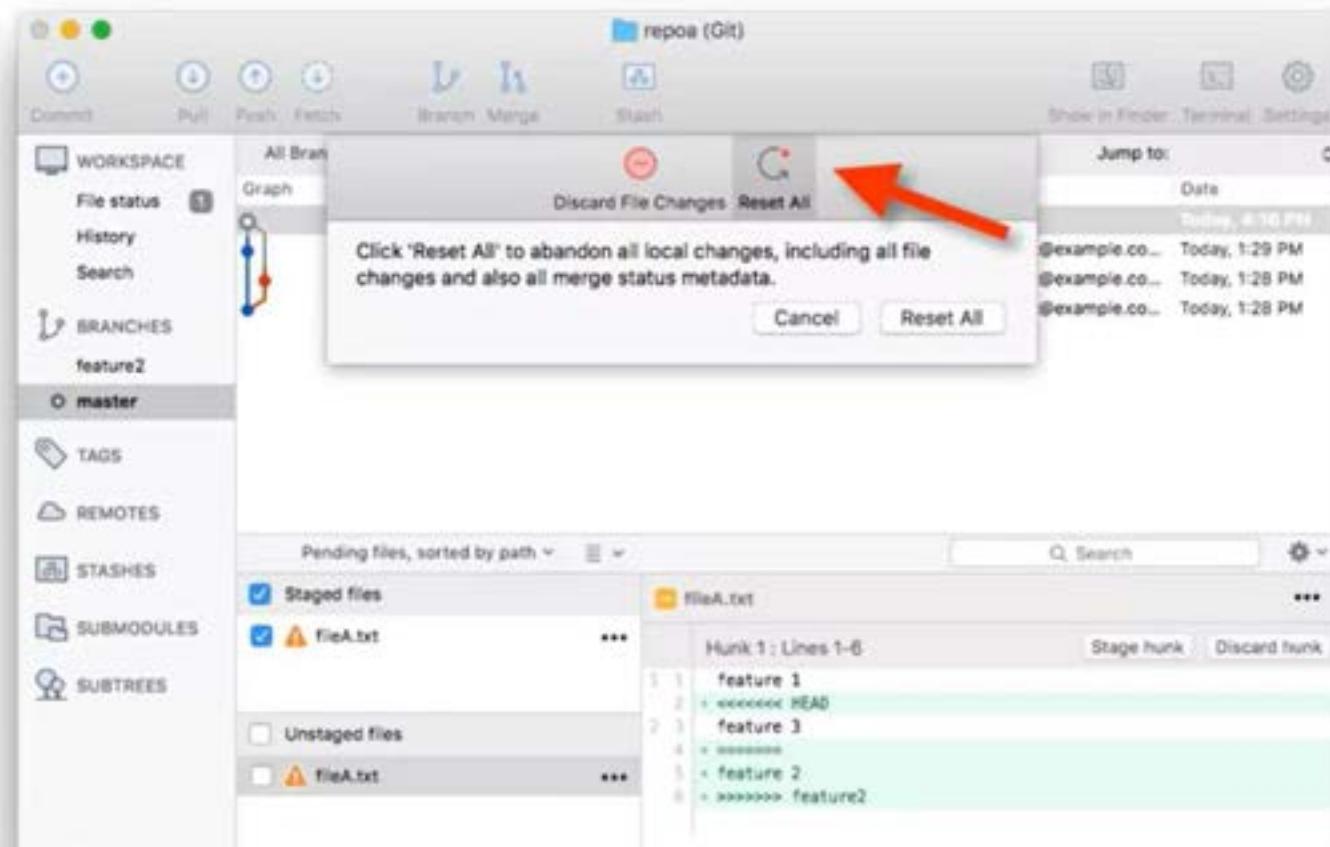


ABORTING A MERGE ATTEMPT (1 OF 3)



Repository	Actions	Window
Repository Settings...	⇧⌘R	
Refresh	⌘R	
Refresh Remote Status	⌃⌘R	
Commit...	⇧⌘C	
Commit Selected...		
Reset...	⇧⌘R	
Stash Changes...	⇧⌘S	
Push...	⇧⌘P	
Pull...	⇧⌘L	
Fetch...	⇧⌘F	
Update	⇧⌘U	
Checkout...	⌃⌘U	

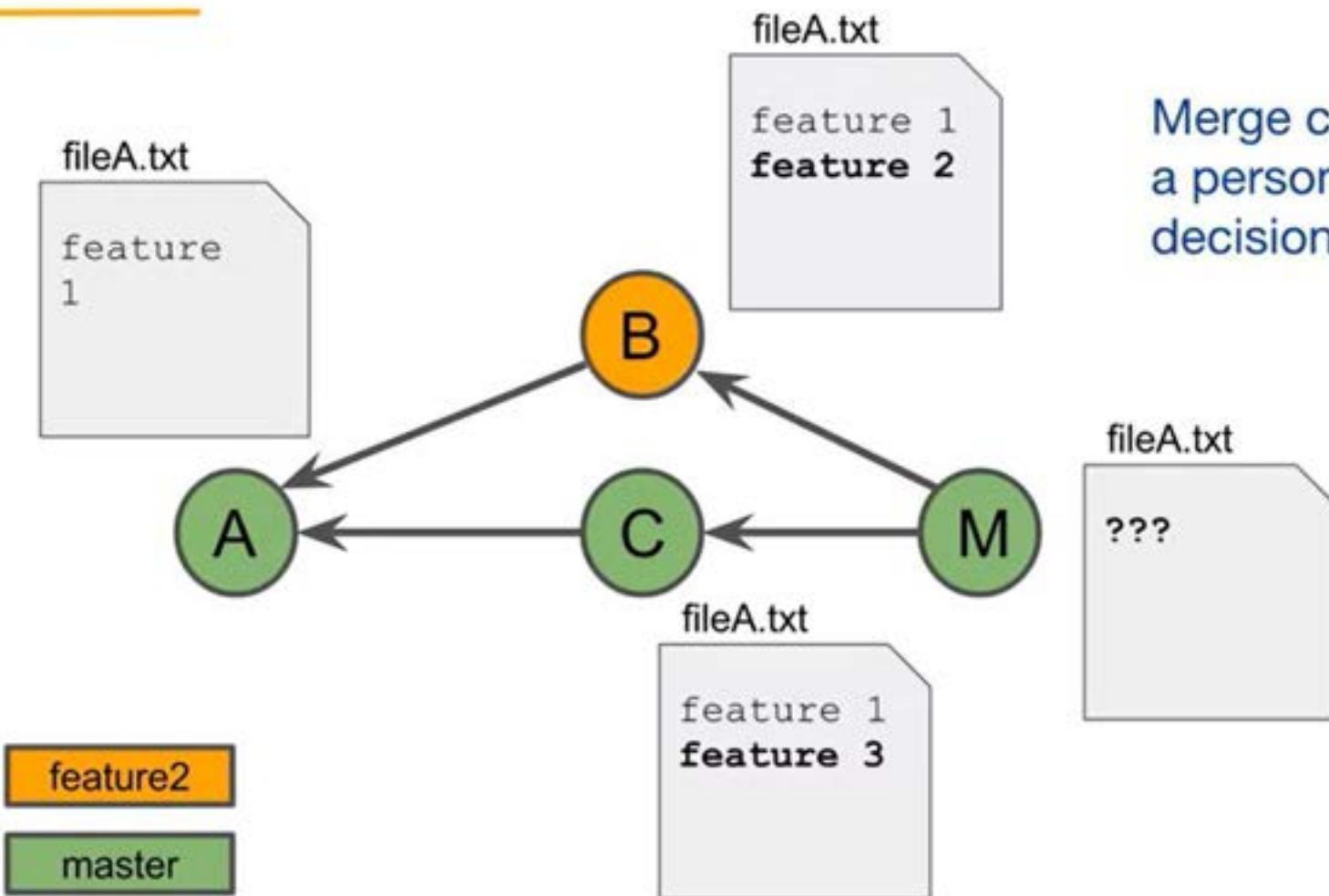
ABORTING A MERGE ATTEMPT (2 OF 3)



REVIEW

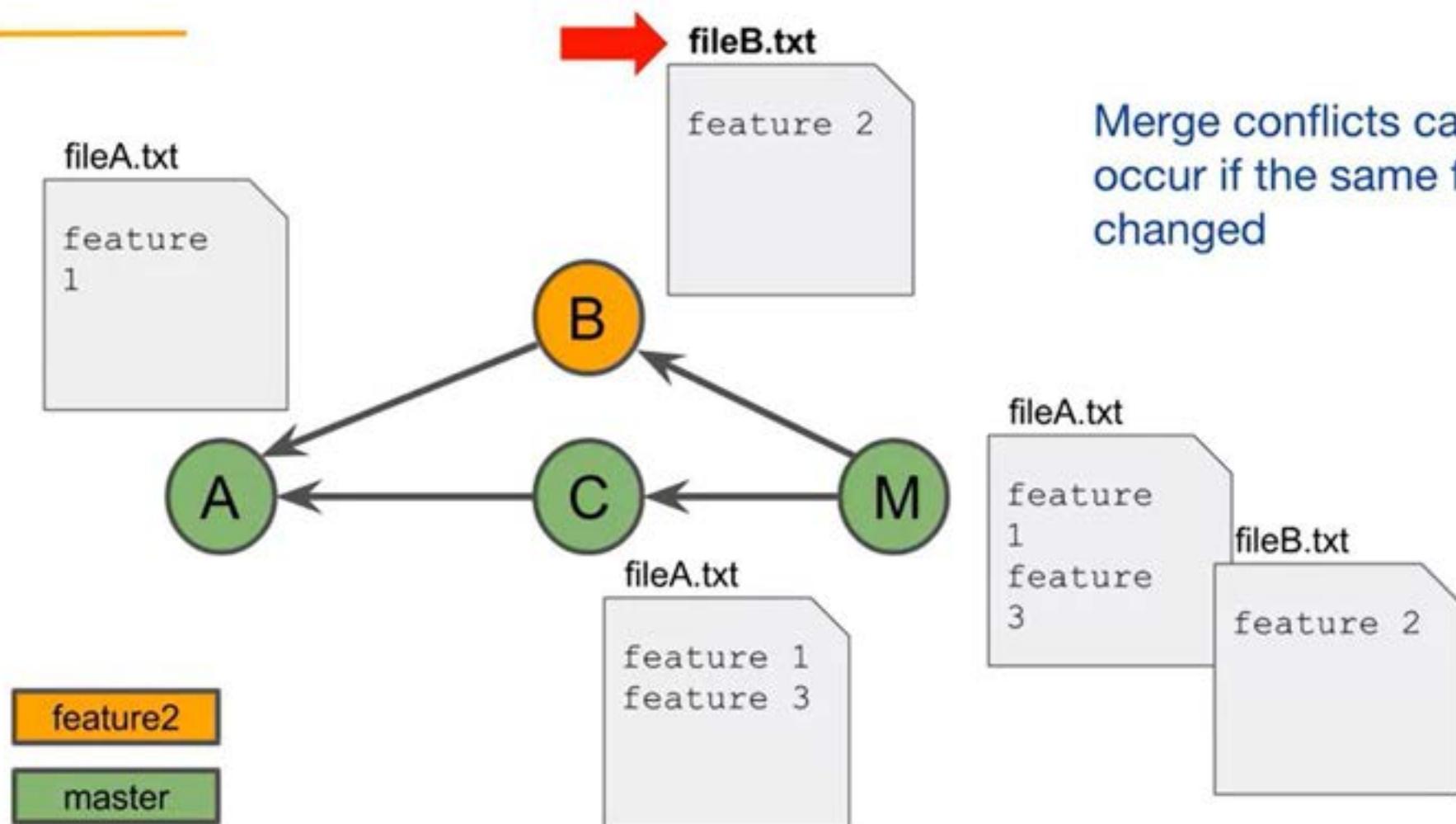
- Merge conflicts occur when two branches modify the same hunk
- When a conflict occurs:
 - Git will create files in the working tree containing conflict markers
 - Fix, add and commit the conflicted files

MERGE CONFLICTS



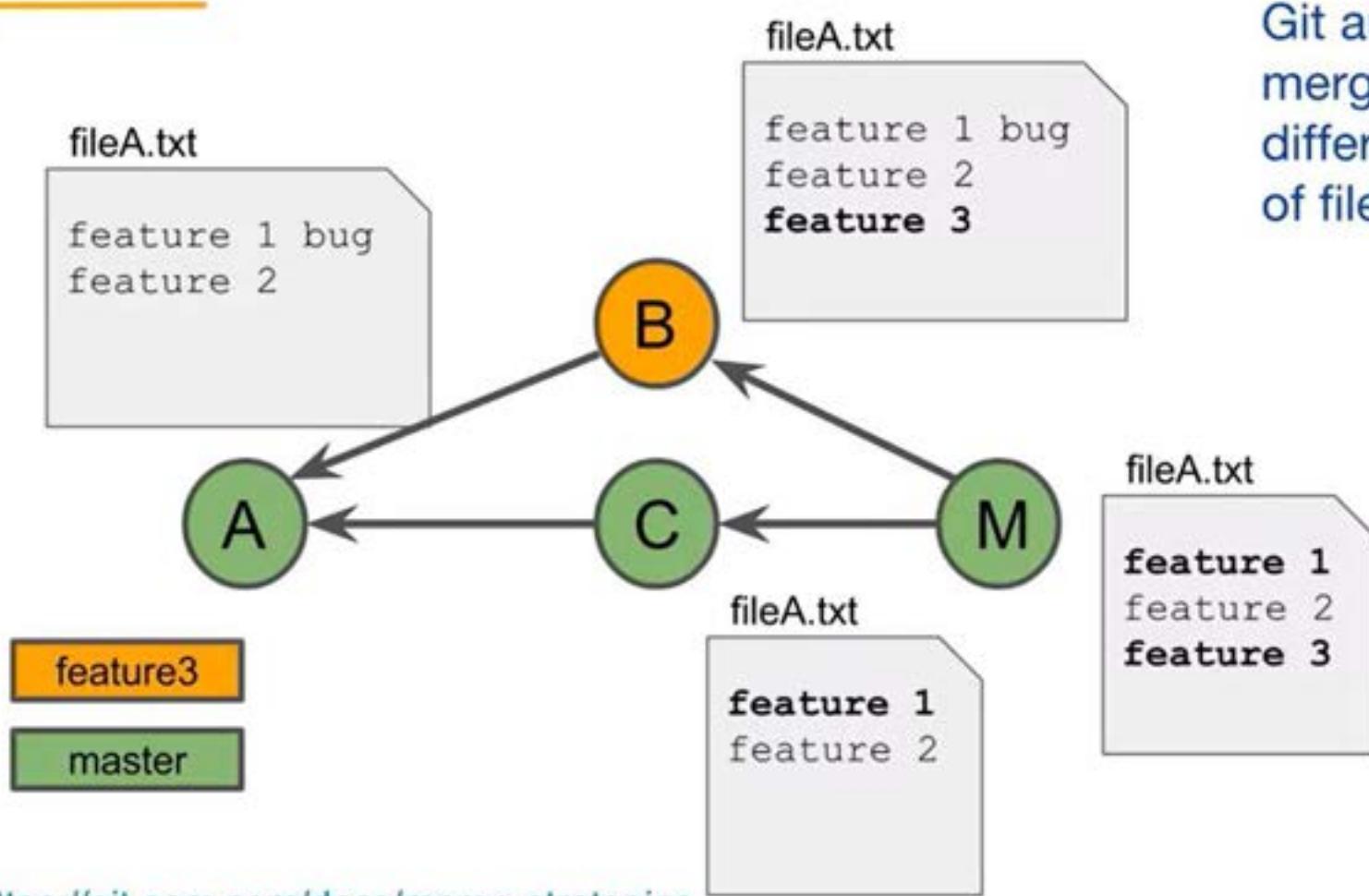
Merge conflicts occur when a person needs to make a decision

NOT A MERGE CONFLICT- DIFFERENT FILES



Merge conflicts can only occur if the same file is changed

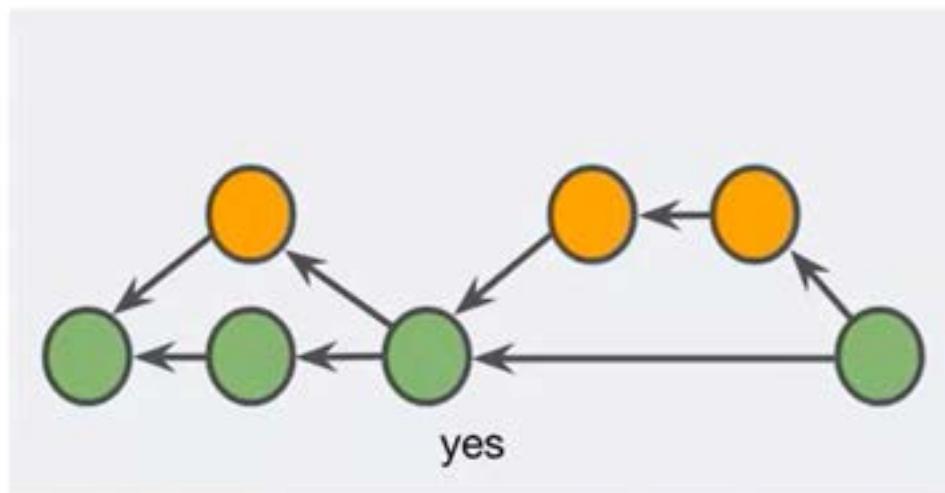
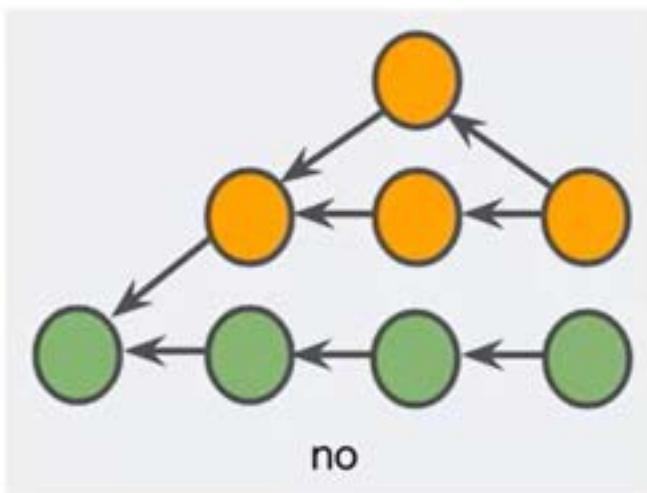
NOT A MERGE CONFLICT- DIFFERENT HUNKS



Git automatically merges changes to different parts (hunks) of files

AVOIDING MERGE CONFLICTS

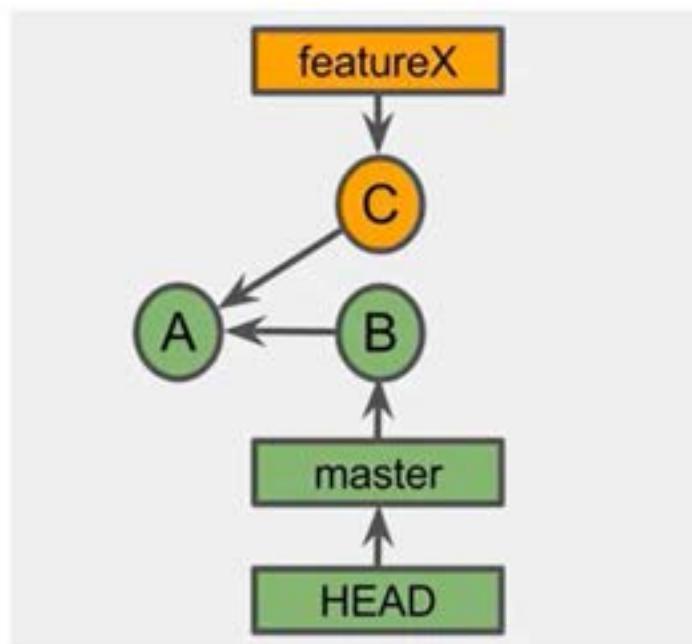
- Git merges are usually quite easy
- Small, frequent merges are the easiest



RESOLVING A MERGE CONFLICT

Involves three commits:

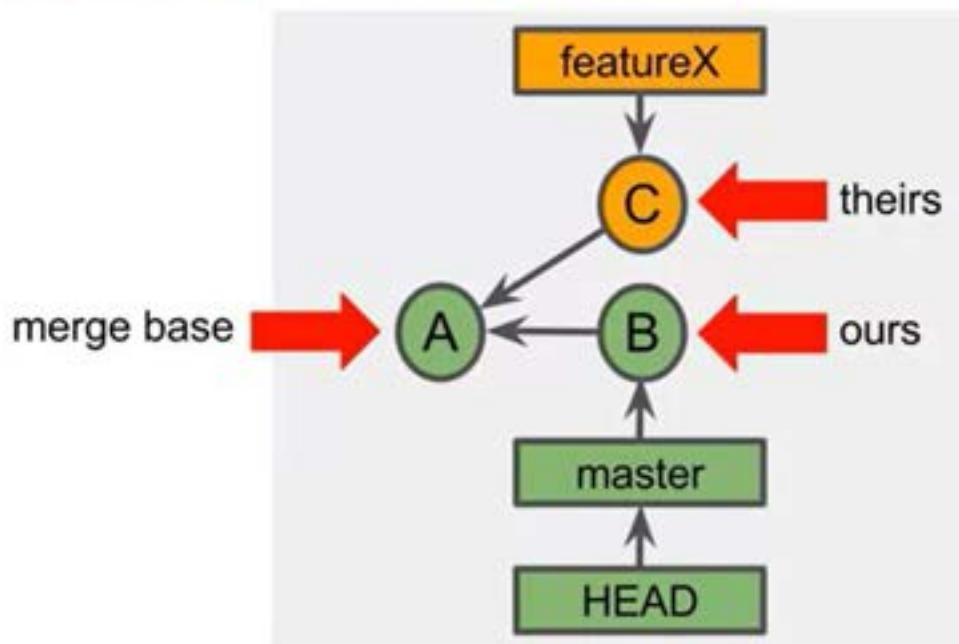
- 1.
- 2.
- 3.



RESOLVING A MERGE CONFLICT

Involves three commits:

1. The tip of the current branch (B)- "ours" or "mine"
2. The tip of the branch to be merged (C)- "theirs"
3. A common ancestor (A)- "merge base"



BASIC STEPS TO RESOLVE A MERGE CONFLICT

1. Checkout master
2. Merge featureX
 - a. CONFLICT- Both modified fileA.txt
3. Fix fileA.txt
4. Stage fileA.txt
5. Commit the merge commit
6. Delete the featureX branch label

BASIC STEPS TO RESOLVE A MERGE CONFLICT

1. Checkout master
2. Merge featureX
 - a. CONFLICT- Both modified fileA.txt
3. Fix fileA.txt
4. Stage fileA.txt
5. Commit the merge commit
6. Delete the featureX branch label

When attempting a merge, files with conflicts are **modified by Git** and placed in the working tree

MERGE CONFLICT

```
$ git log --oneline --graph --all
* c1633f9 (HEAD -> master) added feature 3
| * 942b91e (feature2) added feature 2
|/
* c431e4b added feature 1
$ git merge feature2
Auto-merging fileA.txt
CONFLICT (content): Merge conflict in fileA.txt
Automatic merge failed; fix conflicts and then commit the result.
$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

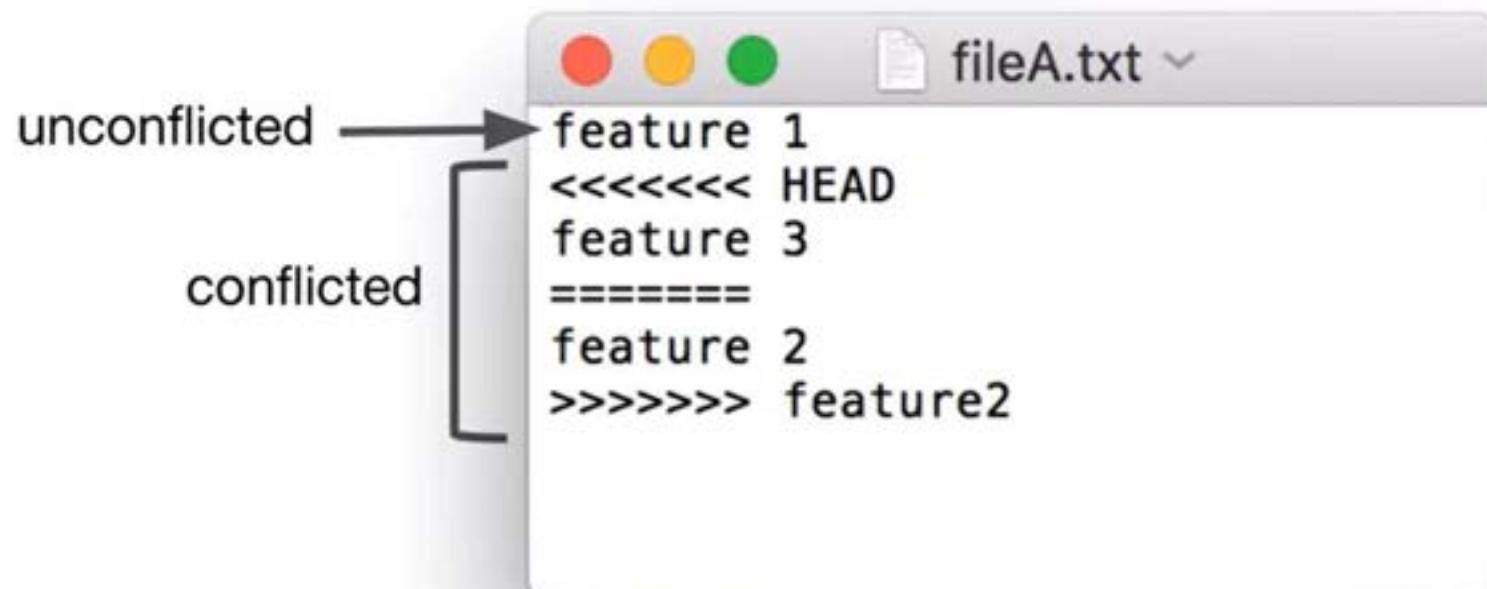
Unmerged paths:
  (use "git add <file>..." to mark resolution)

    both modified:   fileA.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

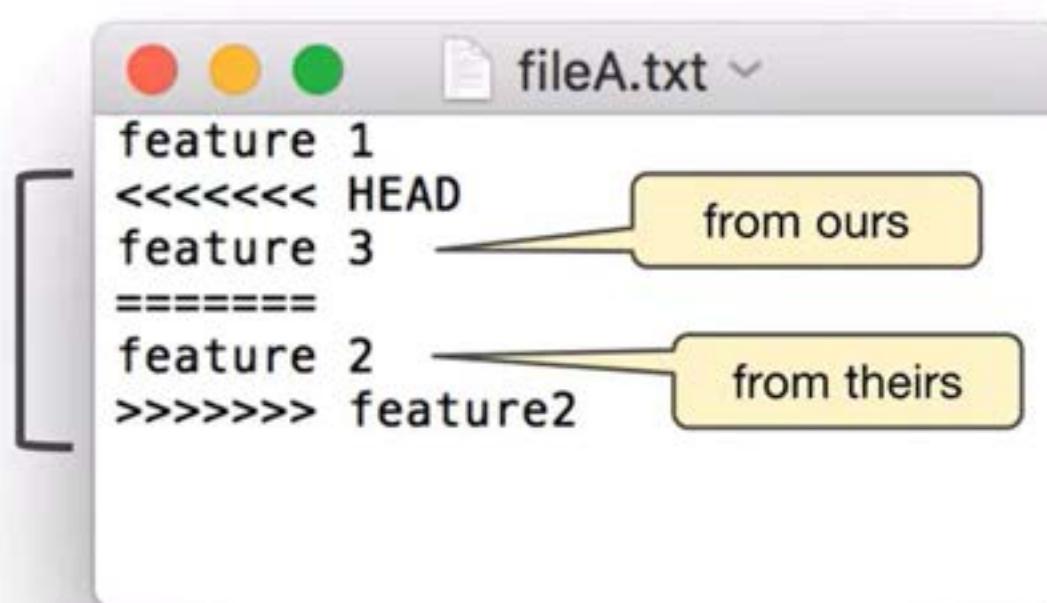
CONFLICTED HUNKS

Conflicted hunks are surrounded by conflict markers <<<<< and
>>>>>



READING CONFLICT MARKERS

- Text from the HEAD commit is between <<<<< and =====
- Text from the branch to be merged is between ===== and >>>>>



FIXING A CONFLICTED FILE

1. Checkout master
2. Merge featureX
3. **Fix fileA.txt**
4. Add fileA.txt
5. Commit the merge
6. Delete featureX

```
fileA.txt ~
feature 1
<<<<< HEAD
feature 3
=====
feature 2
>>>>> feature2
```

before fixing a file

```
fileA.txt — Edited ~
feature 1
feature 2
feature 3
```

after fixing a file

FIX AND COMMIT THE MERGE

```
$ cat fileA.txt
feature 1
feature 2
feature 3
$ git add fileA.txt
$ git commit
(edited merge message if desired)
[master a8899d8] Merge branch 'feature2'
$ git log --oneline --graph --all
*   a8899d8 (HEAD -> master) Merge branch 'feature2'
|\ \
| * 942b91e (feature2) added feature 2
* | c1633f9 added feature 3
| /
* c431e4b added feature 1
```



FIX AND COMMIT THE MERGE

```
$ cat fileA.txt
feature 1
feature 2
feature 3
$ git add fileA.txt
$ git commit
(edit merge message if desired)
[master a8899d8] Merge branch 'feature2'
$ git log --oneline --graph --all
*   a8899d8 (HEAD -> master) Merge branch 'feature2'
|\ \
| * 942b91e (feature2) added feature 2
* | c1633f9 added feature 3
| /
* c431e4b added feature 1
$ git branch -d feature2
Deleted branch feature2 (was 942b91e).
```

REVIEW

- Merge conflicts occur when two branches modify the same hunk
- When a conflict occurs:
 - Git will create files in the working tree containing conflict markers
 - Fix, add and commit the conflicted files

Topics

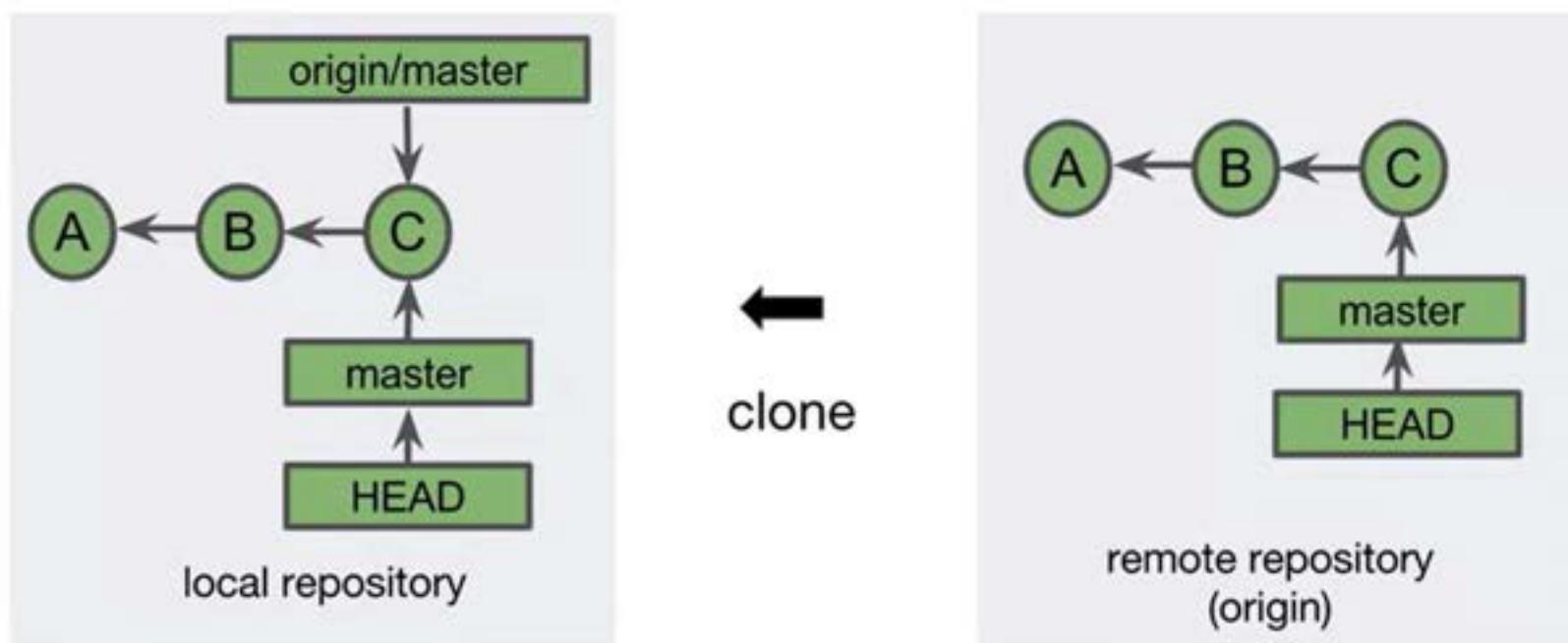
Tracking branch overview

Viewing tracking branch names

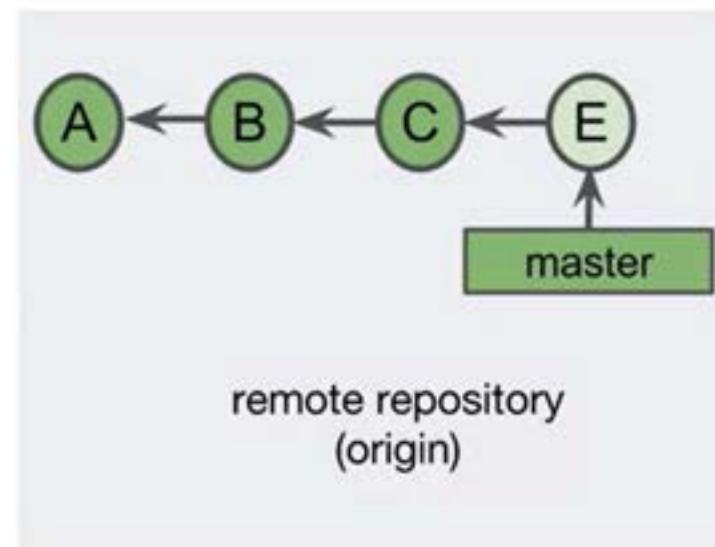
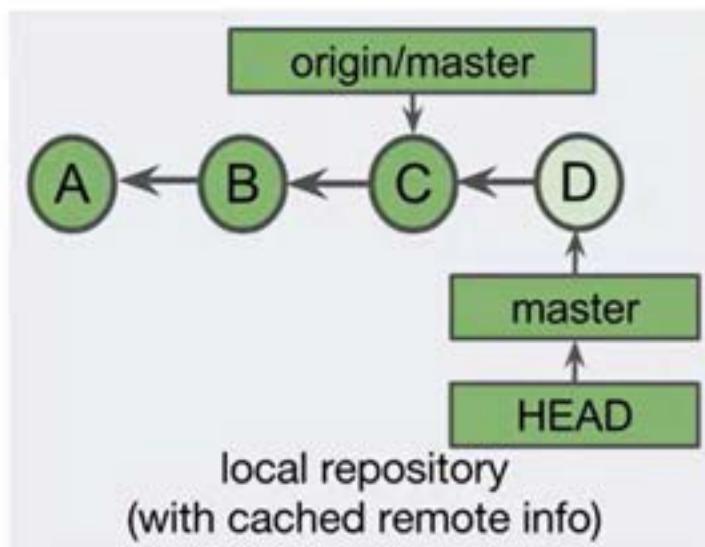
Viewing tracking branch status

TRACKING BRANCH

A local branch that represents a remote branch
`<remote>/<branch>`



TRACKING BRANCHES- RELATED BUT DECOUPLED



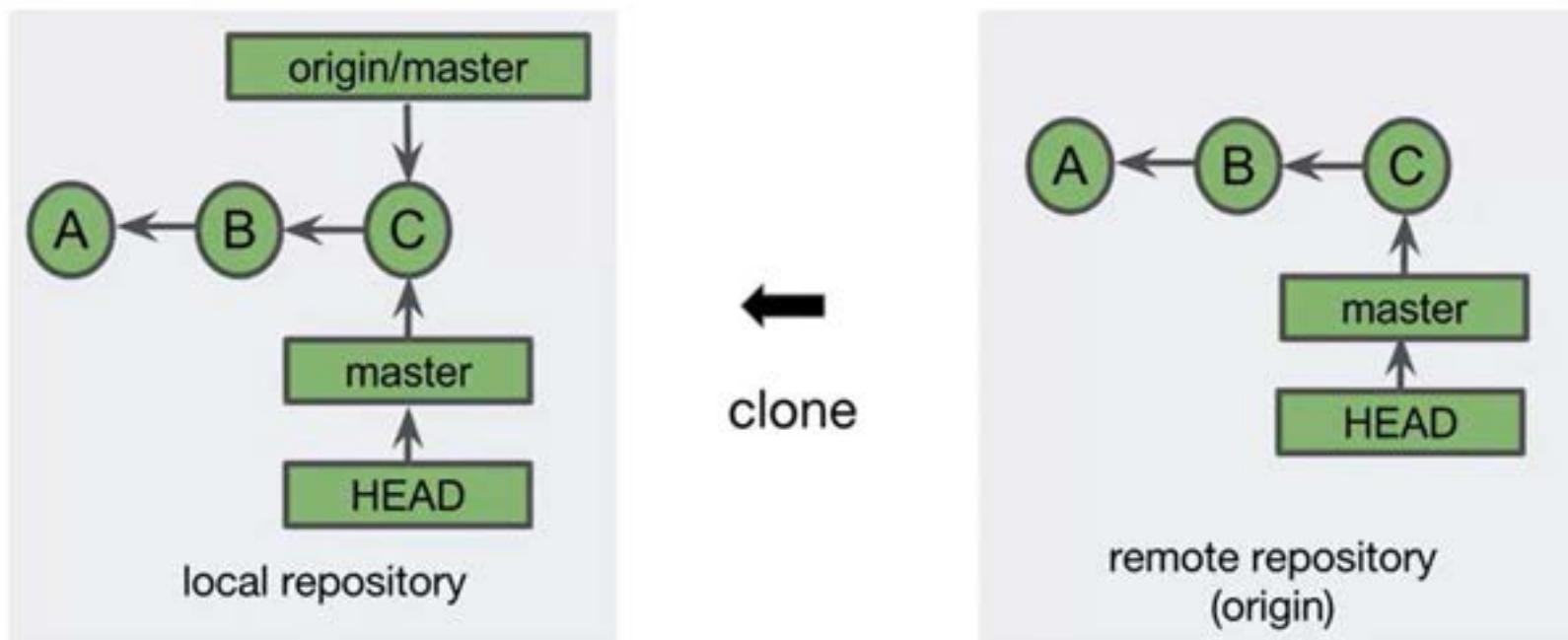
Topics

Tracking branch overview

Viewing tracking branch names

Viewing tracking branch status

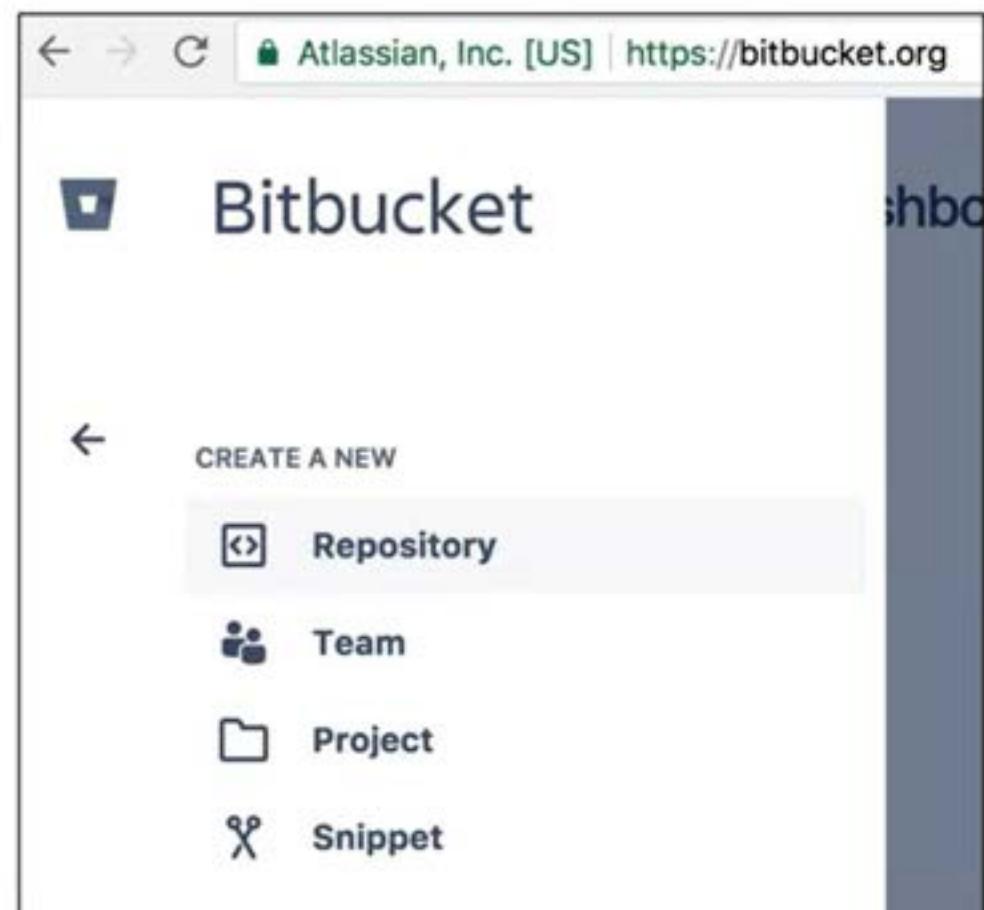
CLONING CREATES A TRACKING BRANCH



CREATE A REMOTE REPOSITORY



The screenshot shows the Bitbucket homepage. The URL in the address bar is https://bitbucket.org. The main navigation menu on the left includes: Bitbucket (selected), Overview (highlighted in blue), Repositories, Projects, Pull requests, Issues, and Snippets. To the right of the menu, there is a "Dashboard" section.



The screenshot shows the "CREATE A NEW" page in Bitbucket. The URL in the address bar is https://bitbucket.org. The page lists four options: Repository (selected and highlighted in blue), Team, Project, and Snippet. There is a back arrow at the top left.

CREATE A REMOTE REPOSITORY

Create a new repository [Import repository](#)

Repository name *

Access level

Version control system Git Mercurial

[Advanced settings](#)

[Create repository](#) [Cancel](#)

CREATE A COMMIT IN THE REPOSITORY

Overview



Put some bits in your bucket

Add some code or content and start bringing your ideas to life. [Learn how](#)

Get started the easy way

Creating a `README` or a `.gitignore` is a quick and easy way to get something into your repository.

[Create a `README`](#)

[Create a `.gitignore`](#)

CREATE A COMMIT IN THE REPOSITORY



A screenshot of a code editor interface. The title bar says "projecte / README.md". The main area shows the content of the README.md file:
1 # PROJECTE README #
2

Syntax mode: Markdown Indent mode: Tabs Line wrap: Off **Commit**



VIEW THE COMMIT

Commits		
All branches ▾		
Author	Commit	Message
Pat	0b374c5	add README.md

CLONE THE REPOSITORY

The screenshot shows a user interface for managing a Git repository. On the left, there's a sidebar with icons for Home, Overview, Source, Commits (which is selected), Branches, and Pull requests. The main area has a header "Atlassian, Inc. [US]" and a title "projecte". A modal window is open, titled "Clone this repository". It contains a dropdown menu set to "HTTPS" with the URL "git clone https://". Below this is a button "Clone in SourceTree". At the bottom right of the modal is a "Close" button. The main interface also lists "CREATE A NEW" options: Repository, Team, Project, and Snippet. Under "GET TO WORK", it lists: Clone this repository (which is highlighted), Create a branch, Create a pull request, Compare branches or tags, and Fork this repository.

Atlassian, Inc. [US]

projecte

CREATE A NEW

- Repository
- Team
- Project
- Snippet

GET TO WORK

- Clone this repository
- Create a branch
- Create a pull request
- Compare branches or tags
- Fork this repository

Clone this repository

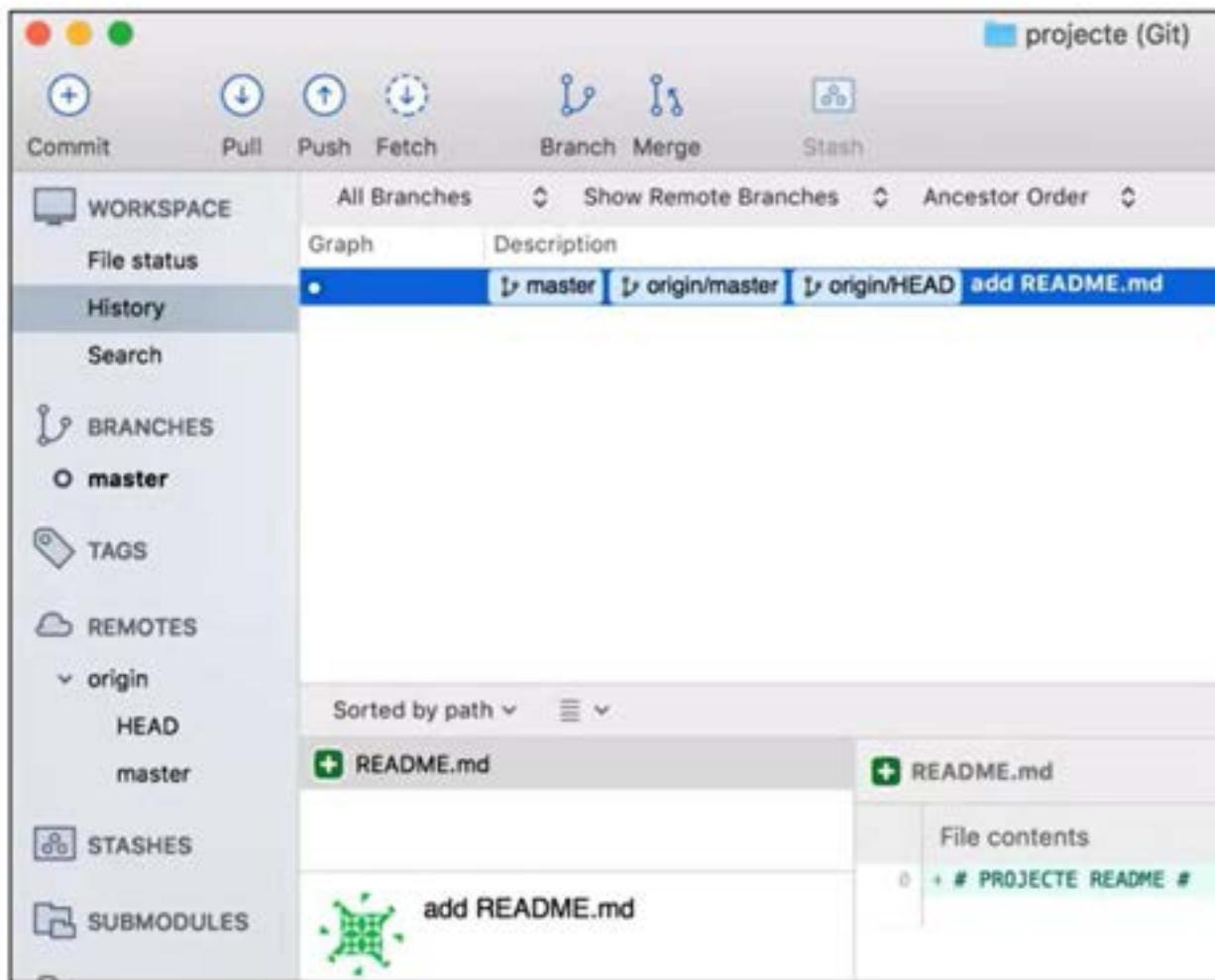
HTTPS

Clone in SourceTree

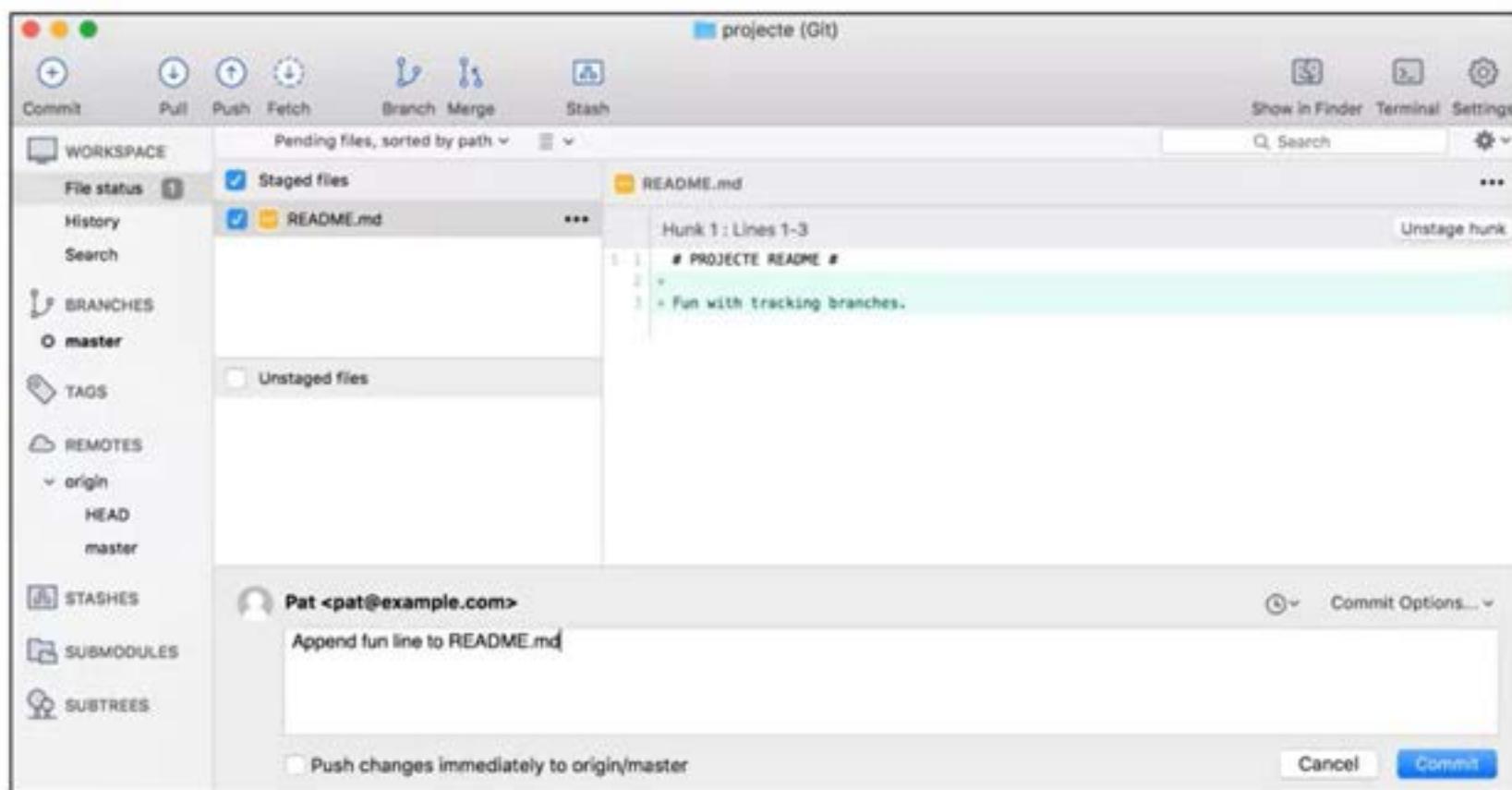
Learn more about cloning

Close

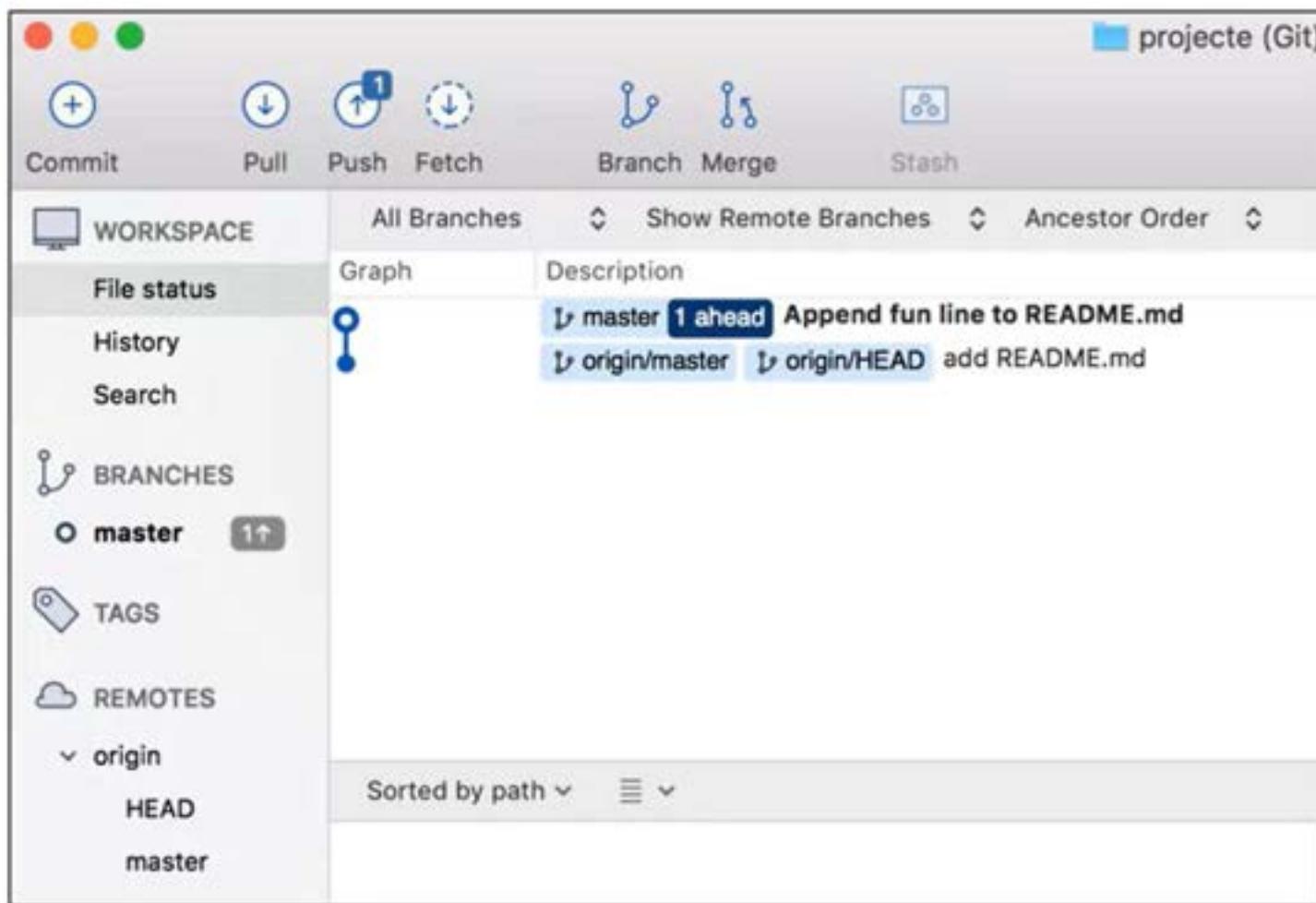
VIEW THE TRACKING BRANCH



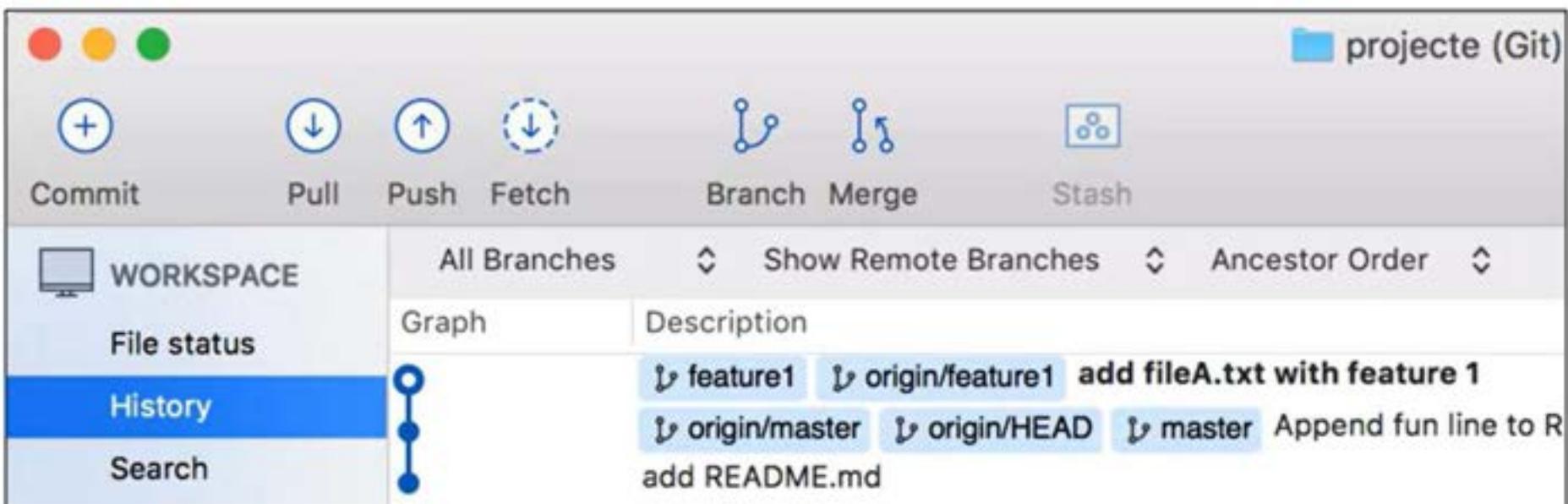
CREATE A LOCAL COMMIT



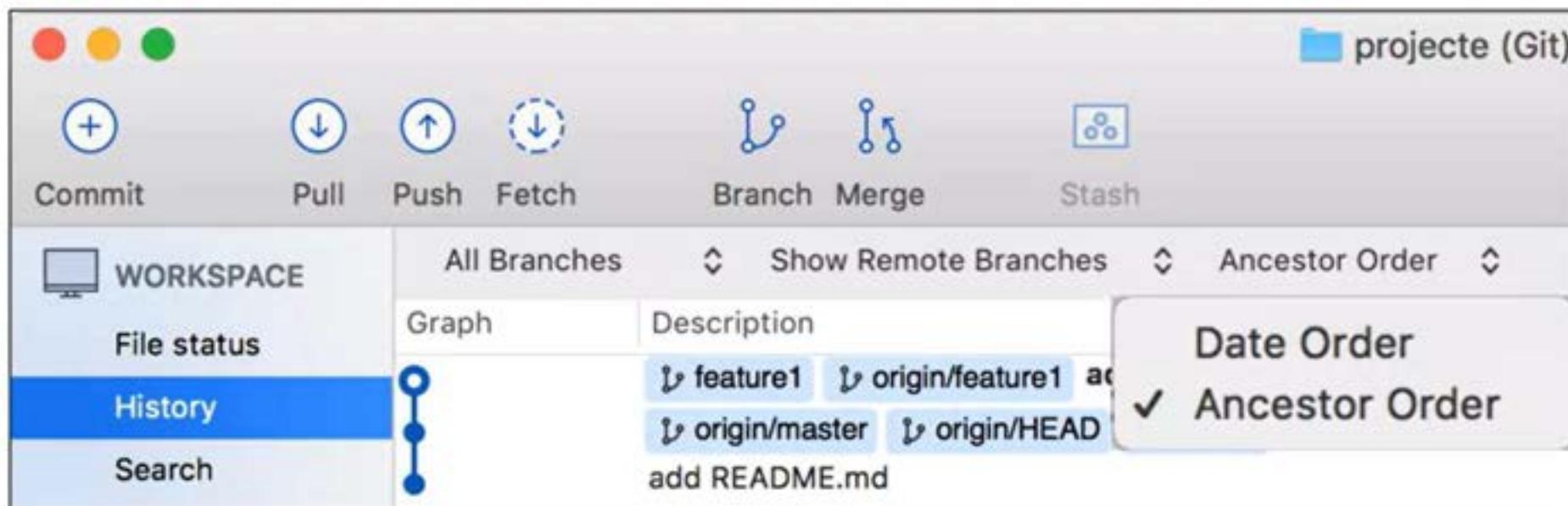
VIEW THE ONE AHEAD STATE



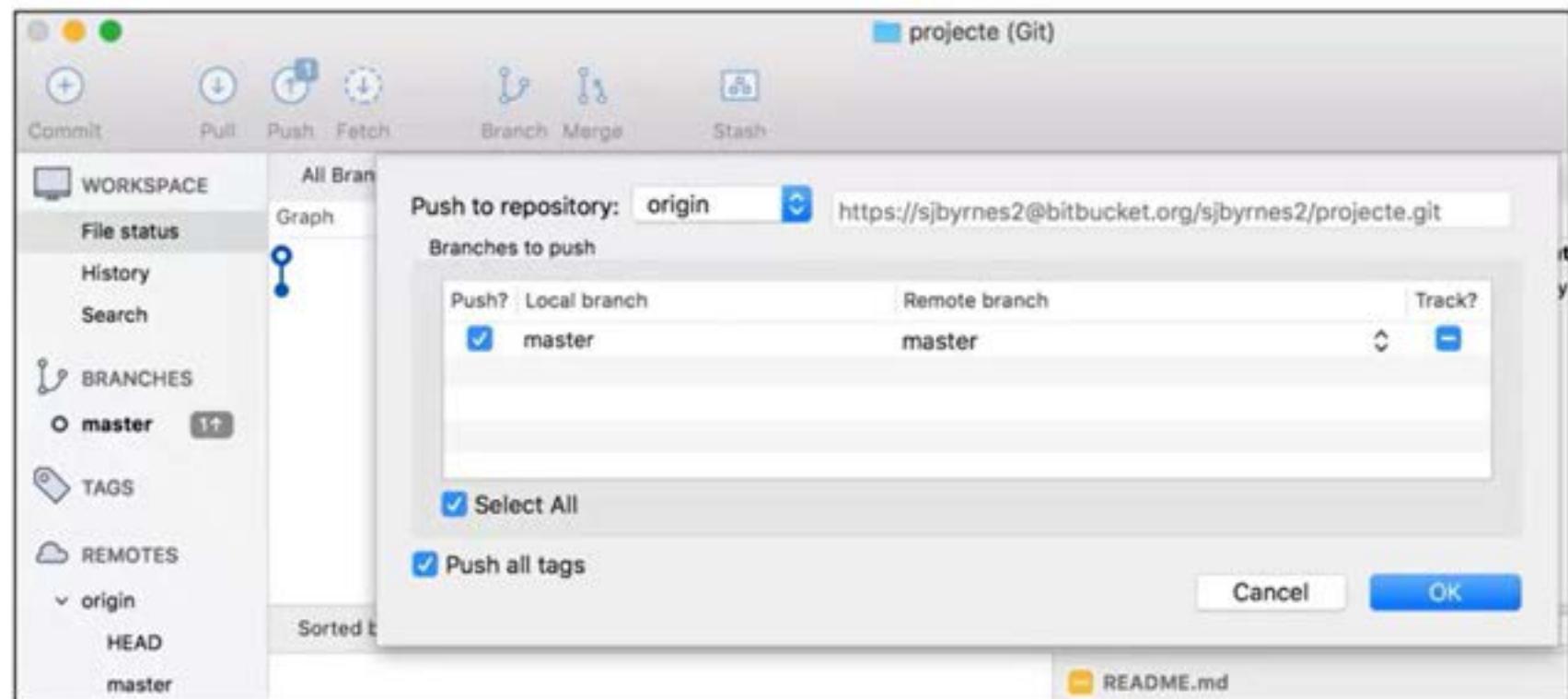
HISTORY COLUMN FILTERS



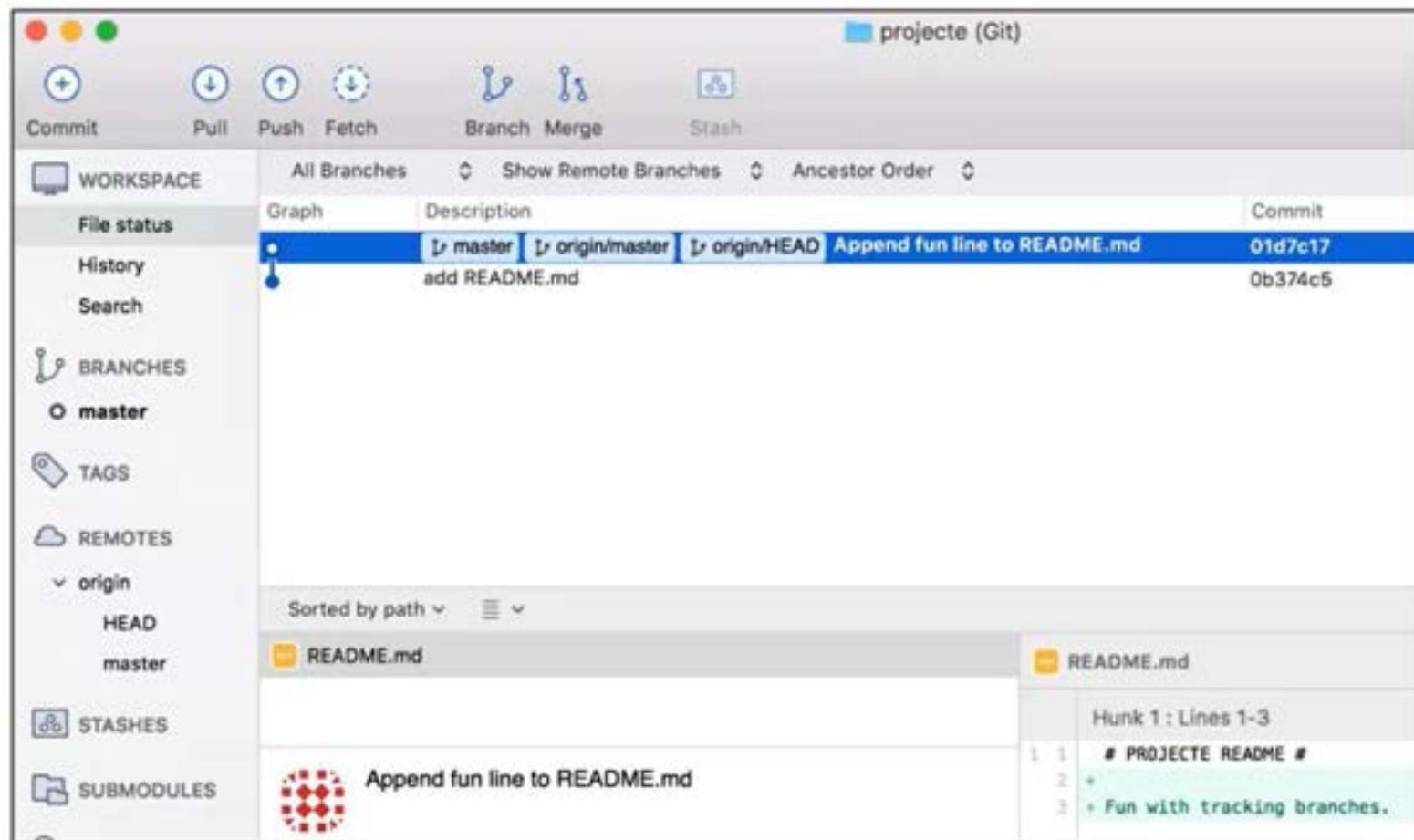
HISTORY COLUMN FILTERS



PUSH



REPOSITORIES SYNCHRONIZED



REVIEW- TRACKING BRANCHES

- Local branches that represent remote branches
- Named **<remote>/<branch>**, for example **origin/master**
- Can become out of synch with local branches
- Updated with network commands like **clone**, **fetch**, **pull** and **push**

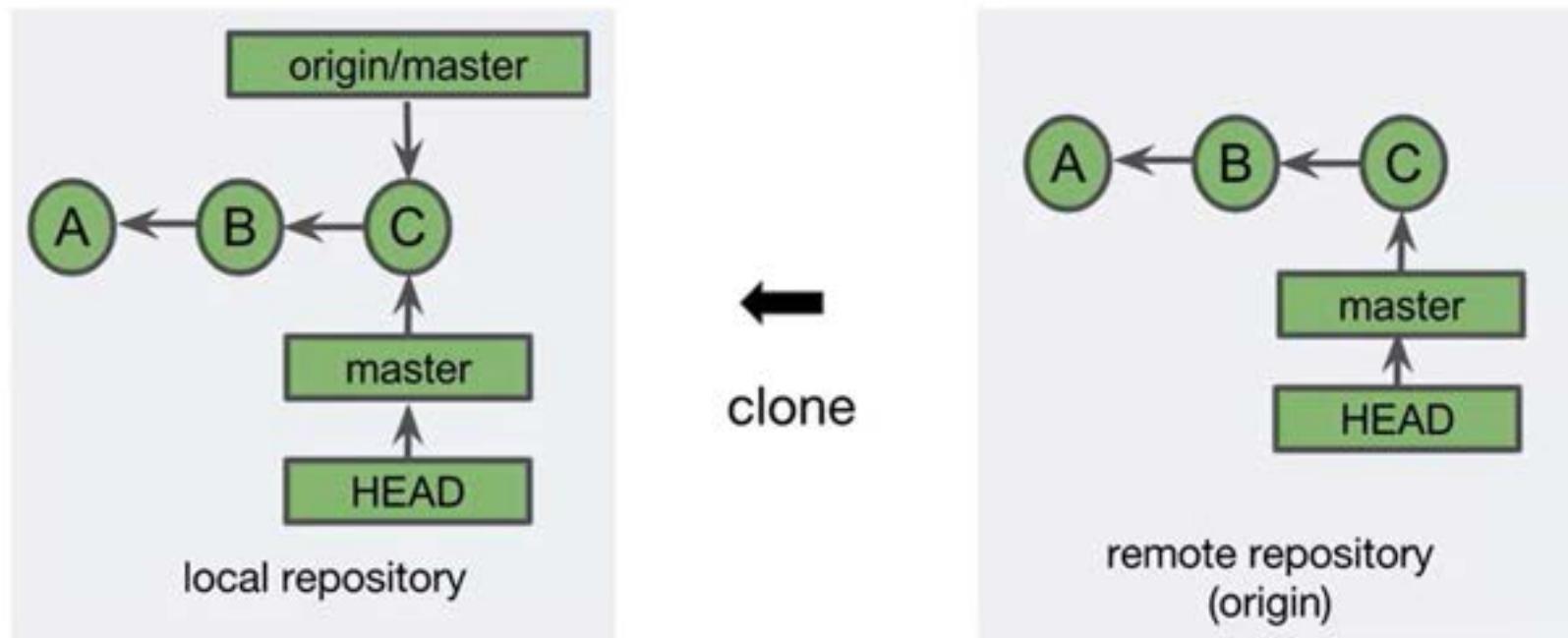
Tracking Branches

using a command line

Copyright © 2018 Atlassian

TRACKING BRANCH

A local branch that represents a remote branch
`<remote>/<branch>`



Topics

Tracking branch overview

Viewing tracking branch names

Viewing tracking branch status

VIEWING TRACKING BRANCH NAMES

git branch --all

Displays local and tracking branch names

```
repos$ git clone https://bitbucket.org/me/projecte.git
Cloning into 'projecte'...
$ cd projecte
projecte$ git branch
* master
projecte$ git branch --all
* master
  remotes/origin/HEAD -> origin/master
  remotes/origin/master
```

remotes/origin/HEAD

Specifies the default remote tracking branch

- Allows <remote> to be specified instead of <remote>/<branch> in Git commands

```
$ git branch --all
* master
  remotes/origin/HEAD -> origin/master
  remotes/origin/master
$ git log origin/master --oneline
215b50a (origin/master, origin/HEAD) add feature 1
f92ad48 (HEAD -> master) add fileA.txt
$ git log origin --oneline
215b50a (origin/master, origin/HEAD) add feature
f92ad48 (HEAD -> master) add fileA.txt
```

CHANGING remotes/origin/HEAD

Change the default remote tracking branch with

git remote set-head <remote> <branch>

```
$ git branch --all
* develop
  master
    remotes/origin/HEAD -> origin/master
    remotes/origin/develop
    remotes/origin/master
$ git remote set-head origin develop
$ git branch --all
* develop
  master
    remotes/origin/HEAD -> origin/develop
    remotes/origin/develop
    remotes/origin/master
```

CHANGING THE REMOTE REPOSITORY'S HEAD

This sets the default branch for all users

Screenshot of a GitHub repository settings page for 'projectb'.

The left sidebar shows:

- Overview
- Source
- Commits
- Pull requests
- Pipelines
- Downloads
- Boards
- Settings** (selected)

The main content area shows 'Repository details' with the following fields:

- Name: projectb
- Size: 124.0 KB
- Git Large File Storage: 0 bytes of space used, (1.0 GB remaining)
- Description: (empty)
- Access level: This is a private repository
- Forking: Allow only private forks
- Landing page: Overview
- Website: (empty)
- Language: Select language...
- Main branch: master
- Google Analytics key: (empty)

A blue 'Save repository details' button is at the bottom.

To the right, a diagram titled 'remote repository' illustrates a branching structure:

```
graph TD; A((A)) --> B((B)); B --> C((C)); C --> master[master]; master --> HEAD[HEAD]
```

The diagram shows three green circles labeled A, B, and C connected by arrows pointing left. Below them is a green rectangle labeled 'master'. An arrow points from 'master' to a green rectangle labeled 'HEAD'.

Below the diagram, a callout box contains the text:

Main branch **master** ▾

VIEWING TRACKING BRANCH STATUS

git status includes tracking branch status

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

nothing to commit, working tree clean
```

STATUS WITH NEW LOCAL COMMITS

git status will inform you if the cached tracking branch information is out of sync with your local branch

```
$ git commit -m "added feature 2"
[master 63f4add] added feature 2
 1 file changed, 1 insertion(+)
$ git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
```

VIEWING COMMITS OF ALL LOCAL AND TRACKING BRANCHES

Use `git log --all` to see a combined log of all local and tracking branches

```
(edit fileA.txt)
$ git add fileA.txt
$ git commit -m "added feature 2"
[master 3ed820b] added feature 2
 1 file changed, 1 insertion(+)
$ git log --all --oneline --graph
* 3ed820b (HEAD -> master) added feature 2
* 6da3214 (origin/master) added feature 1
```

HANDS ON

- View tracking branches
- Create a state with the local branch one ahead of the tracking branch

REVIEW- TRACKING BRANCHES

- Local branches that represent remote branches
- Named **<remote>/<branch>**, for example **origin/master**
- Can become out of synch with local branches
- Updated with network commands like **clone**, **fetch**, **pull** and **push**

Topics

Network command overview

Fetch

Pull

Push

NETWORK COMMANDS

Clone- Copies a remote repository

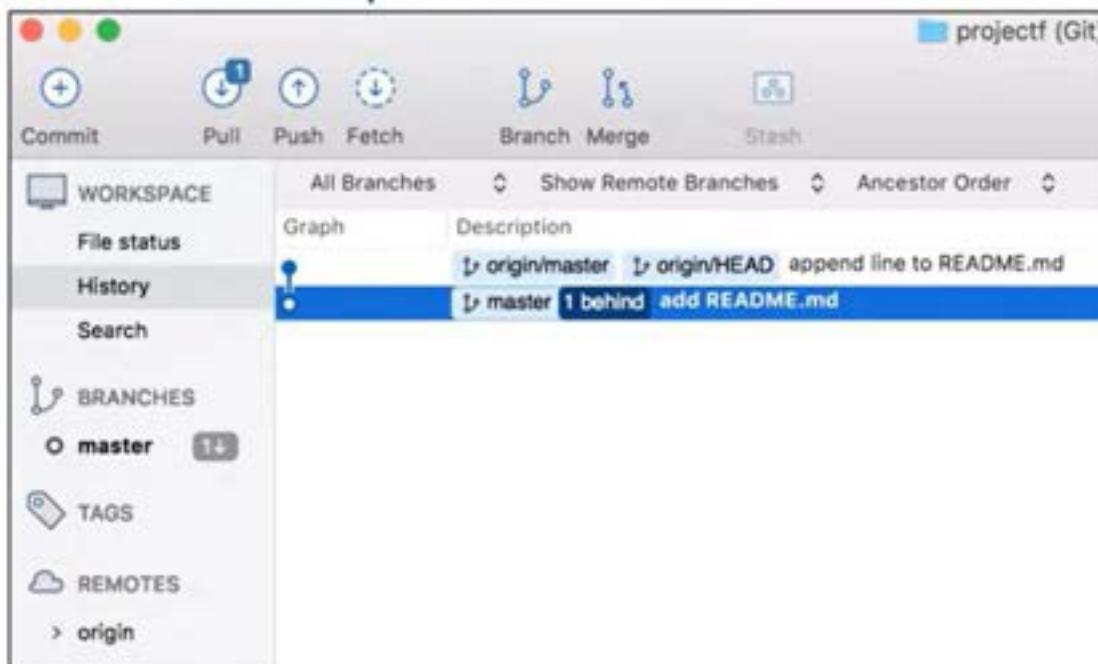
Fetch- Retrieves new objects and references from the remote repository

Pull- Fetches and merges commits locally

Push- Adds new objects and references to the remote repository

FETCH

- Retrieves new objects and references from another repository
- Tracking branches are updated



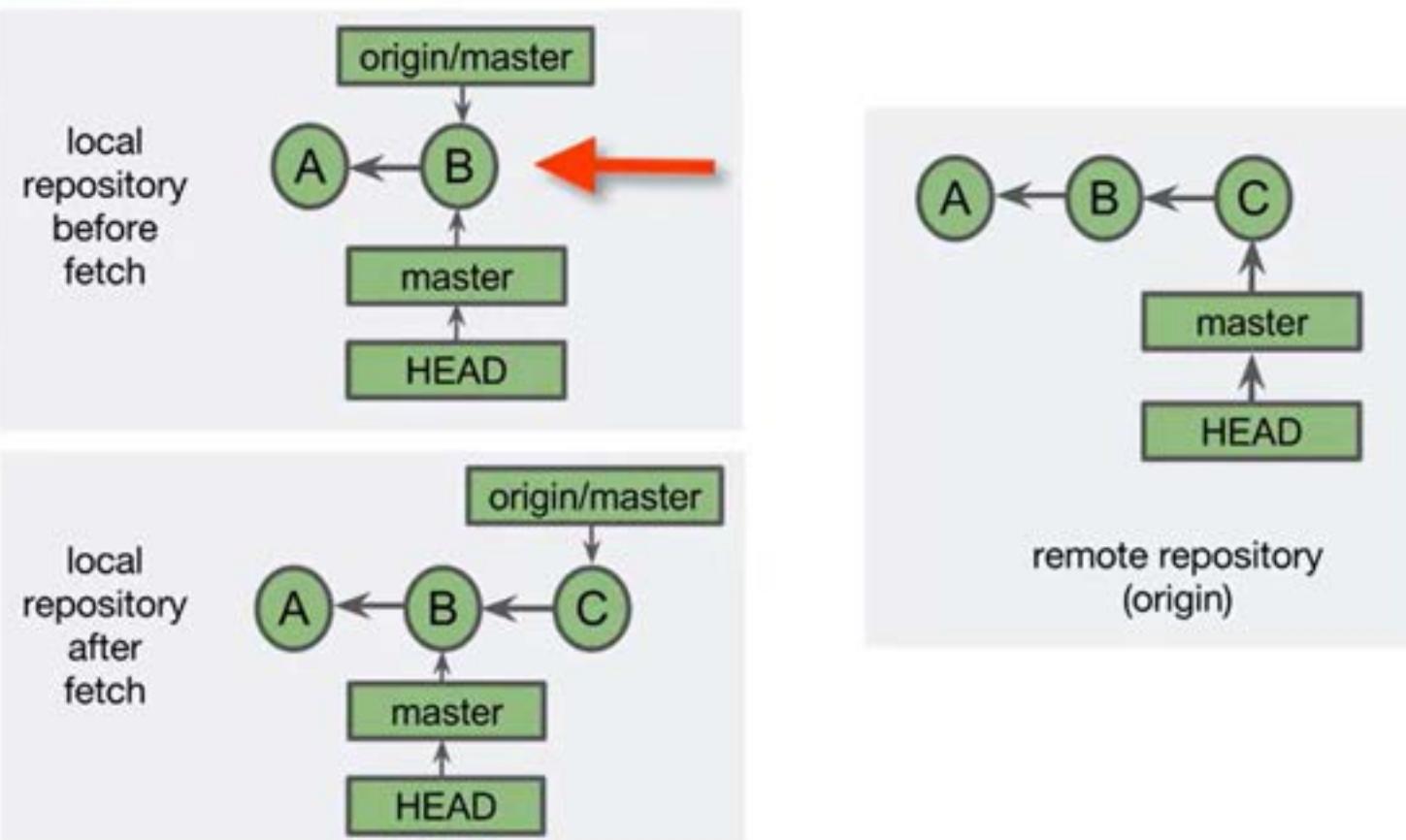
<https://git-scm.com/docs/git-fetch>

6

SOURCETREE AUTOMATICALLY FETCHES



FETCHING THE LATEST COMMITS



Topics

Network command overview

Fetch

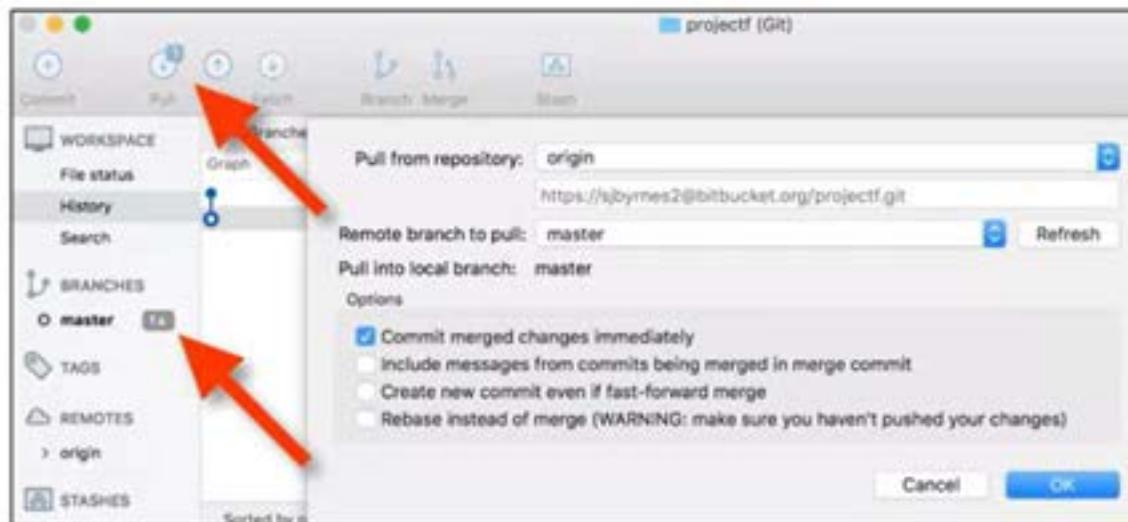
Pull

Push

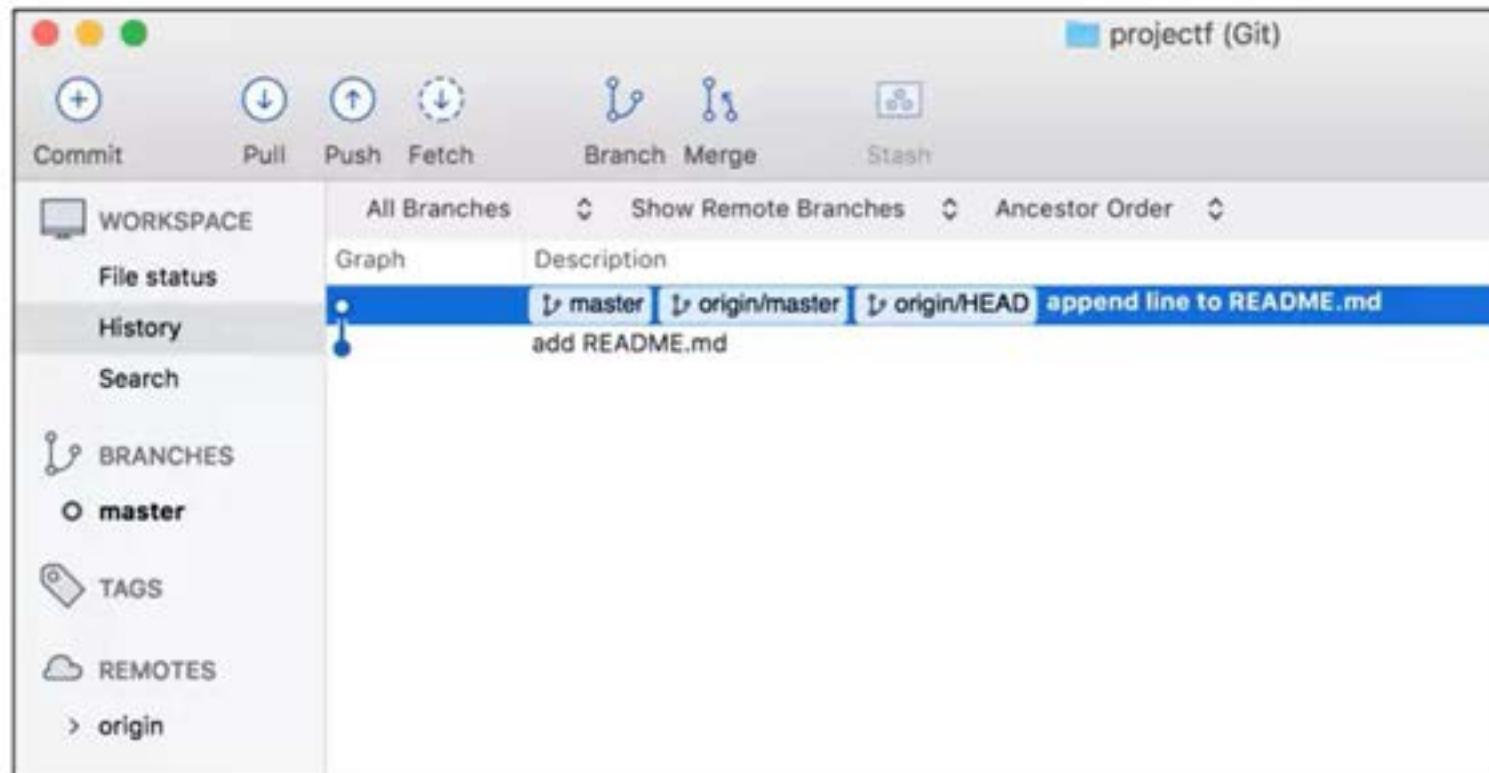
PULL

Combines fetch and merge

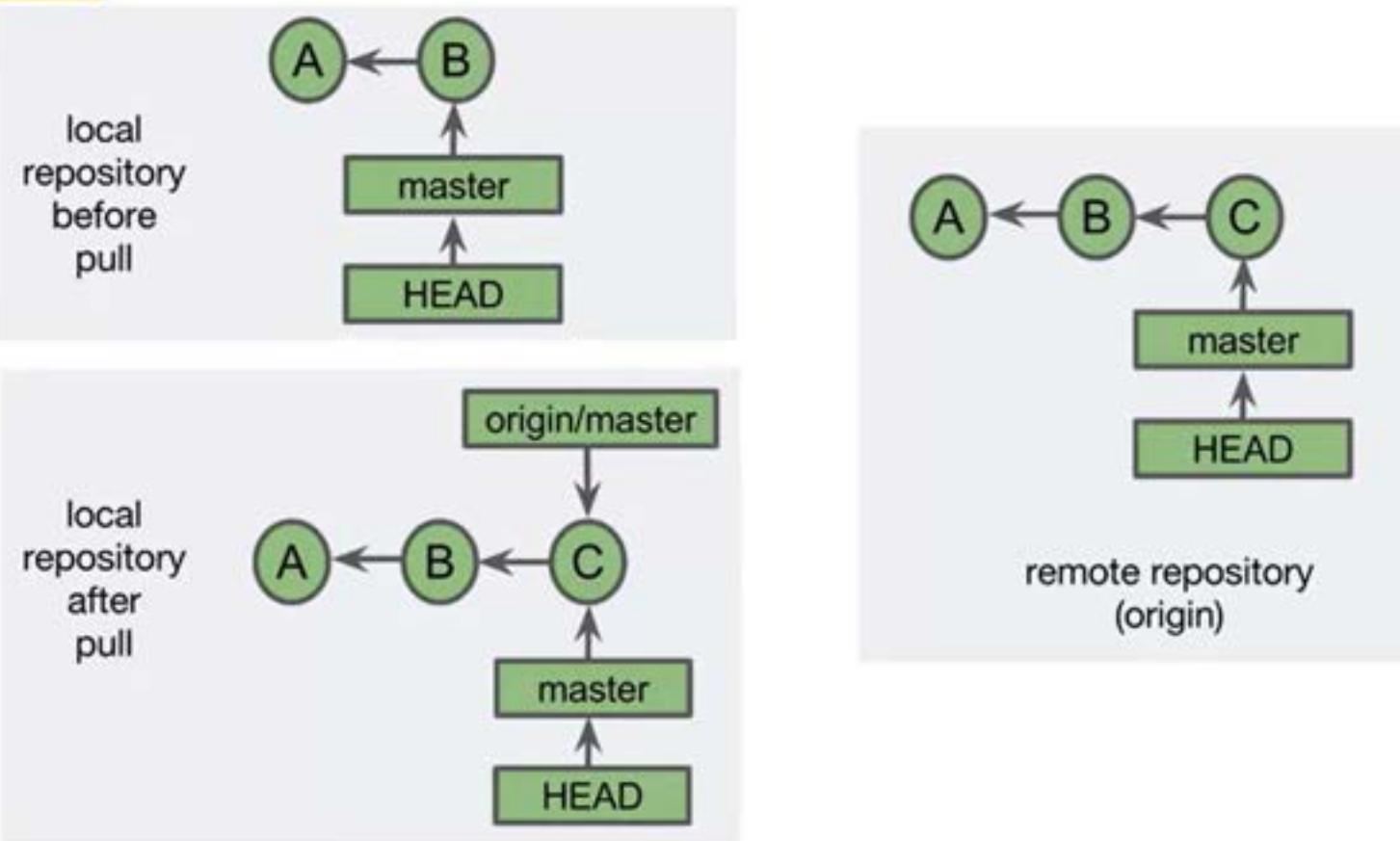
- If objects are fetched, the tracking branch is merged into the current local branch
- This is similar to a topic branch merging into a base branch



AFTER A PULL

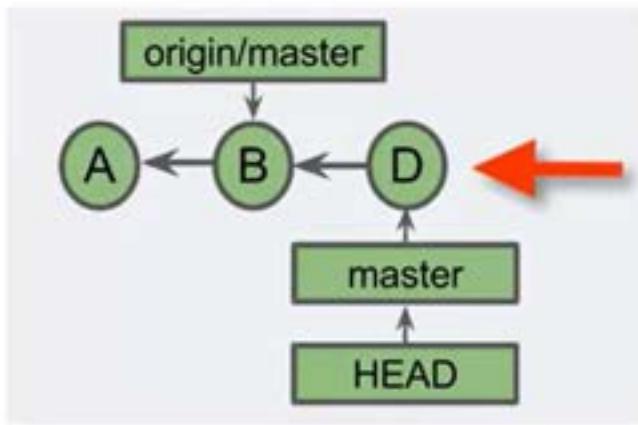


PULL WITH A FAST-FORWARD MERGE

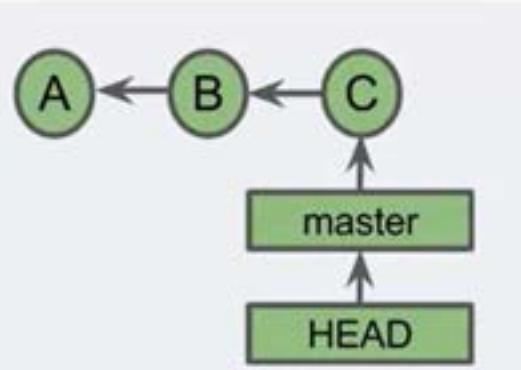
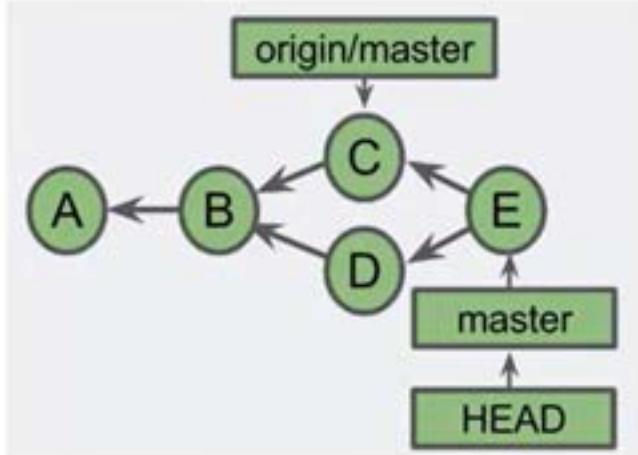


PULL WITH A MERGE COMMIT

local
repository
before
pull

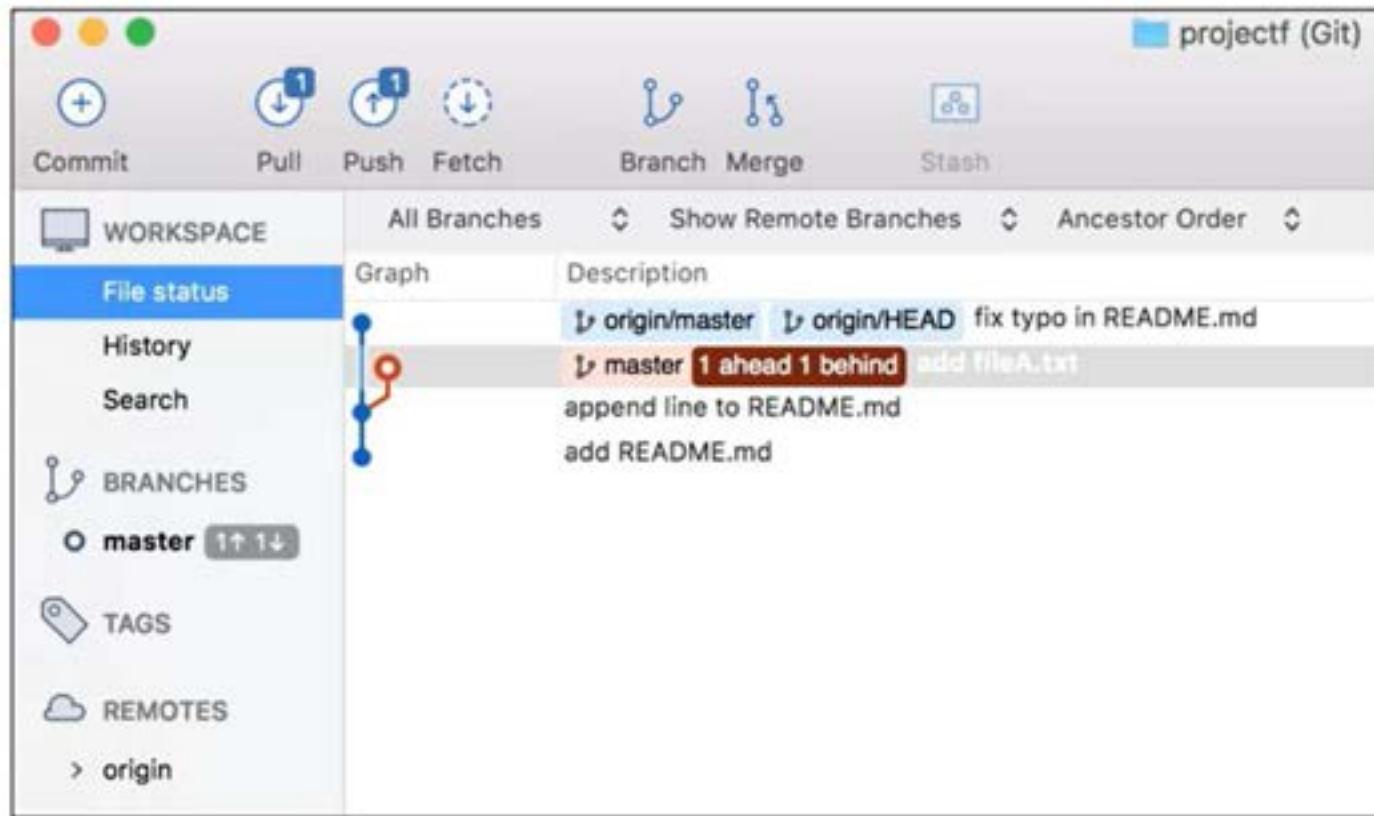


local
repository
after
pull

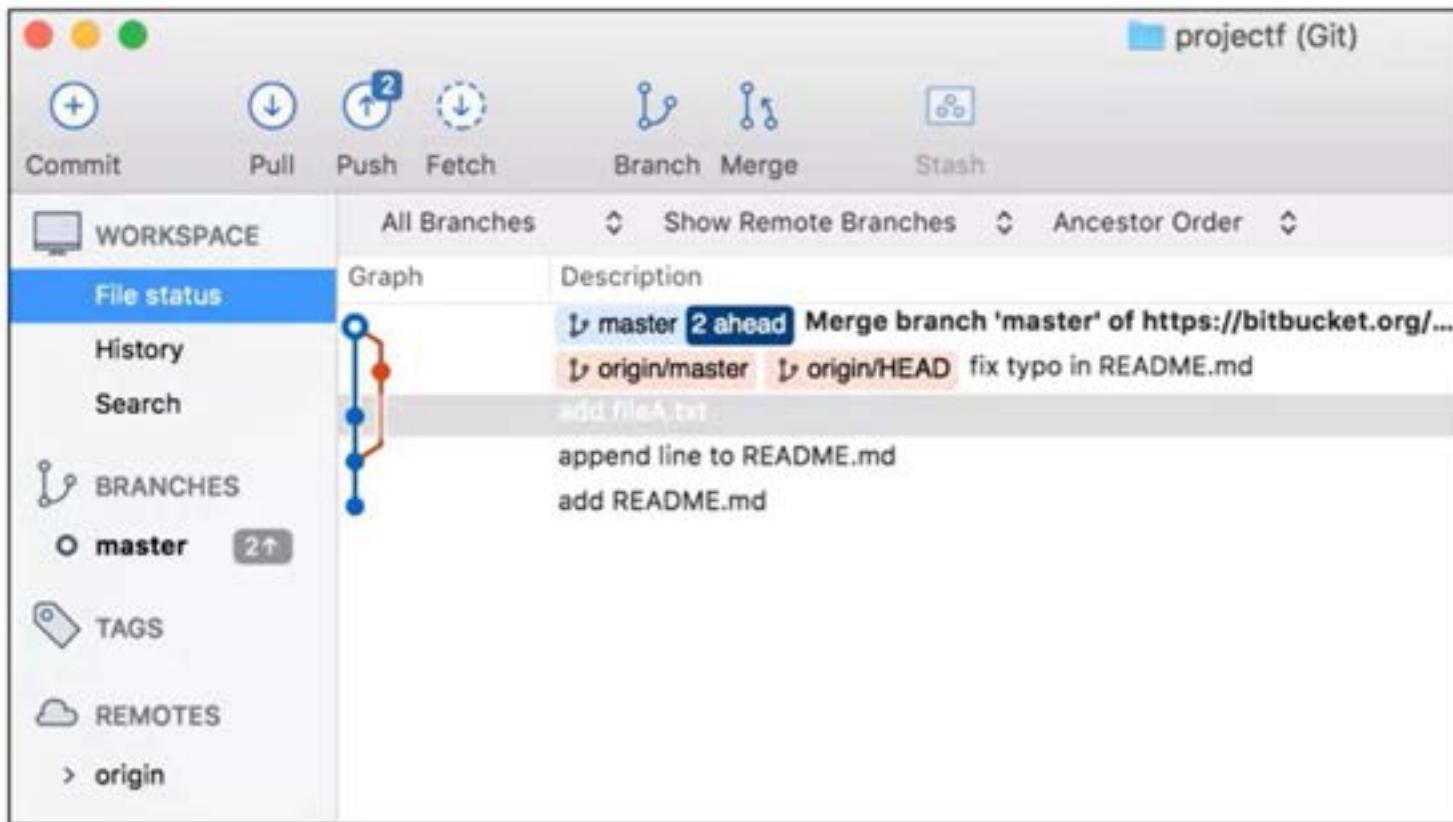


remote repository
(origin)

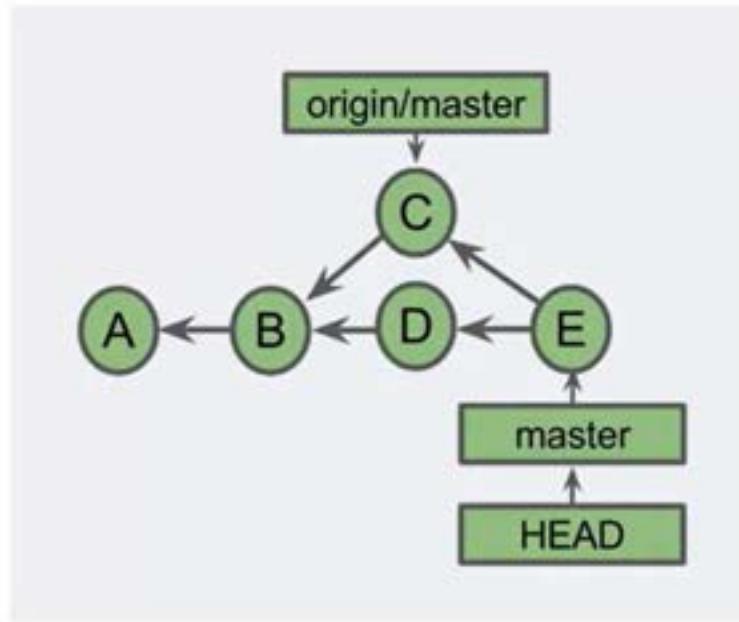
PULL WITH A MERGE COMMIT



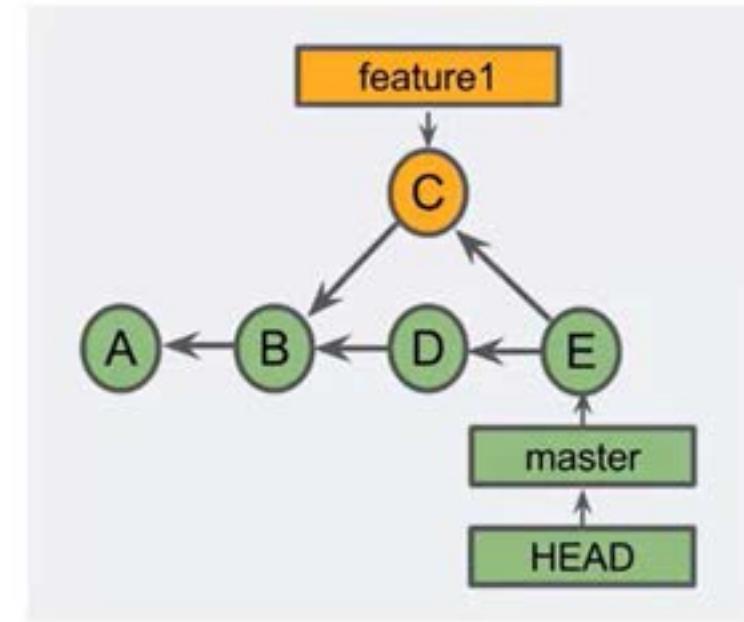
AFTER A PULL



THE TRACKING BRANCH IS LIKE A TOPIC BRANCH

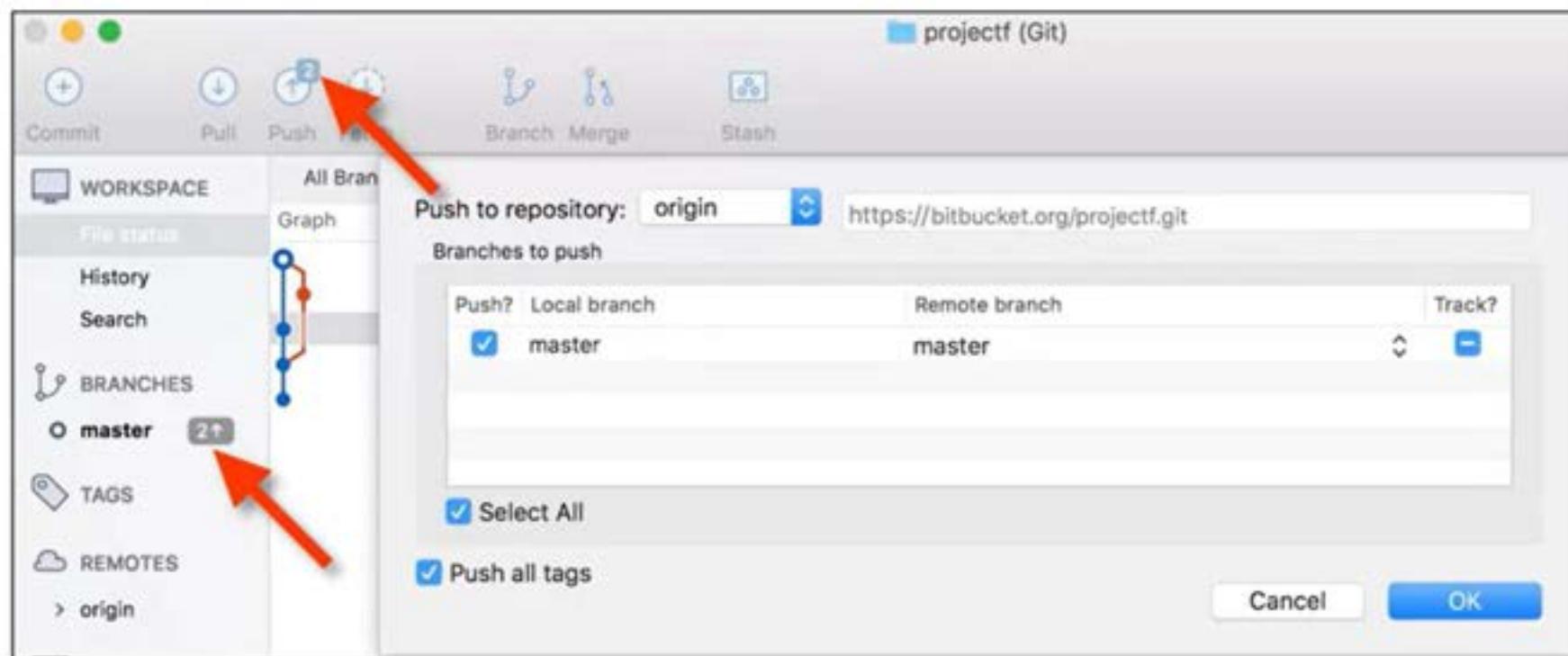


pull with merge
commit

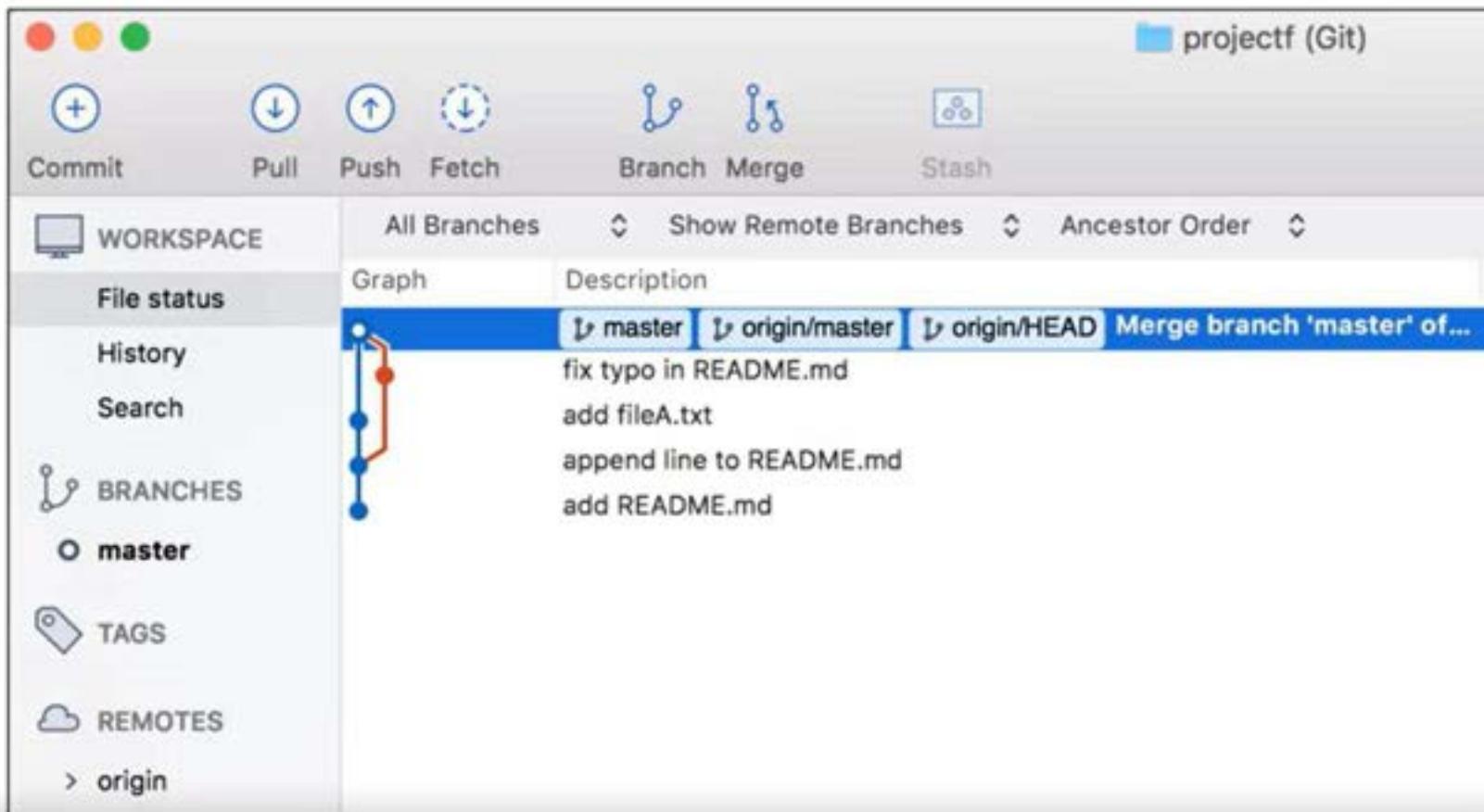


feature branch merge

PUSHING LOCAL COMMITS TO REMOTE

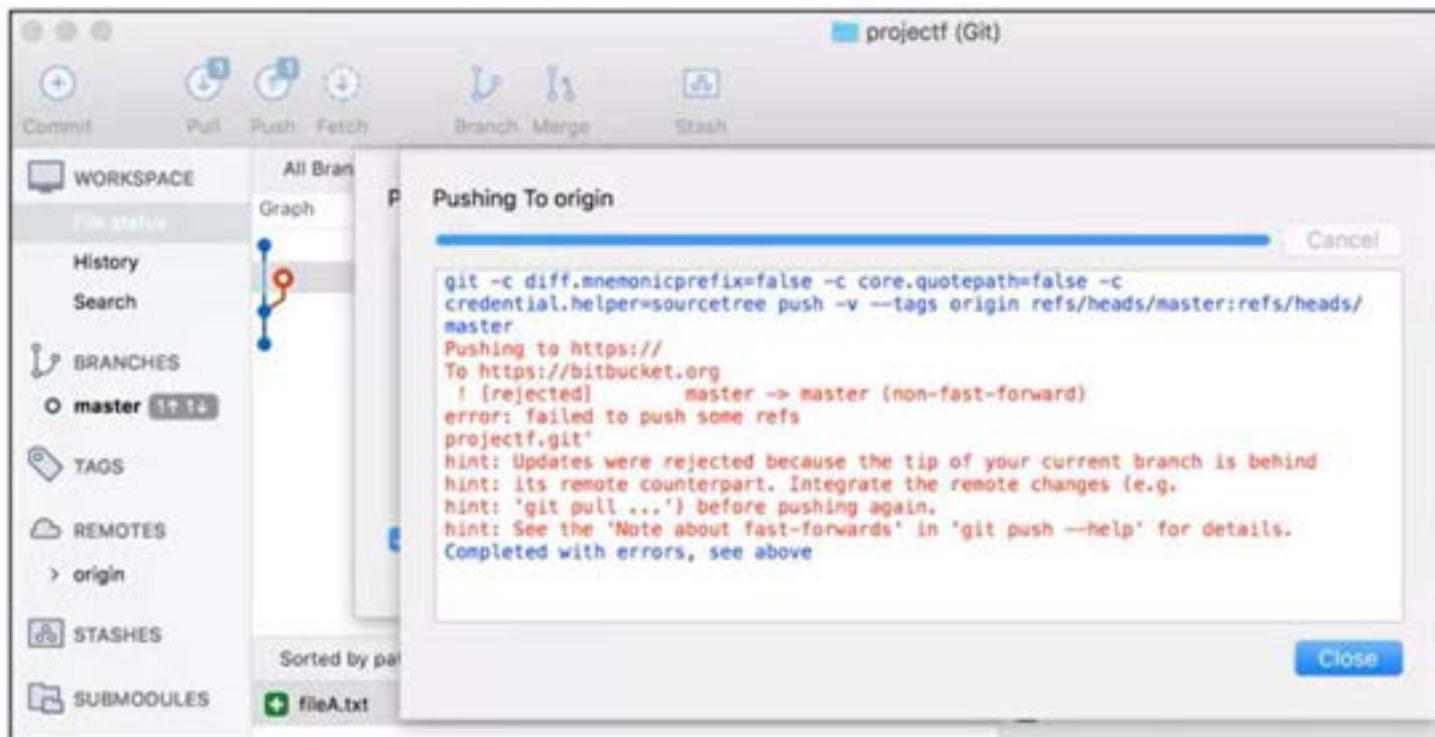


AFTER THE PUSH



FETCH OR PULL BEFORE PUSH

Fetching or pulling before you push is suggested



REVIEW

- Clone, fetch, pull and push commands communicate with a remote repository
- Fetch updates tracking branch information
- Pull combines a fetch and a merge
- Push adds commits to the remote repository

NETWORK COMMANDS

Clone- Copies a remote repository

Fetch- Retrieves new objects and references from the remote repository

Pull- Fetches and merges commits locally

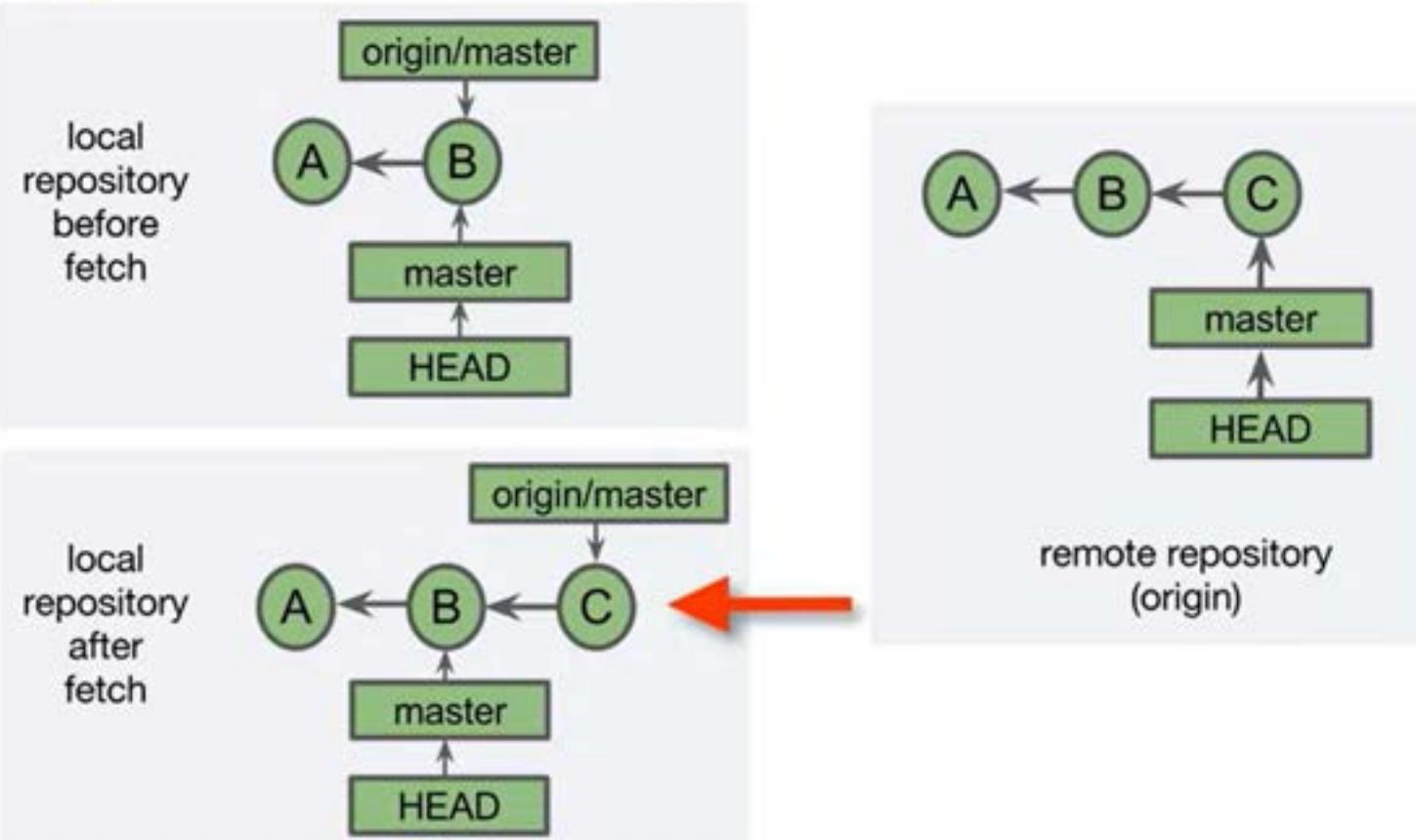
Push- Adds new objects and references to the remote repository

```
git fetch <repository>
```

- Retrieves new objects and references from another repository
- Tracking branches are updated

```
$ git log origin/master --oneline --graph --all
* 482b095 (HEAD -> master, origin/master) add feature 1
$ git fetch
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From https://bitbucket.org/me/projecta
  482b095..5ced2f3  master      -> origin/master
$ git log origin/master --oneline --graph --all
* 5ced2f3 (origin/master) add feature 2
* 482b095 (HEAD -> master) add feature 1
```

FETCHING THE LATEST COMMITS



AFTER git fetch

git status will inform you that your current branch is behind the tracking branch

```
$ git fetch
remote: Counting objects: 3, done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From https://bitbucket.org/me/projecta
  667fd0d..53d1b4d  master      -> origin/master
$ git status
On branch master
Your branch is behind 'origin/master' by 1 commit, and can be fast-forwarded.
  (use "git pull" to update your local branch)

nothing to commit, working tree clean
```

```
git pull [<repository>] [<branch>]
```

Combines git fetch and git merge FETCH_HEAD

- If objects are fetched, the tracking branch is merged into the current local branch
- This is similar to a topic branch merging into a base branch

```
$ git pull
Updating a65b42d..482b095
Fast-forward
  fileA.txt | 2 +-
  1 file changed, 1 insertion(+), 1 deletion(-)
```

git pull MERGING OPTIONS

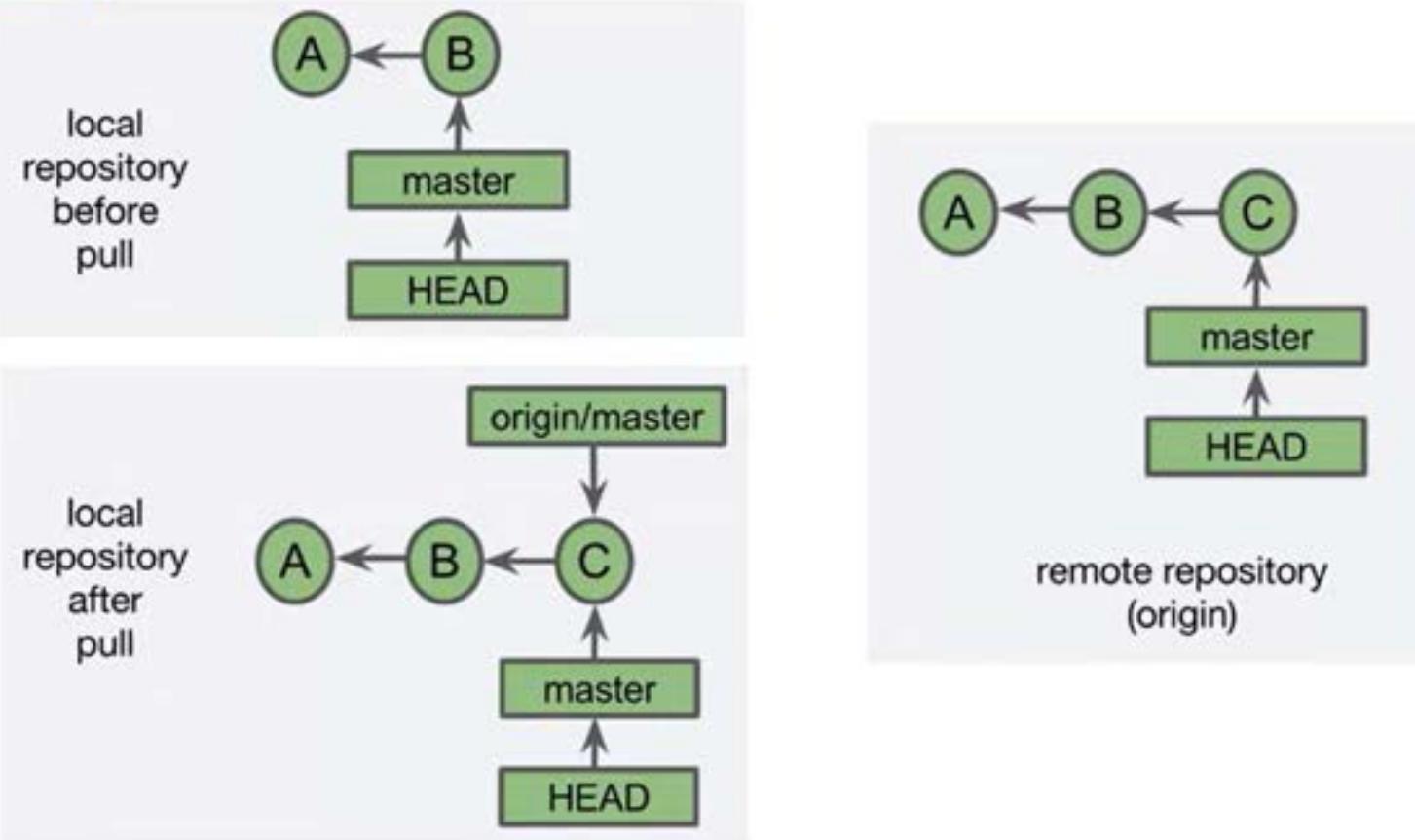
--ff (default) - fast-forward if possible, otherwise perform a merge commit

--no-ff - always include a merge commit

--ff-only - cancel instead of doing a merge commit

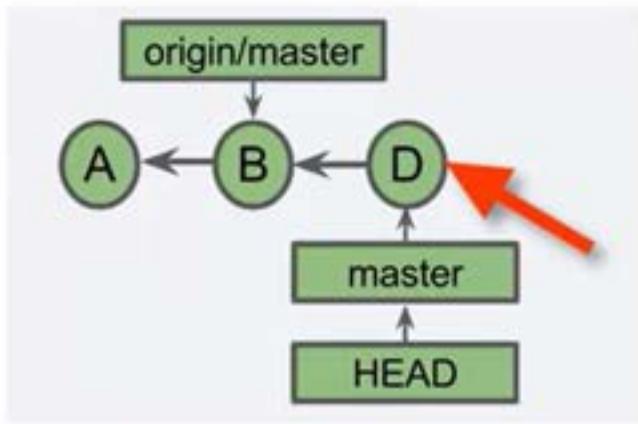
--rebase [--preserve-merges] - discussed later

PULL WITH A FAST-FORWARD MERGE

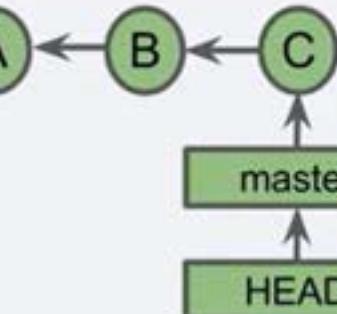
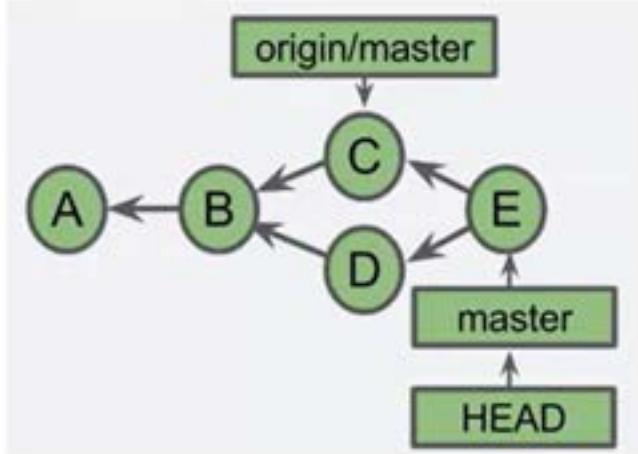


PULL WITH A MERGE COMMIT

local
repository
before
pull



local
repository
after
pull



remote repository
(origin)

git pull WITH A FAST-FORWARD MERGE

```
$ git status
On branch master
Your branch is behind 'origin/master' by 1 commit, and can be
fast-forwarded.
  (use "git pull" to update your local branch)

nothing to commit, working tree clean
$ git pull
Updating 667fd0d..53d1b4d
Fast-forward
  fileA.txt | 1 +
  1 file changed, 1 insertion(+)
$ git log --oneline
53d1b4d (HEAD -> master, origin/master) added feature 3
667fd0d added feature 2
5d5e128 initial commit
```

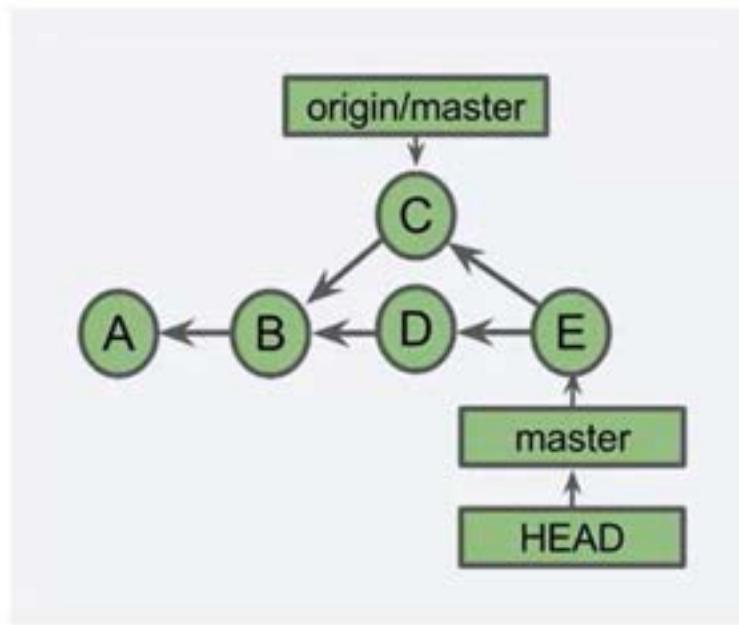
git pull WITH CONFLICTING UNCOMMITTED CHANGES

```
$ echo "feature4" >> fileA.txt
$ git pull
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From https://bitbucket.org/me/projecta
  53d1b4d..63f4add master      -> origin/master
Updating 53d1b4d..63f4add
error: Your local changes to the following files would be overwritten by merge:
  fileA.txt
Please commit your changes or stash them before you merge.
Aborting
```

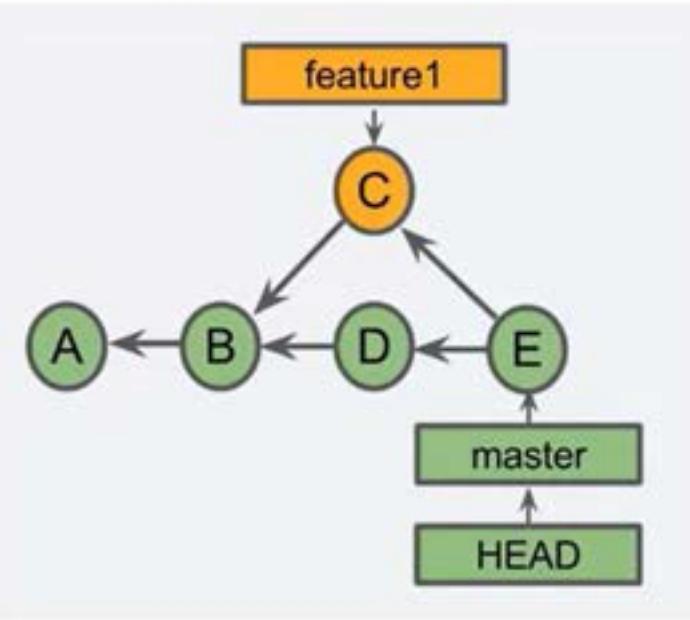
git pull WITH SAFE UNCOMMITTED CHANGES

```
$ touch fileB.txt # new file
$ git pull
Updating 53d1b4d..63f4add
Fast-forward
  fileA.txt | 1 +
  1 file changed, 1 insertion(+)
$ ls
fileA.txt      fileB.txt
```

THE TRACKING BRANCH IS LIKE A TOPIC BRANCH



pull with merge
commit



feature branch merge

git pull WITH A MERGE

```
$ touch fileC.txt # new file
$ git add fileC.txt
$ git commit -m "added fileC.txt"
[master 6209be3] added fileC.txt
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 fileC.txt
$ git pull
remote: Counting objects: 3, done.
(snip)
From https://bitbucket.org/me/projecta
  c2ccd19..6d5539b  master      -> origin/master
Merge made by the 'recursive' strategy.
 fileA.txt | 1 +
 1 file changed, 1 insertion(+)
$ git log --oneline --graph -4
*   c090487 (HEAD -> master) Merge branch 'master' of https://bitbucket.org/me/projecta
 |\ 
| * 6d5539b (origin/master) added feature 5
* | 6209be3 added fileC.txt
|/
* c2ccd19 added fileB.txt
```

PUSHING LOCAL COMMITS TO REMOTE

git push [-u] [<repository>] [<branch>]

- -u Track this branch (--set-upstream)

```
$ git push -u origin master
Username for 'https://me@bitbucket.org': me # you may need to set up an app pw
Password for 'https://me@bitbucket.org':
Counting objects: 3, done.
Writing objects: 100% (3/3), 223 bytes | 223.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://bitbucket.org/me/_projecta.git
 * [new branch]      master -> master
Branch master set up to track remote branch master from origin.
```

FETCH OR PULL BEFORE PUSH

Fetching or pulling before you push is suggested

```
(create a commit on the remote repository)
$ touch fileB.txt
$ git add fileB.txt
$ git commit -m "add fileB.txt"
[master 07c5e2a] add fileB.txt
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 fileB.txt
$ git push
To https://bitbucket.org/me/projectf.git
 ! [rejected]      master -> master (fetch first)
error: failed to push some refs to 'https://me@bitbucket.org/me/projectf.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

Topics

Rebasing overview

Executing a rebase

Rebasing with merge conflicts

Press `Esc` to exit full screen

REWRITING COMMIT HISTORY

- The topics discussed here rewrite the commit history
- This should be done with caution
- General rule: Do not rewrite history that has been shared with others



TWO TYPES OF REBASE

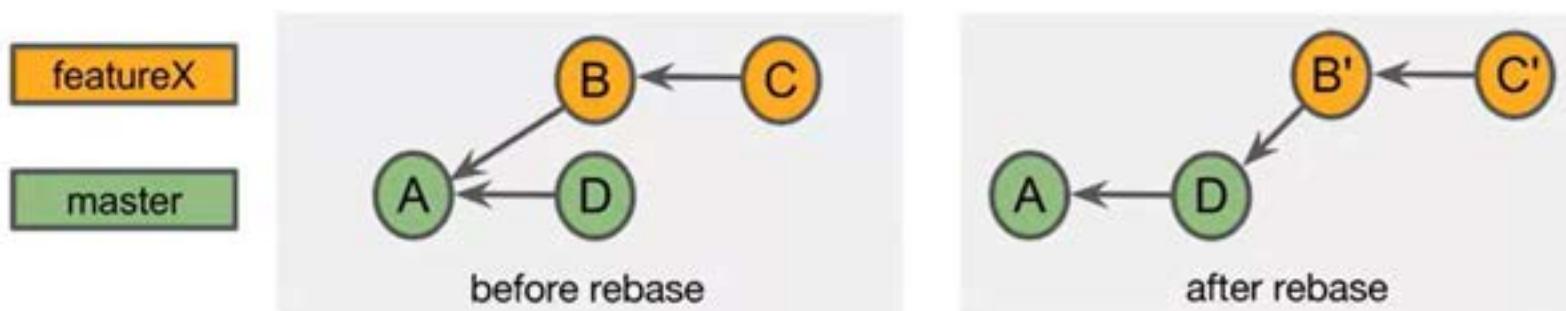
1. Rebase
2. Interactive rebase

These can be very different, as we will see later...

REBASE

Moves commits to a new parent (base)

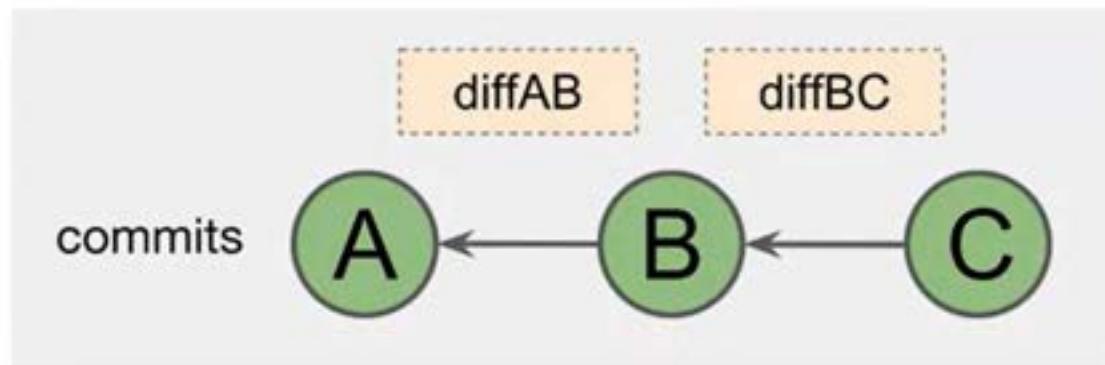
- The unique commits of the featureX branch (B and C) are reapplied to the tip of the master branch (D)
- Because the ancestor chain is different, each of the reapplied commits has a different commit ID (B' and C')



<https://git-scm.com/docs/git-rebase>

DIFFS

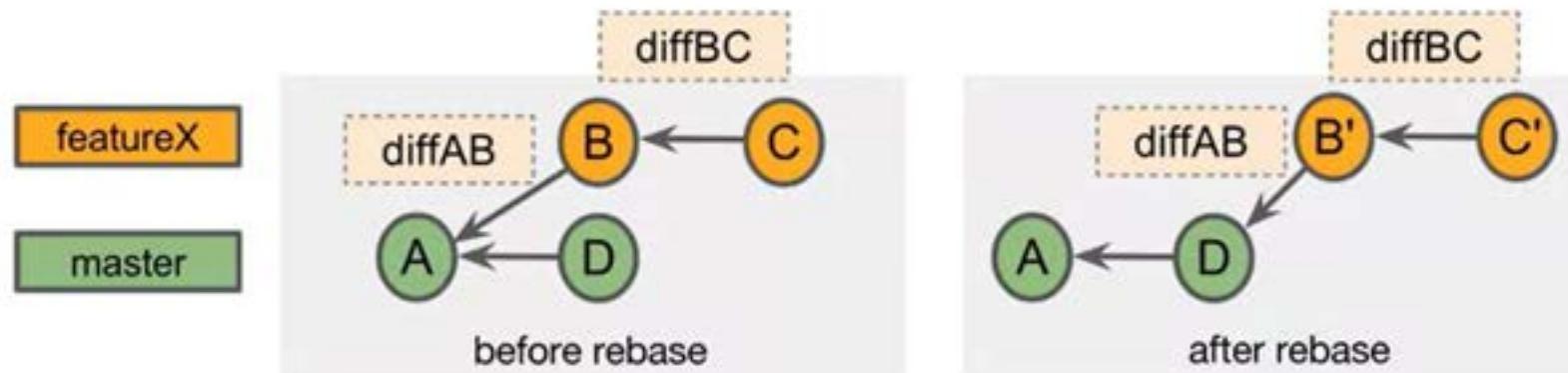
- Each commit contains a snapshot of the complete project
- Git can *calculate* the difference between commits
 - This is known as a *diff* or a *patch*



REBASING REAPPLIES COMMITS

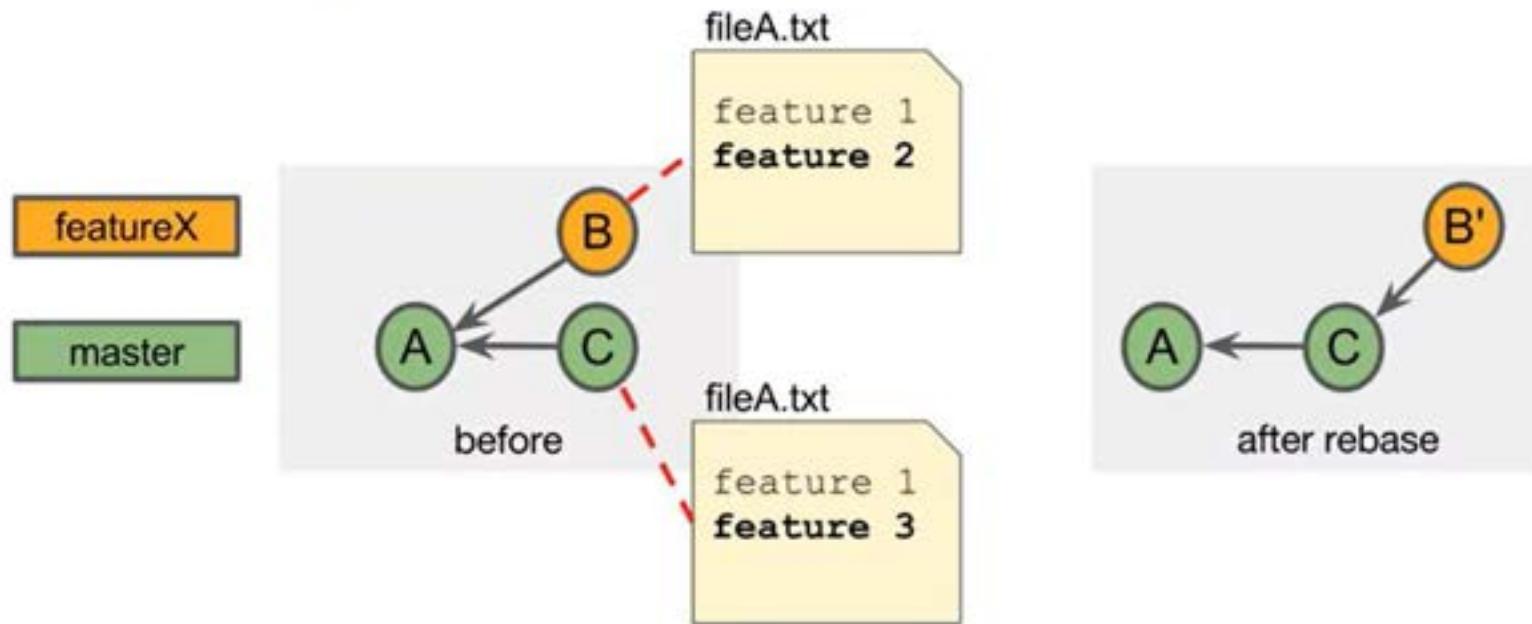
When rebasing, Git applies the diffs to the new parent commit

- This is called "reapplying commits"



REBASING IS A MERGE

- Reapplying commits is a form of merge and is susceptible to merge conflicts
- For example, commits B and C can change the same file, causing a merge conflict during the rebase



REBASING PROS AND CONS

- Pros:

- You can incorporate changes from the parent branch
 - You can use the new features/bugfixes
 - Tests are on more current code
 - It makes the eventual merge into master fast-forwardable
- Avoids "unnecessary" commits
 - It allows you to shape/define clean commit histories

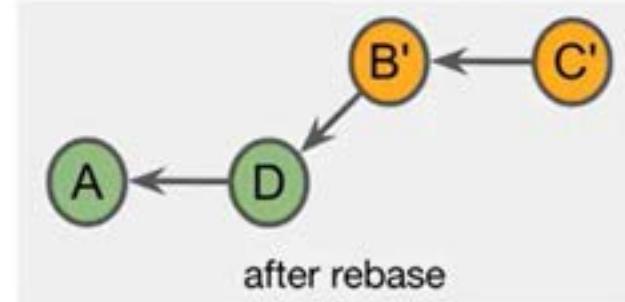
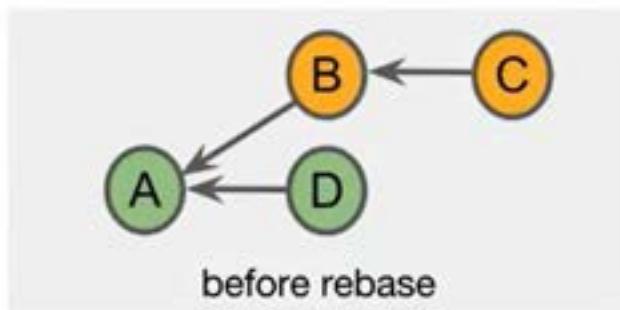
- Cons:

- Merge conflicts may need to be resolved
- It can cause problems if your commits have been shared
- You are not preserving the commit history

EXECUTING A REBASE

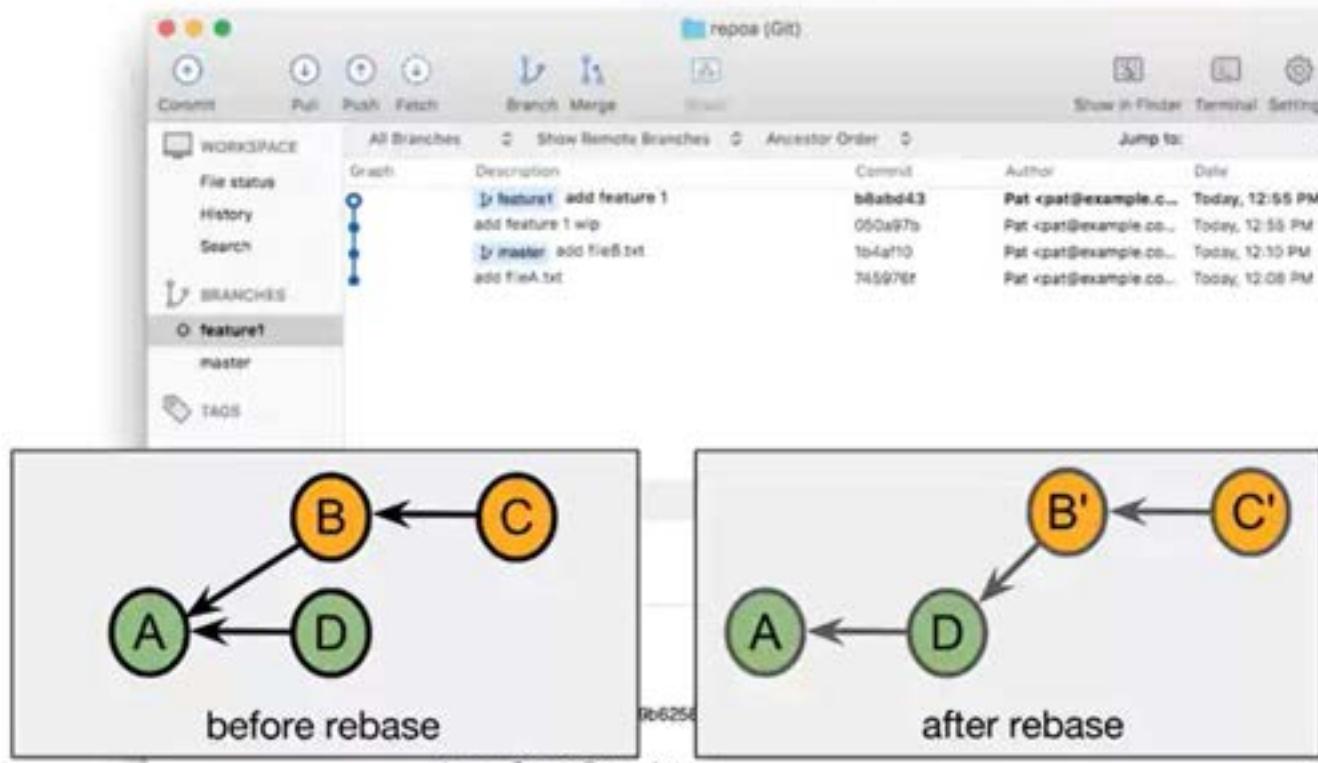
1. Checkout featureX
2. Rebase onto master

featureX
master



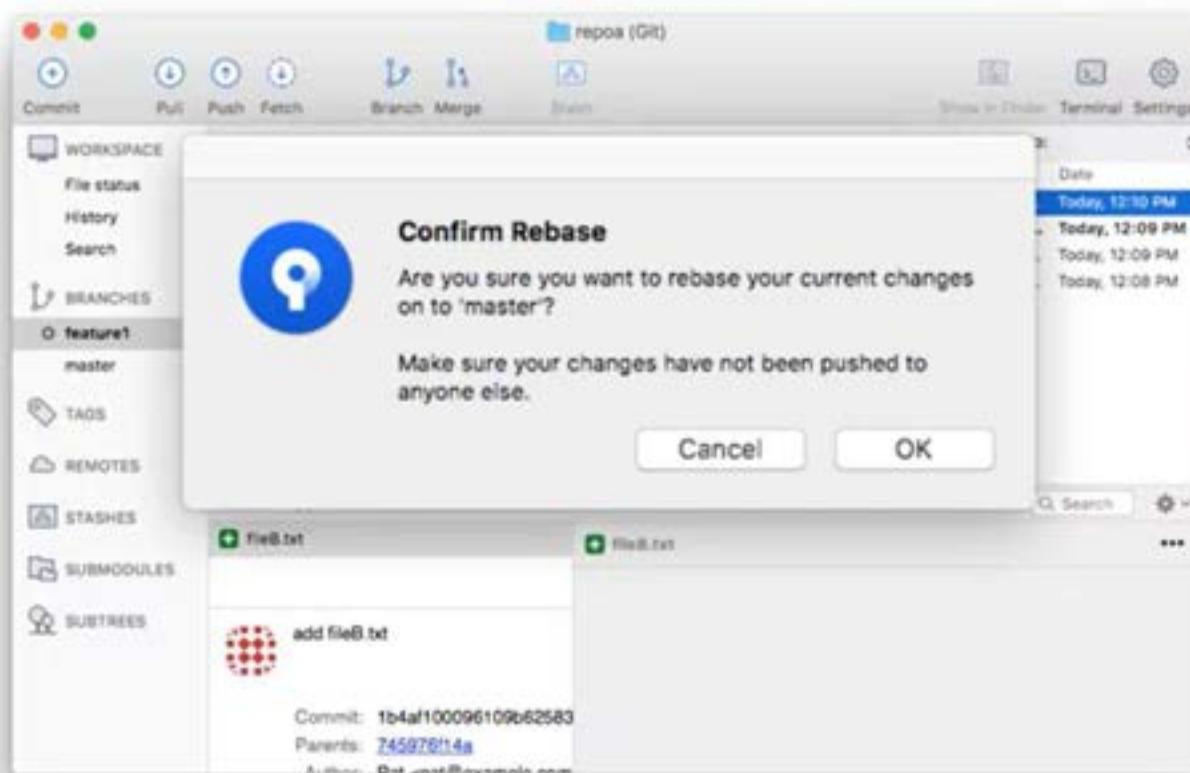
REBASE

1. Checkout featureX
2. Rebase onto master



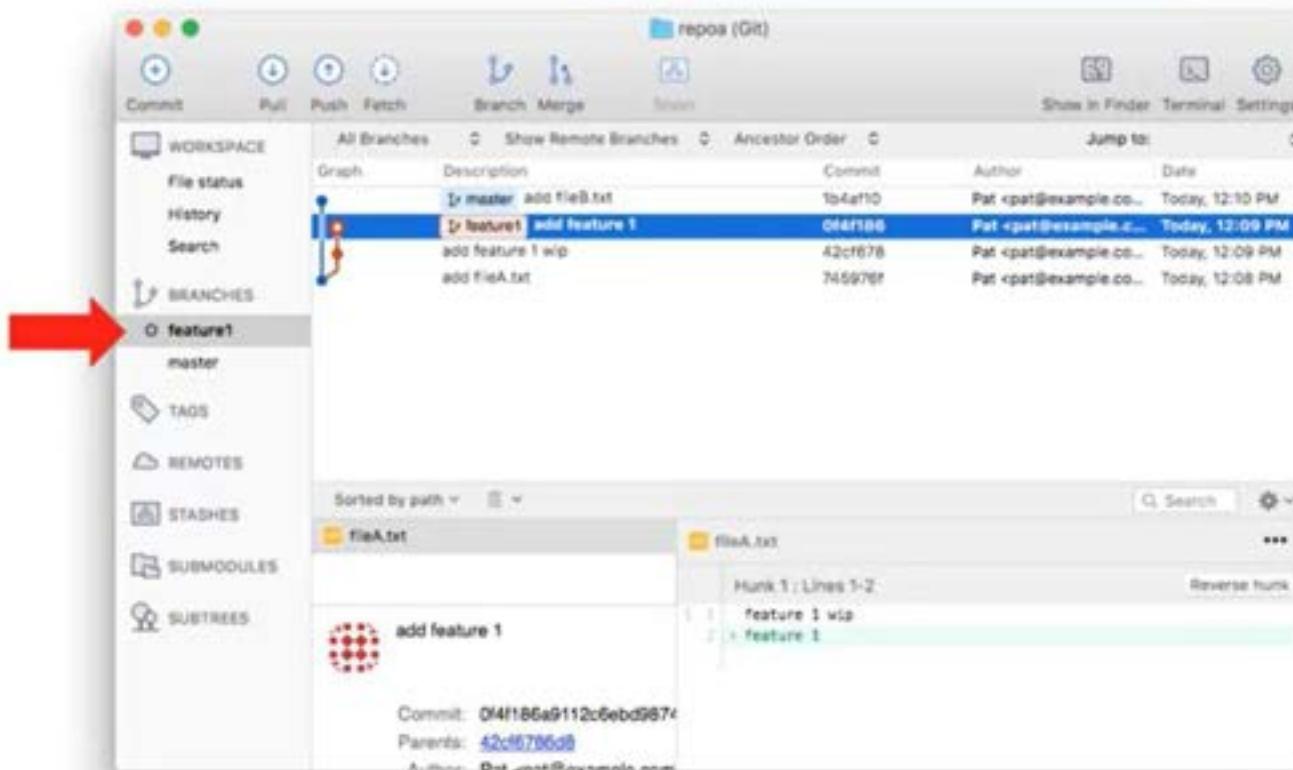
REBASE

1. Checkout featureX
2. Rebase onto master



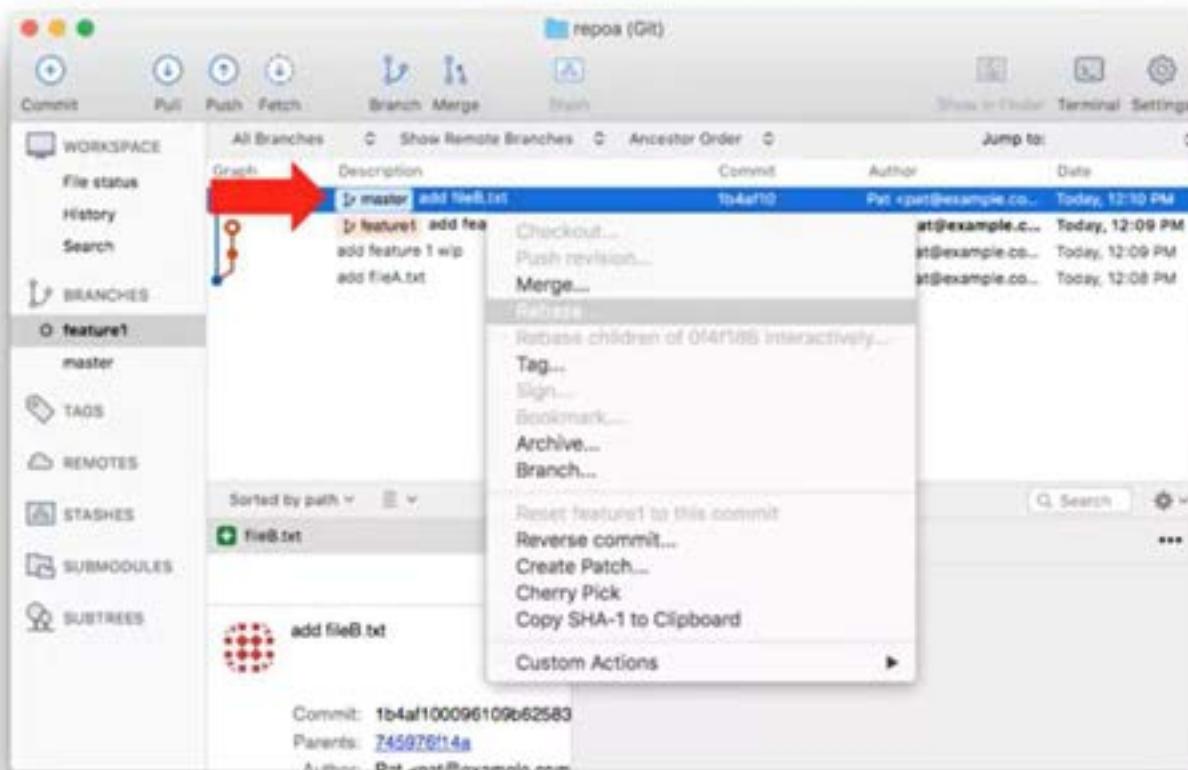
REBASE

1. Checkout featureX
2. Rebase onto master



REBASE

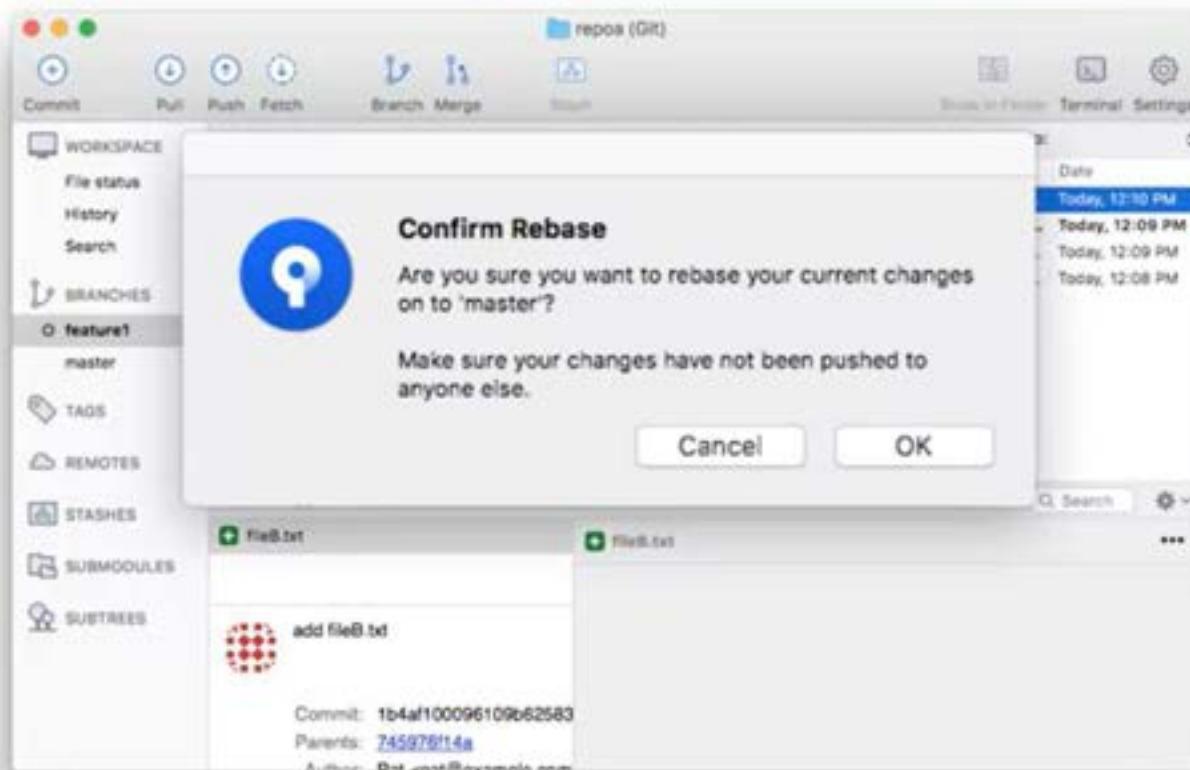
1. Checkout featureX
2. Rebase onto master



With the **featureX** branch checked out, select the **master** branch and **Rebase...**

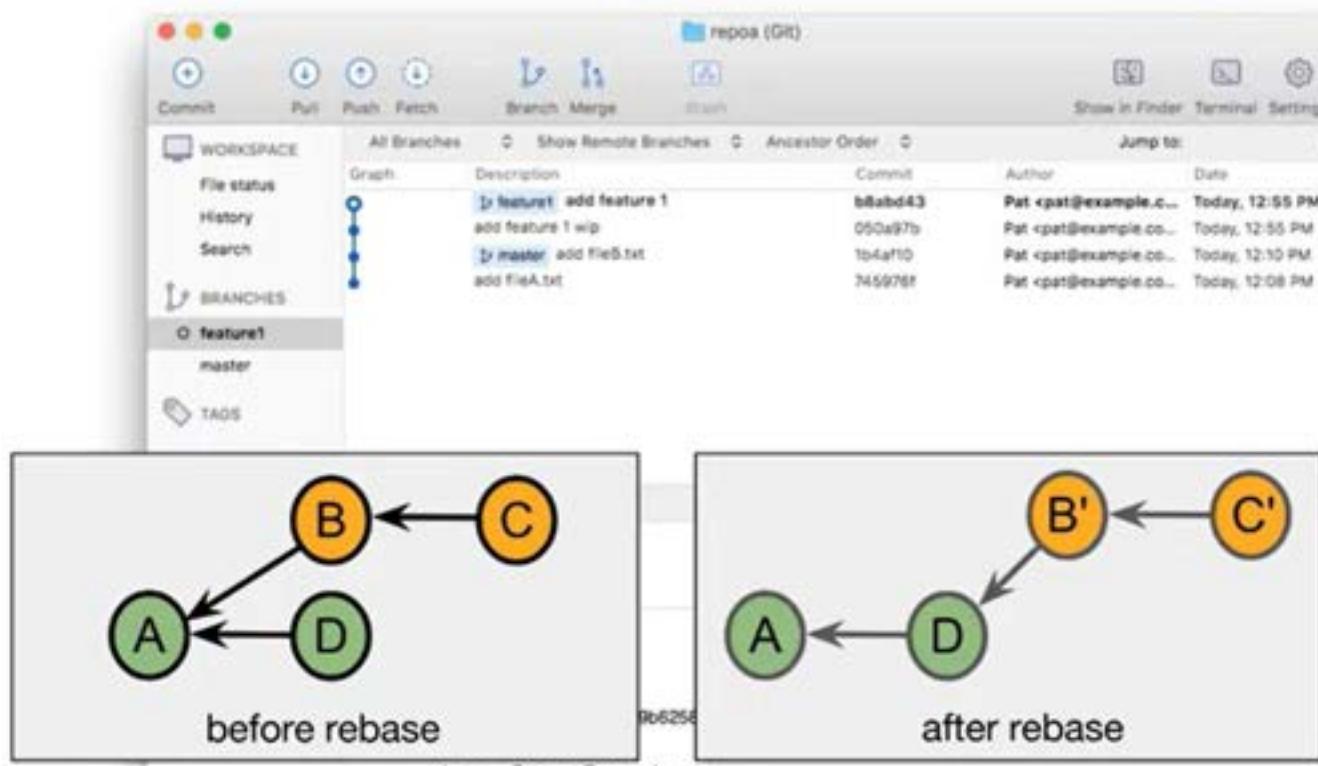
REBASE

1. Checkout featureX
2. Rebase onto master



REBASE

1. Checkout featureX
2. Rebase onto master



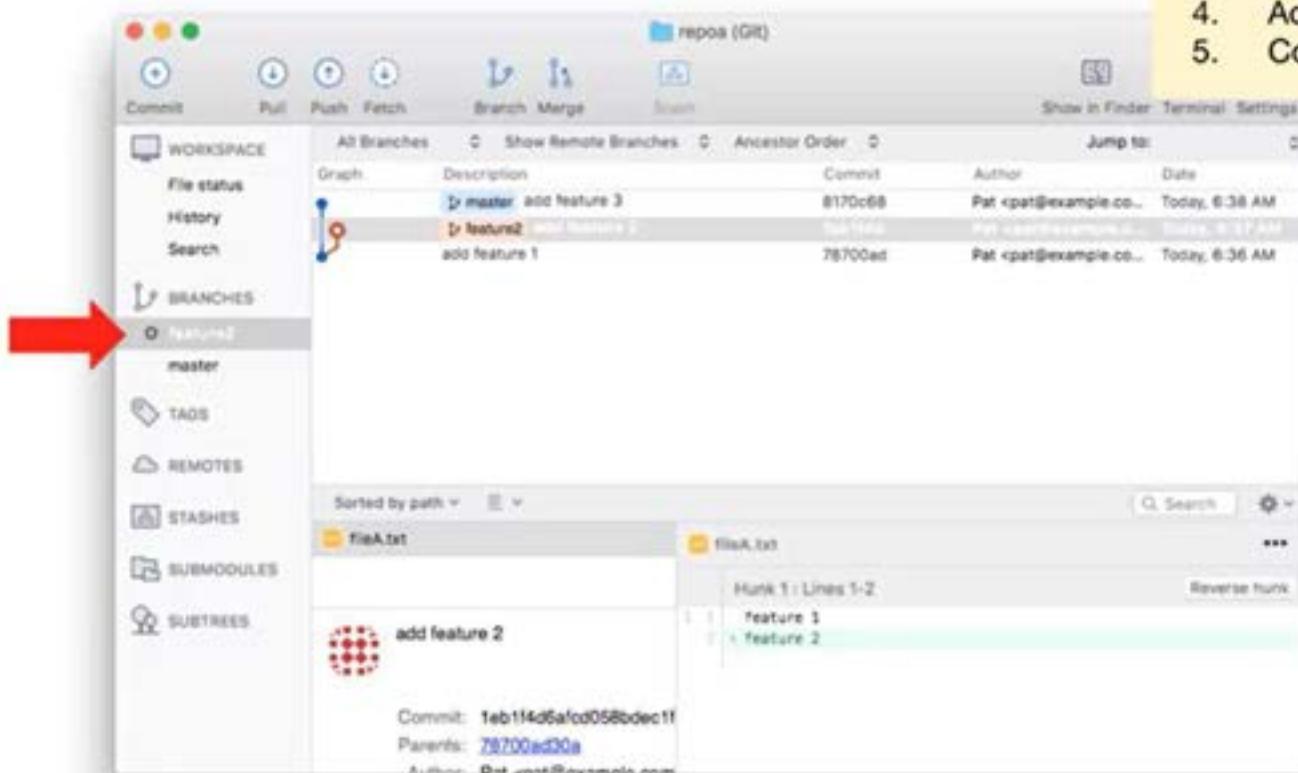
FIXING A MERGE CONFLICT WHILE REBASING

1. Checkout **featureX**
2. **Rebase onto master**
 - a. CONFLICT- both modified fileA.txt
3. Fix fileA.txt
4. **Add fileA.txt**
5. **Continue rebase**

Files with conflicts are **modified by Git** in the working tree

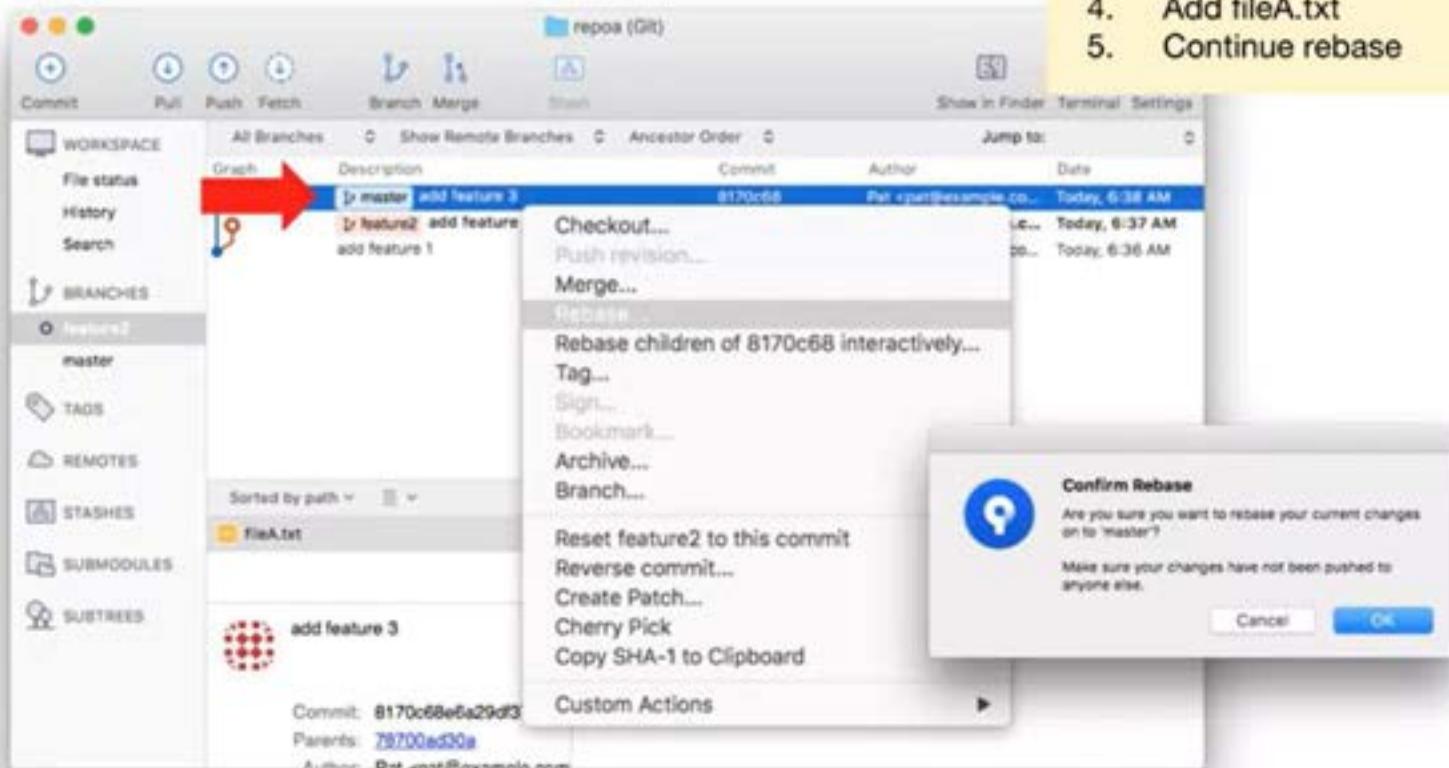
REBASE WITH MERGE CONFLICT

1. Checkout featureX
2. Rebase onto master
 - a. CONFLICT
3. Fix fileA.txt
4. Add fileA.txt
5. Continue rebase



REBASE WITH MERGE CONFLICT

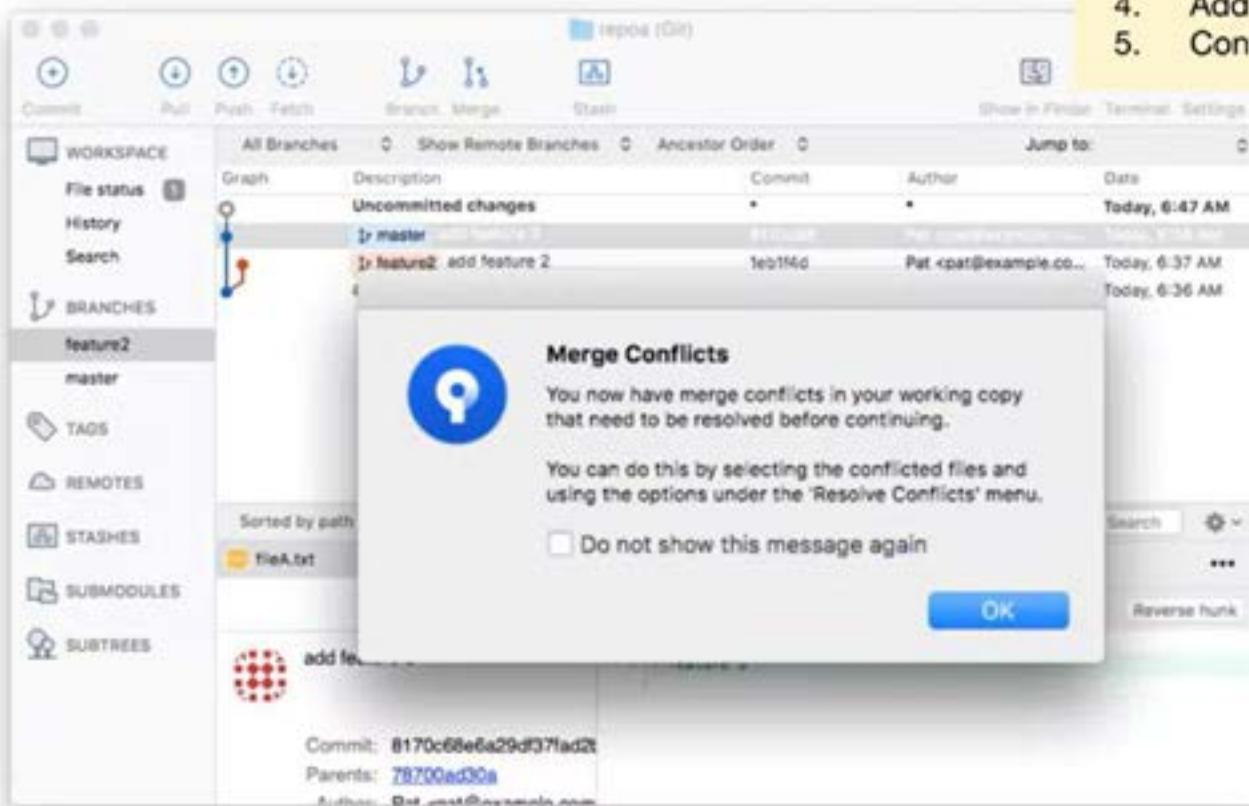
1. Checkout featureX
2. Rebase onto master
 - a. CONFLICT
3. Fix fileA.txt
4. Add fileA.txt
5. Continue rebase



With the **featureX** branch checked out, select the **master** branch and **Rebase...**

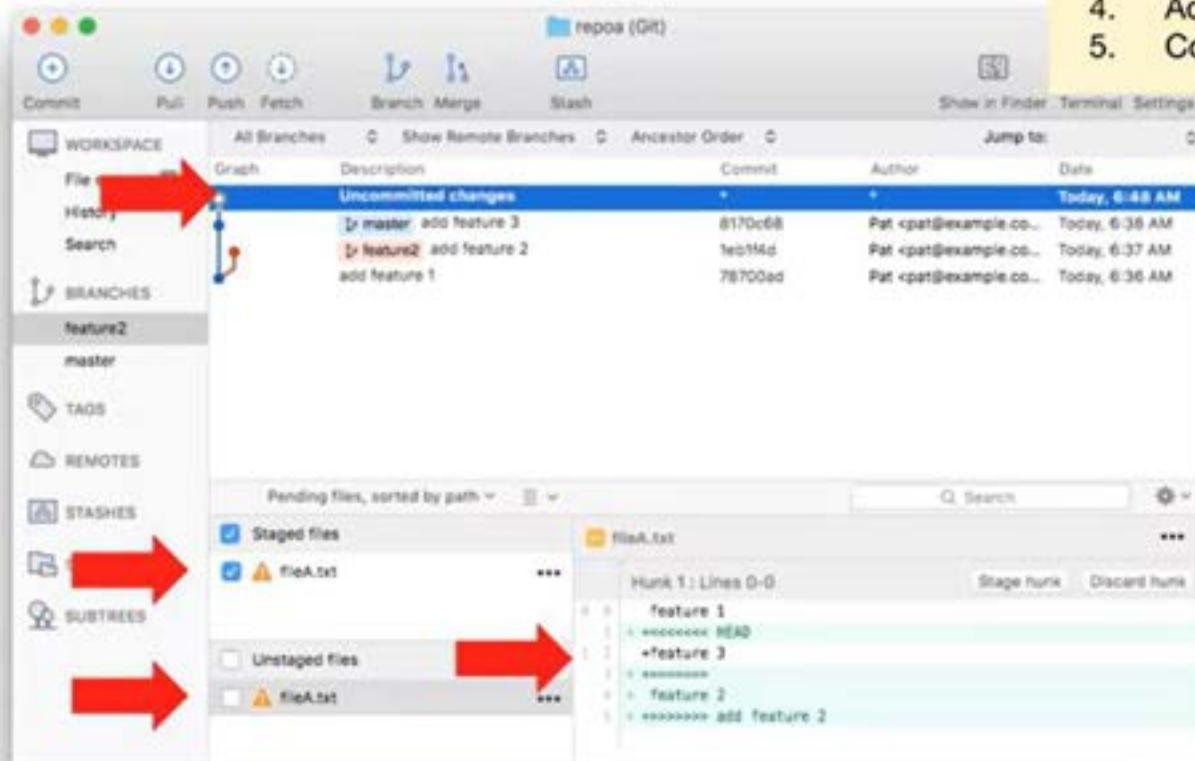
REBASE WITH MERGE CONFLICT

1. Checkout featureX
2. Rebase onto master
 - a. **CONFLICT**
3. Fix fileA.txt
4. Add fileA.txt
5. Continue rebase



REBASE WITH MERGE CONFLICT

1. Checkout featureX
2. Rebase onto master
 - a. **CONFLICT**
3. Fix fileA.txt
4. Add fileA.txt
5. Continue rebase



REBASE WITH MERGE CONFLICT

1. Checkout featureX
2. Rebase onto master
 - a. CONFLICT
3. Fix fileA.txt
4. Add fileA.txt
5. Continue rebase

The image shows two screenshots of a terminal window side-by-side, connected by a large red arrow pointing from left to right.

Left Screenshot: The terminal window title is "fileA.txt — Edited". The content of the file is:

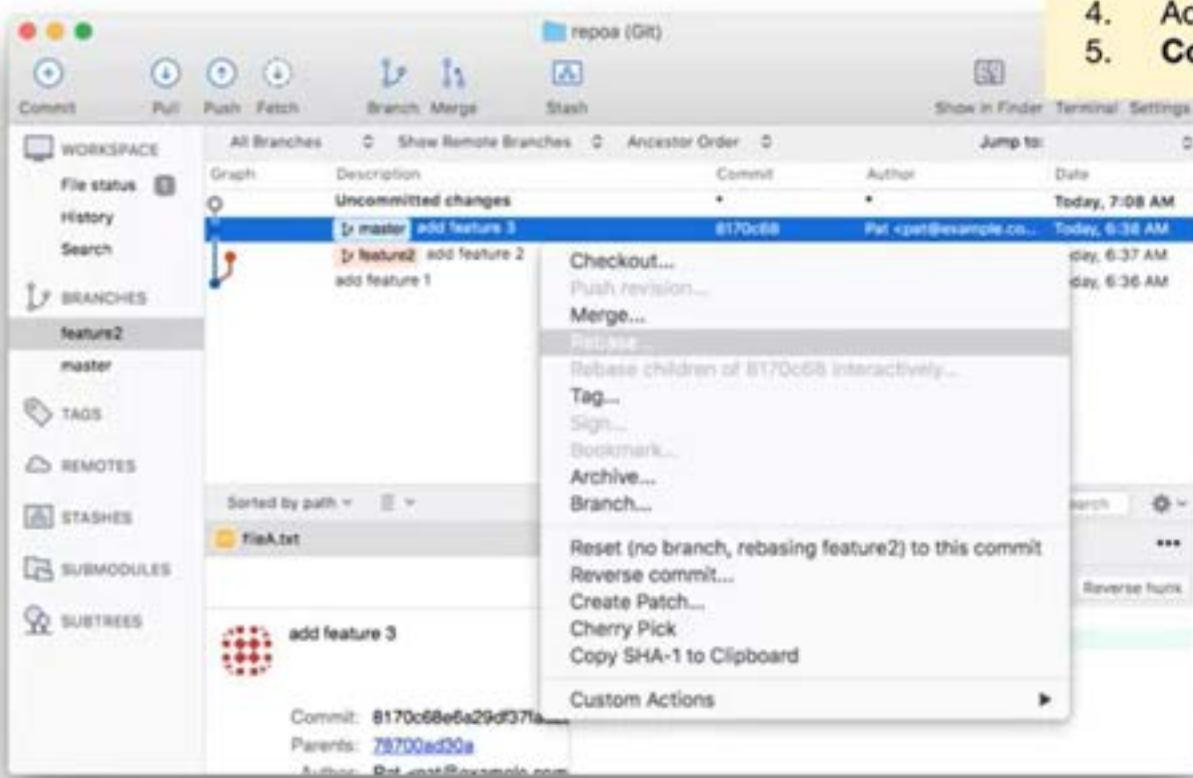
```
feature 1
<<<<< HEAD
feature 3
=====
feature 2
>>>>> add feature 2
```

Right Screenshot: The terminal window title is "fileA.txt — Edited". The content of the file has been modified:

```
feature 1
feature 2
feature 3
```

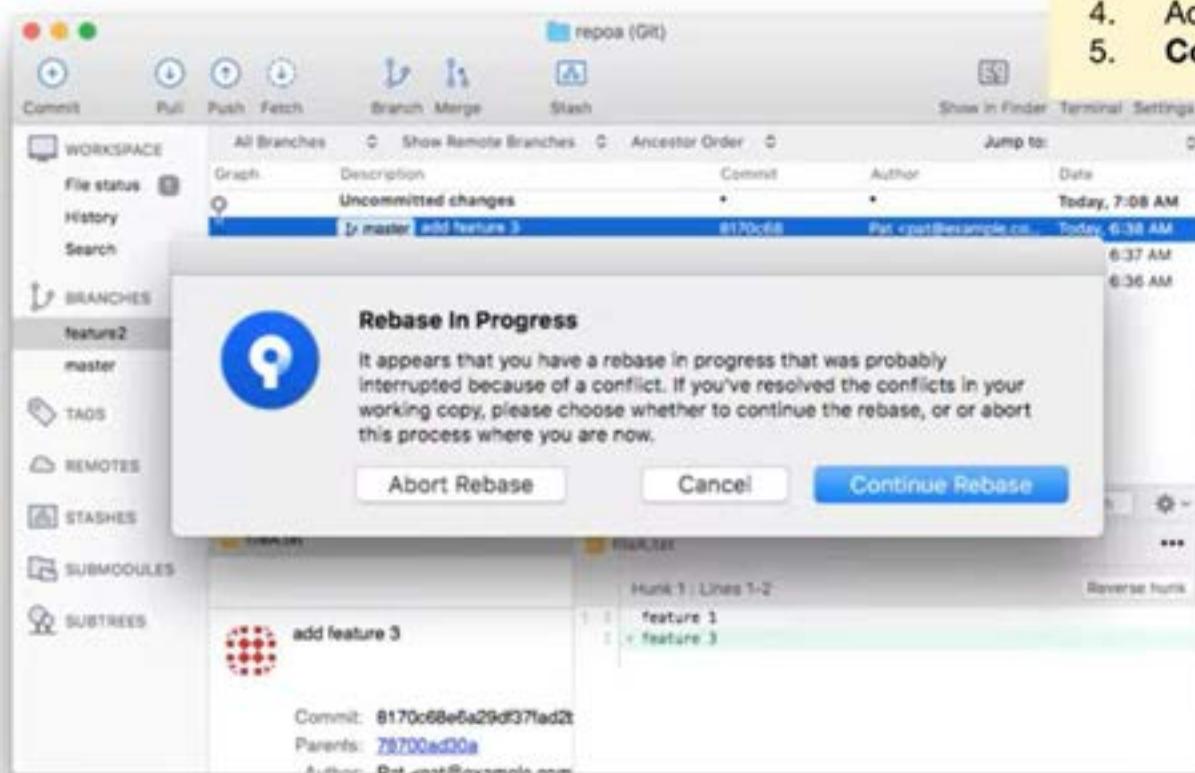
REBASE WITH MERGE CONFLICT

1. Checkout featureX
2. Rebase onto master
 - a. CONFLICT
3. Fix fileA.txt
4. Add fileA.txt
5. Continue rebase



REBASE WITH MERGE CONFLICT

1. Checkout featureX
2. Rebase onto master
 - a. CONFLICT
3. Fix fileA.txt
4. Add fileA.txt
5. Continue rebase



REBASE WITH MERGE CONFLICT

1. Checkout featureX
2. Rebase onto master
 - a. CONFLICT
3. Fix fileA.txt
4. Add fileA.txt
5. Continue rebase

The screenshot shows a Git interface with a commit history table and a commit graph diagram.

Commit History Table:

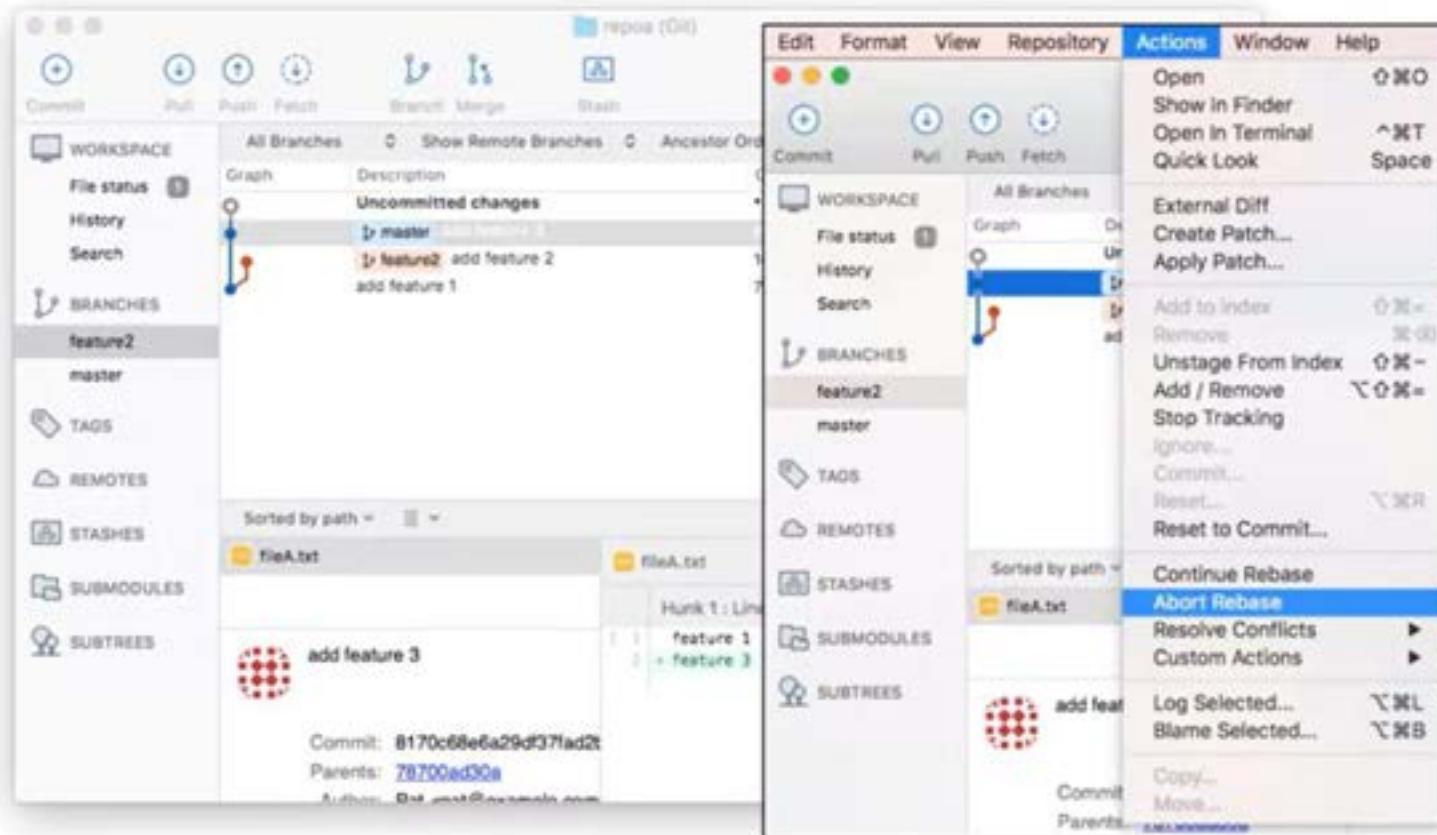
Description	Commit	Author	Date
[feature2] add feature 2	0620d7f	Pat <pat@example.c...	Today, 7:16 AM
[master] add feature 3	8170c68	Pat <pat@example.co...	Today, 6:38 AM
add feature 1	78700ad	Pat <pat@example.co...	Today, 6:36 AM

Commit Graph Diagram:

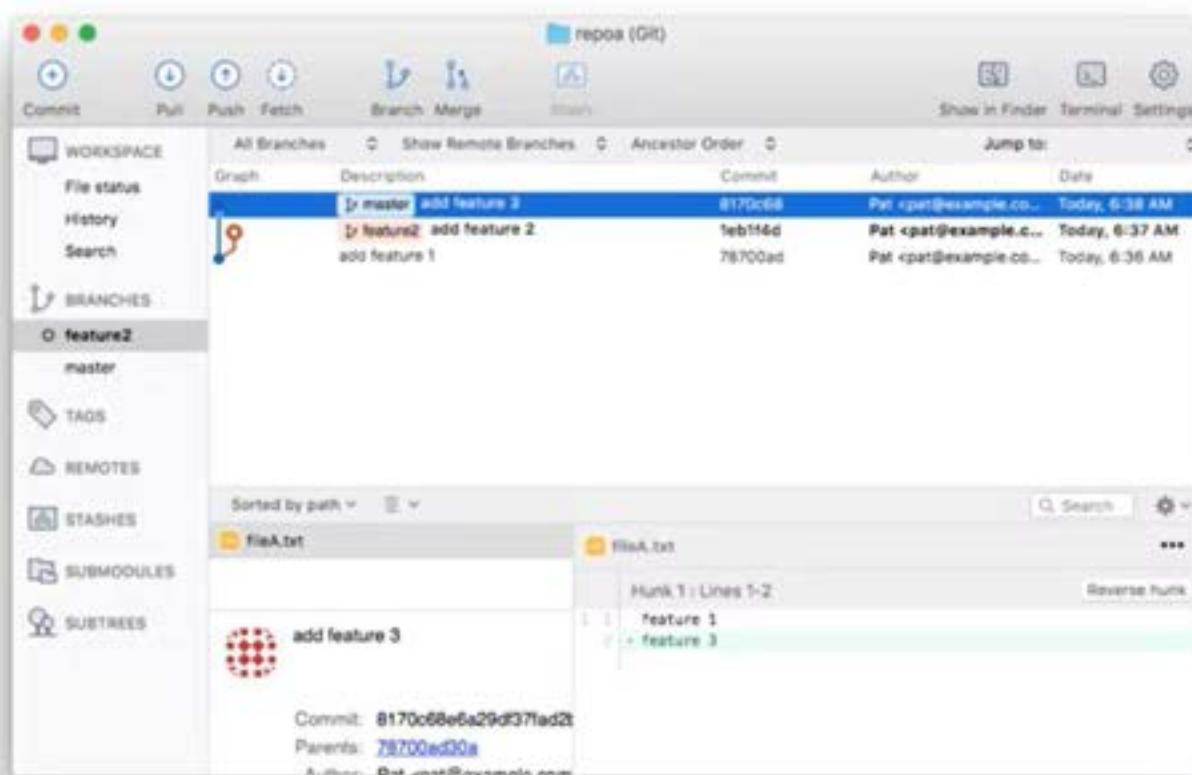
before rebase: A graph showing three commits: A (green), B (orange), and C (green). Commit B is the root, with arrows pointing from A and C to it.

after rebase: A graph showing three commits: A (green), C (green), and B' (orange). Commit B' is the root, with arrows pointing from A and C to it.

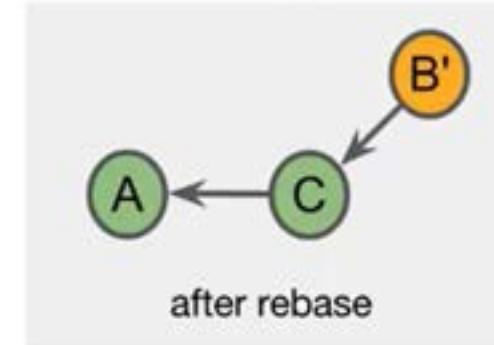
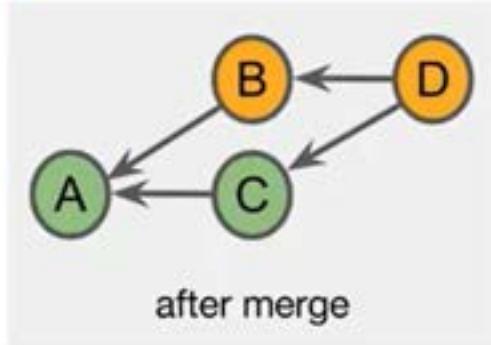
ABORTING A REBASE



ABORTING A REBASE- BACK TO THE ORIGINAL STATE



RESOLVING MERGE CONFLICTS- COMPARING MERGE TO REBASE



1. Checkout **master**
2. **Merge featureX**
 - a. CONFLICT
3. Fix fileA.txt
4. Add fileA.txt
5. **Commit the merge**

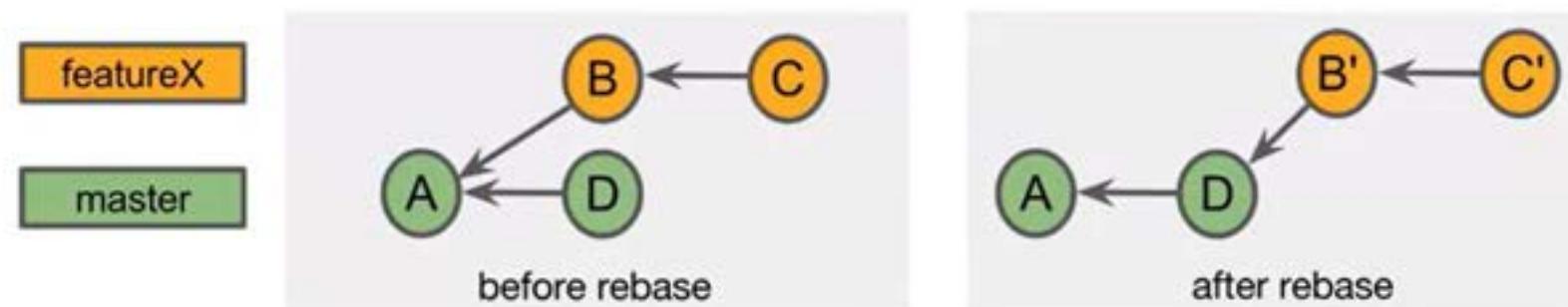
merge

1. Checkout **featureX**
2. **Rebase onto master**
 - a. CONFLICT
3. Fix fileA.txt
4. Add fileA.txt
5. **Continue rebase**

rebase

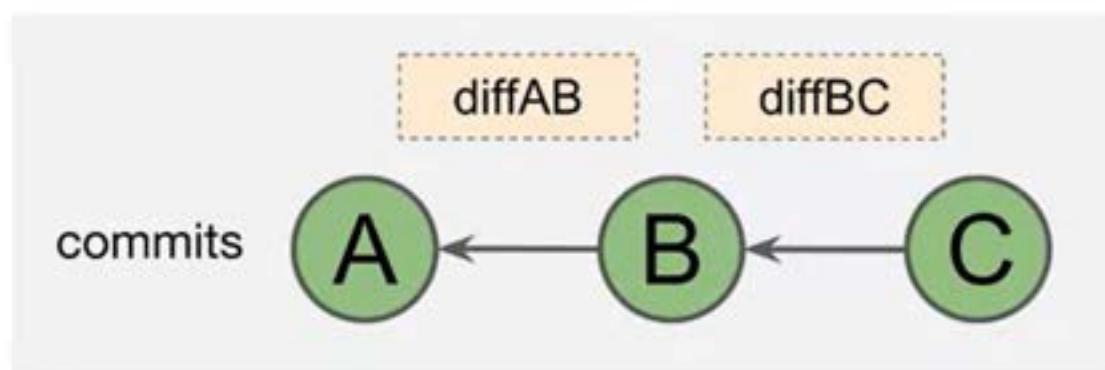
REVIEW

- Rebasing moves a branch to the tip of another branch
- Rebasing is a form of merge and may result in merge conflicts



DIFFS

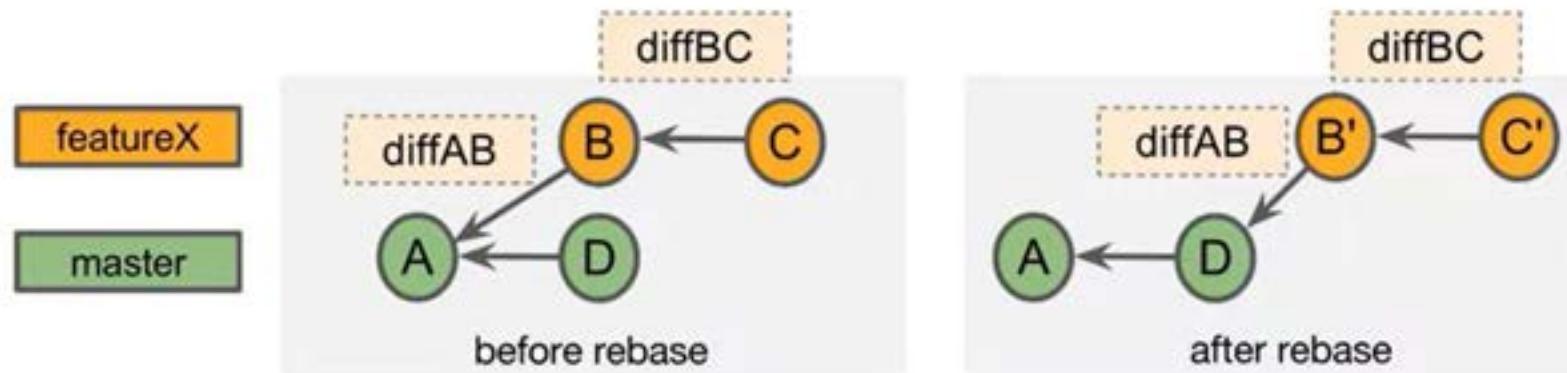
- Each commit contains a snapshot of the complete project
- Git can *calculate* the difference between commits
 - This is known as a *diff* or a *patch*



REBASING REAPPLIES COMMITS

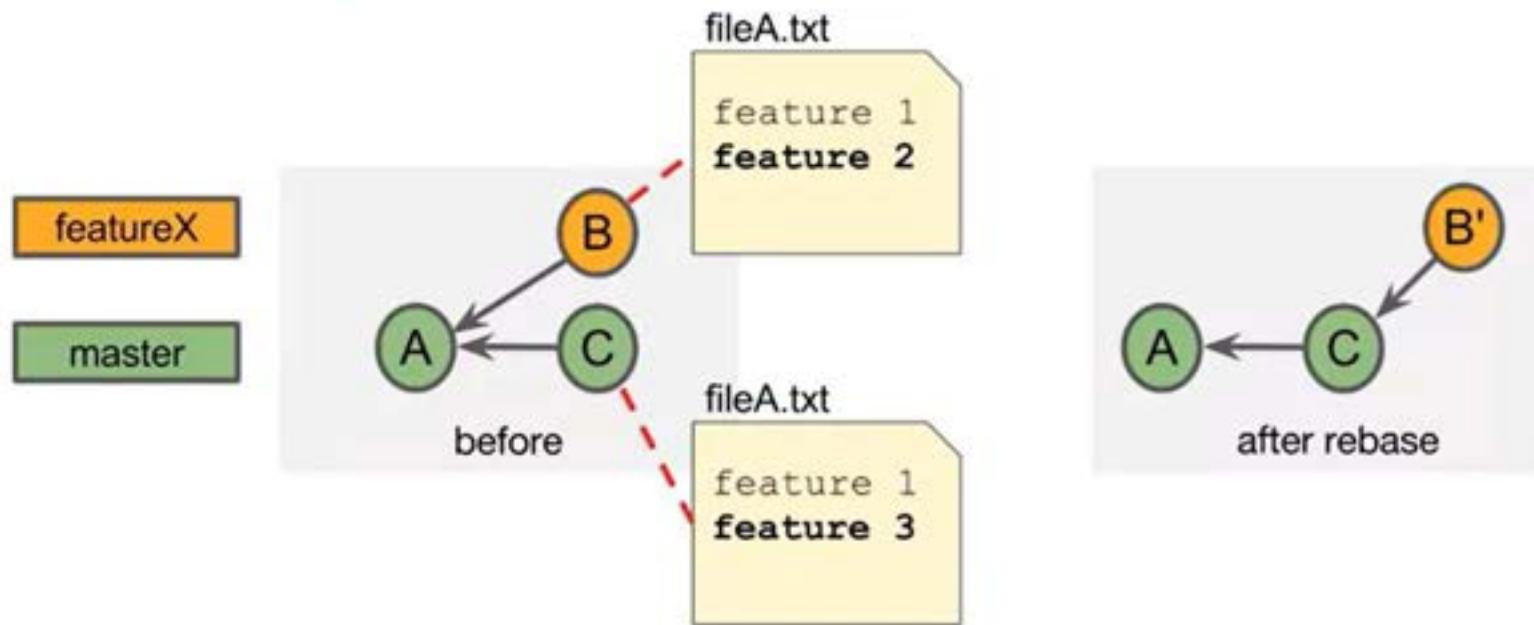
When rebasing, Git applies the diffs to the new parent commit

- This is called "reapplying commits"



REBASING IS A MERGE

- Reapplying commits is a form of merge and is susceptible to merge conflicts
- For example, commits B and C can change the same file, causing a merge conflict during the rebase



REBASING PROS AND CONS

- Pros:

- You can incorporate changes from the parent branch
 - You can use the new features/bugfixes
 - Tests are on more current code
 - It makes the eventual merge into master fast-forwardable
- Avoids "unnecessary" commits
 - It allows you to shape/define clean commit histories

- Cons:

- Merge conflicts may need to be resolved
- It can cause problems if your commits have been shared
- You are not preserving the commit history

git rebase SYNTAX

git rebase <upstream>

- Changes the parent of the currently checked out branch to <upstream>

git rebase <upstream> <branch>

- Checks out <branch> and changes its parent <upstream>
- This is a convenience to avoid issuing two commands

```
$ git checkout featureX  
$ git rebase master  
  
# equivalent to:  
$ git rebase master featureX
```

Upstream usually refers to the parent branch of the rebased branch

FIXING A MERGE CONFLICT WHILE REBASING

1. `git checkout featureX`
2. `git rebase master`
 - a. CONFLICT
3. `git status`
 - a. Both modified fileA.txt
4. Fix fileA.txt
5. `git add fileA.txt`
6. `git rebase --continue`

Files with conflicts are **modified by Git** in the working tree

- Run `git status` to see which files have been modified

REBASE WITH A MERGE CONFLICT (1 of 4)

Since rebase involves a merge, there is the possibility of a merge conflict

```
$ git log --all --graph --oneline
* ee87ce6 (master) added feature 3
| * 8c4fdca (HEAD -> feature) feature 2 wip
|/
* ea6e32e added feature 1
$ git rebase master
First, rewinding head to replay your work on top of it...
(snip)
CONFLICT (content): Merge conflict in fileA.txt
(snip)
When you have resolved this problem, run "git rebase --continue".
If you prefer to skip this patch, run "git rebase --skip" instead.
To check out the original branch and stop rebasing, run "git rebase
--abort".
```

REBASE WITH A MERGE CONFLICT (2 of 4)

```
$ git status
rebase in progress; onto ee87ce6
You are currently rebasing branch 'feature' on 'ee87ce6'.
  (fix conflicts and then run "git rebase --continue")
  (use "git rebase --skip" to skip this patch)
  (use "git rebase --abort" to check out the original branch)

Unmerged paths:
  (use "git reset HEAD <file>..." to unstage)
  (use "git add <file>..." to mark resolution)

    both modified:  fileA.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

REBASE WITH A MERGE CONFLICT (4 of 4)

Complete the rebase

```
$ git add fileA.txt  
$ git rebase --continue  
Applying: feature 2 wip
```

REBASE WITH A MERGE CONFLICT (4 of 4)

Complete the rebase

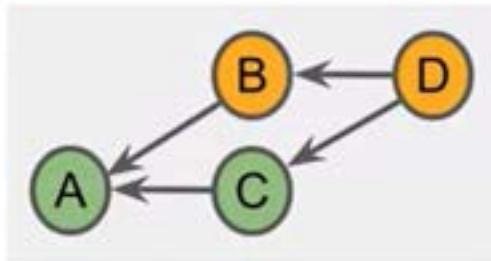
```
$ git add fileA.txt
$ git rebase --continue
Applying: feature 2 wip
$ git status
On branch feature
nothing to commit, working tree clean
$ git log --all --graph --oneline
* eac1bd4 (HEAD -> feature) feature 2 wip
* ee87ce6 (master) added feature 3
* ea6e32e added feature 1
```

ABORTING A REBASE

Use `git rebase --abort` to get back to the pre-rebase state

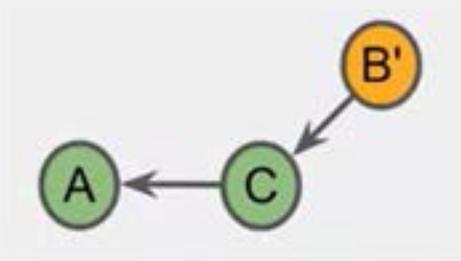
```
$ git checkout feature
Switched to branch 'feature'
$ git rebase master
First, rewinding head to replay your work on top of it...
(snip)
CONFLICT (content): Merge conflict in fileA.txt
(snip)
When you have resolved this problem, run "git rebase --continue".
If you prefer to skip this patch, run "git rebase --skip" instead.
To check out the original branch and stop rebasing, run "git rebase
--abort".
$ git rebase --abort
$ git status
On branch feature
nothing to commit, working tree clean
```

RESOLVING MERGE CONFLICTS- COMPARING MERGE TO REBASE



1. git checkout **master**
2. git **merge** featureX
 - a. CONFLICT
3. git status
 - a. Both modified fileA.txt
4. Fix fileA.txt
5. git add fileA.txt
6. **git commit**

merge



1. git checkout **featureX**
2. git **rebase** master
 - a. CONFLICT
3. git status
 - a. Both modified fileA.txt
4. Fix fileA.txt
5. git add fileA.txt
6. **git rebase --continue**

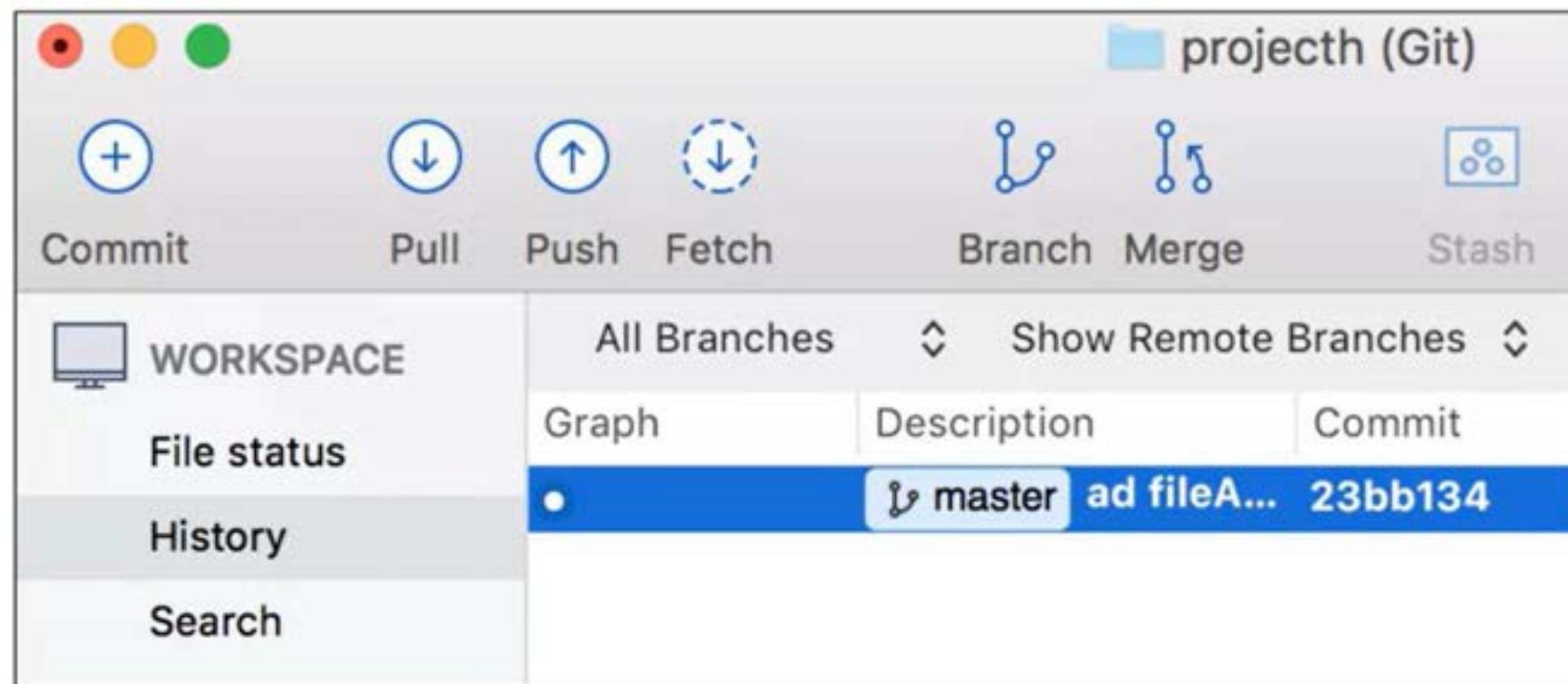
rebase

Topics

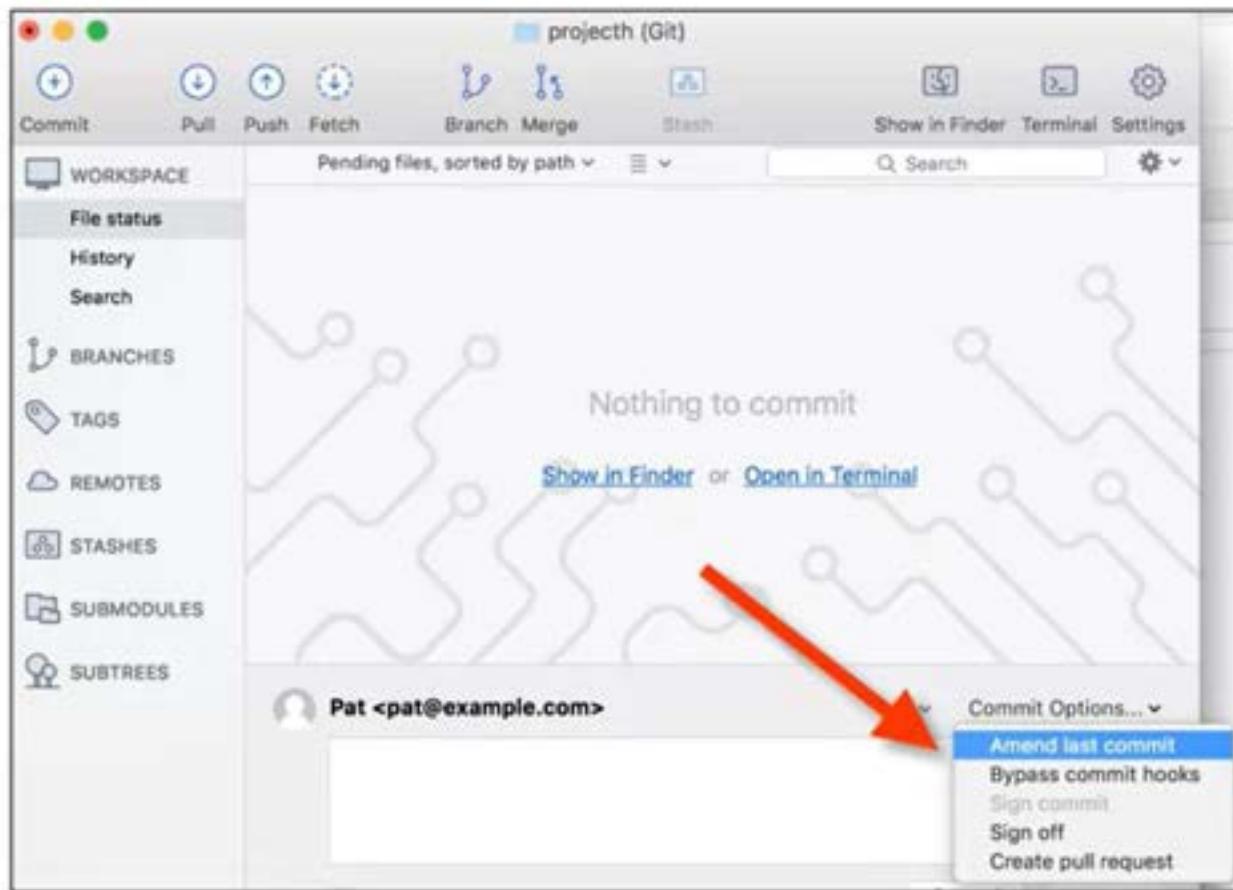
Amending a commit

Interactive rebase

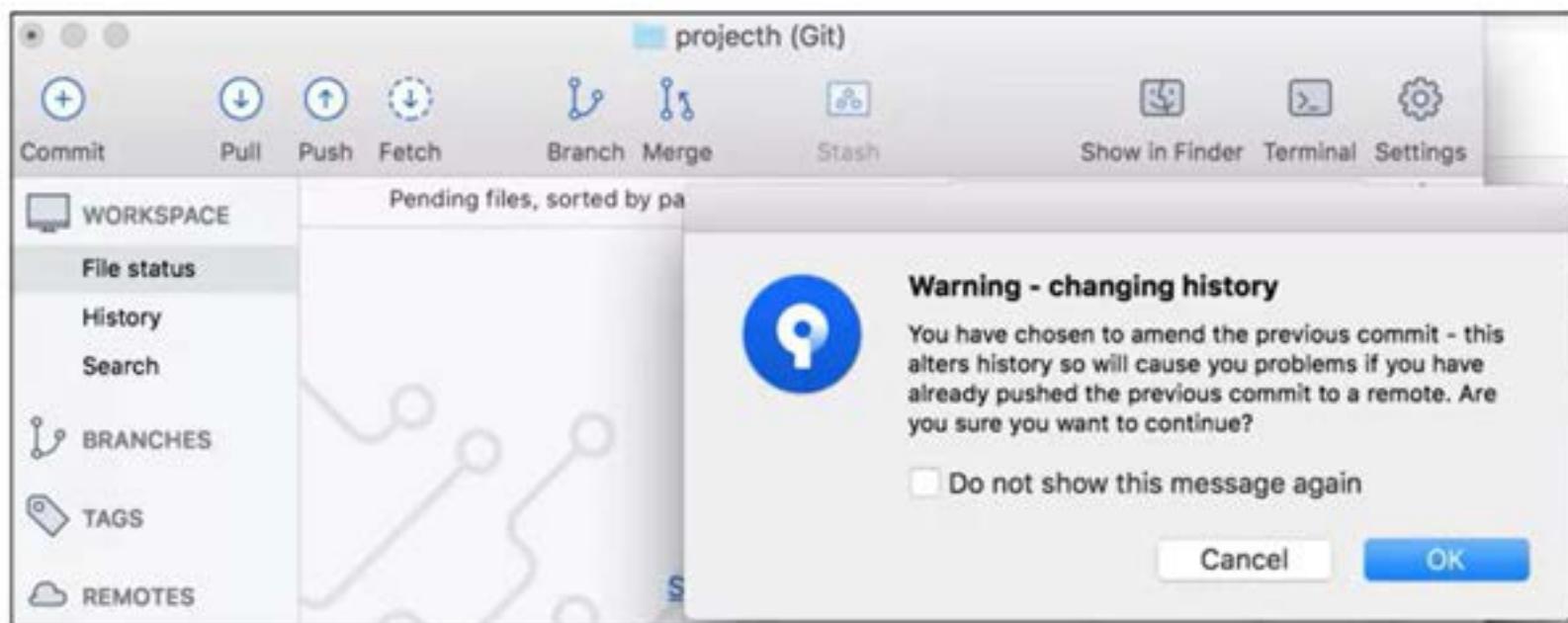
AMENDING A COMMIT



AMENDING A COMMIT

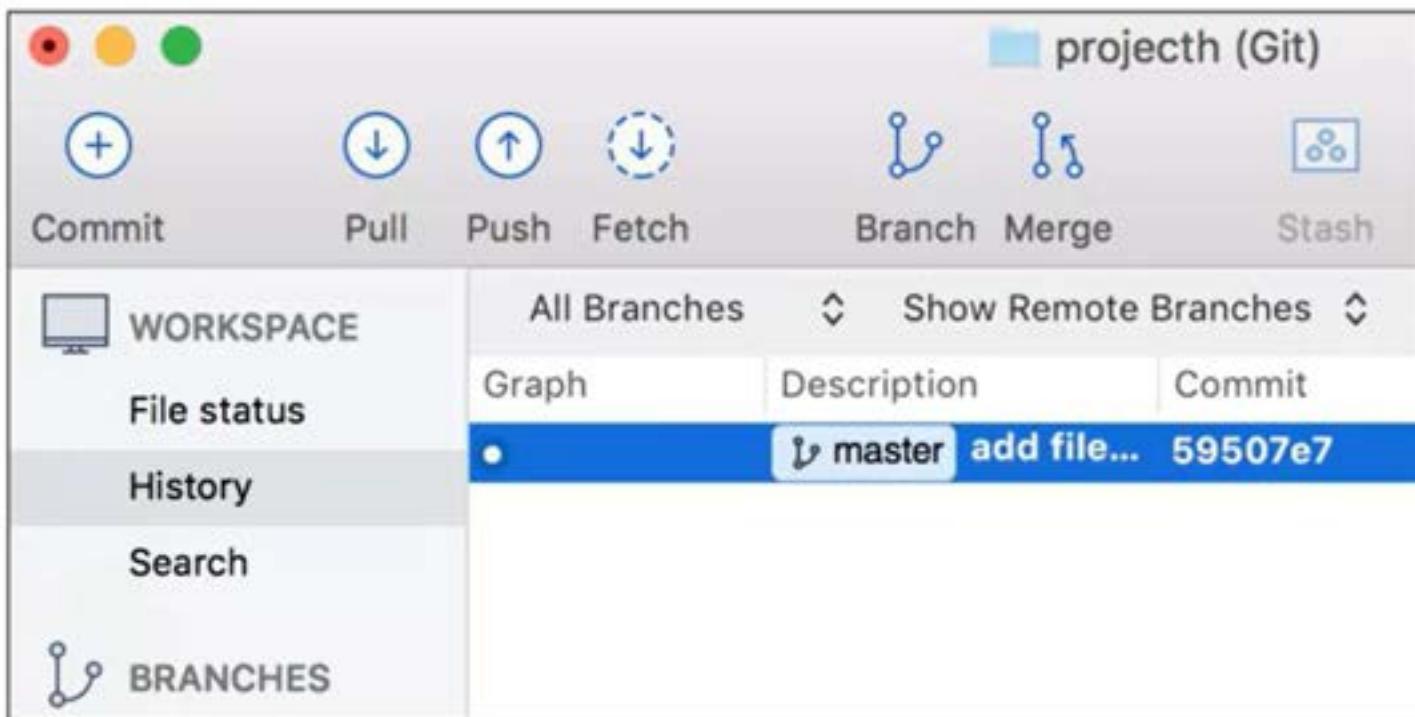


AMENDING A COMMIT

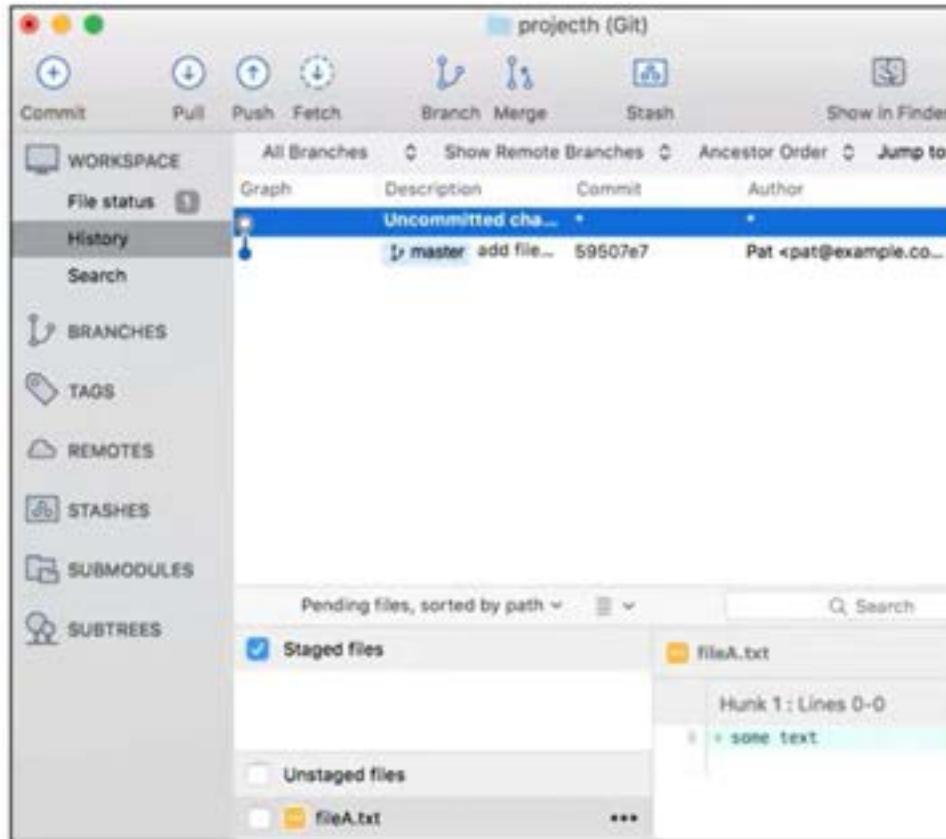


AMENDING A COMMIT

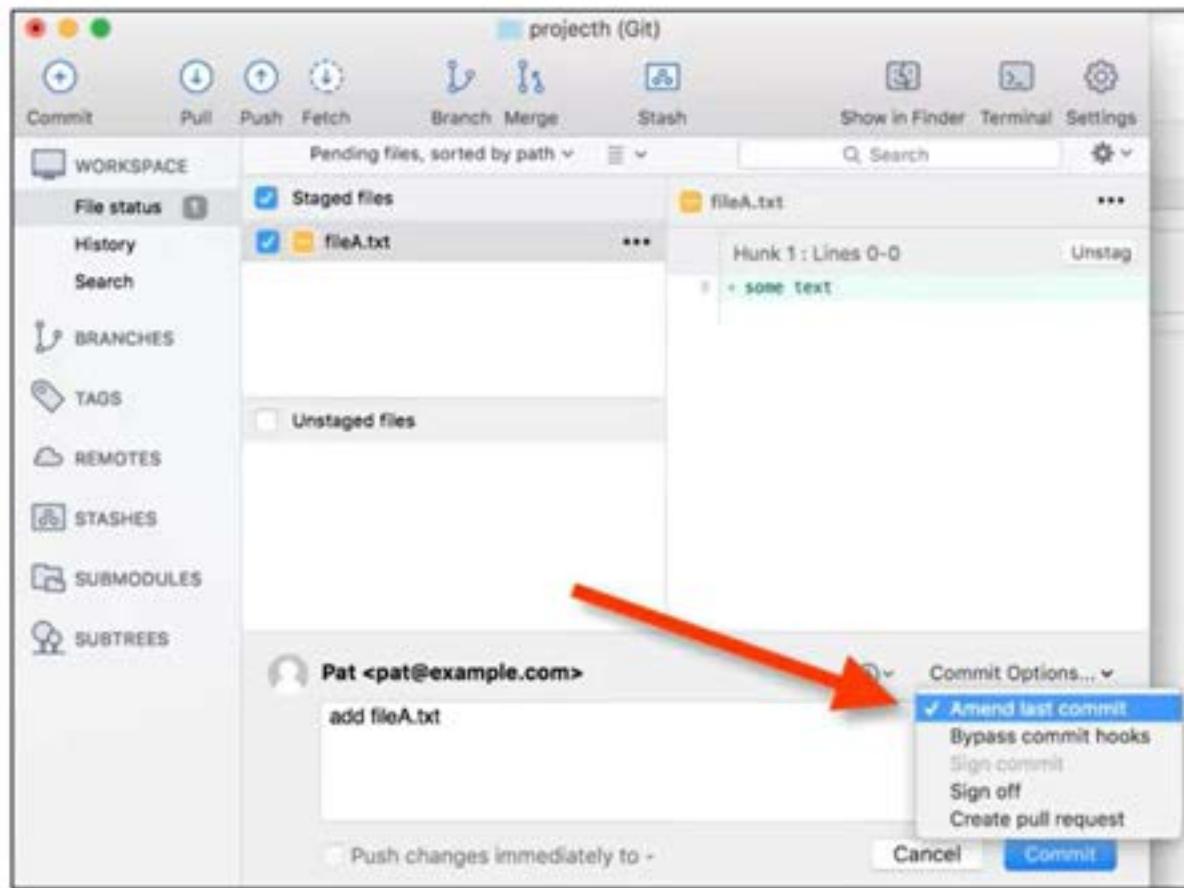
The commit is amended and has a new SHA-1



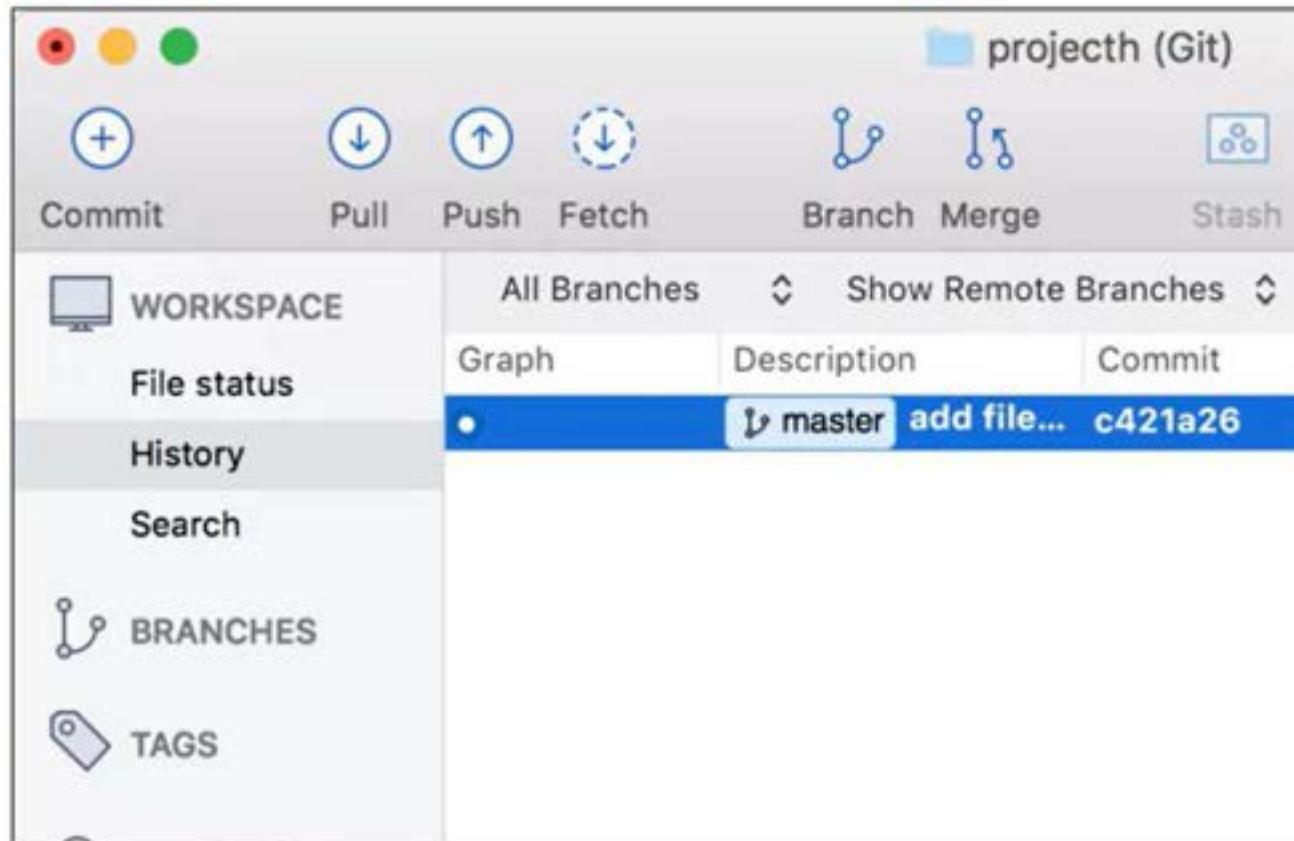
AMENDING COMMITTED FILES



AMENDING COMMITTED FILES



AMENDING COMMITTED FILES



Topics

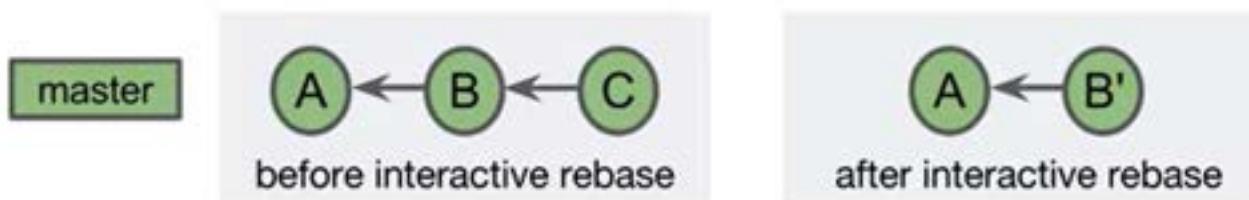
Amending a commit

Interactive rebase

INTERACTIVE REBASE

Interactive rebase lets you edit multiple commits

- The commits can belong to any branch
- The commit history is changed- do not use for shared commits
- Commits after the selected commit can be modified



INTERACTIVE REBASE OPTIONS

- Use the commit as is
- Edit the commit message
- Stop and amend/edit the commit
- Drop/delete the commit
- Squash
- Reorder commits

EDIT A COMMIT MESSAGE

before rebase:

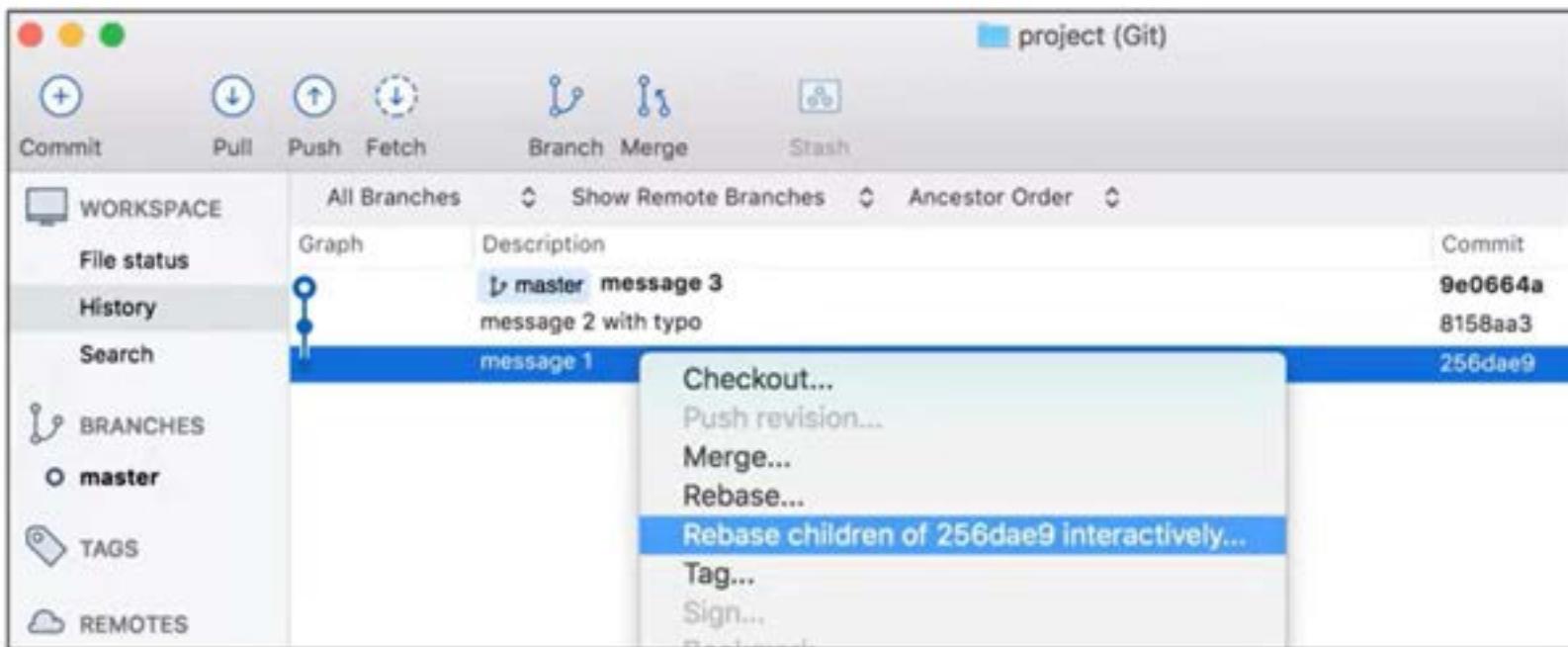
commit	commit message
A	message 1
B	message 2 with typo
C	message 3

after rebase:

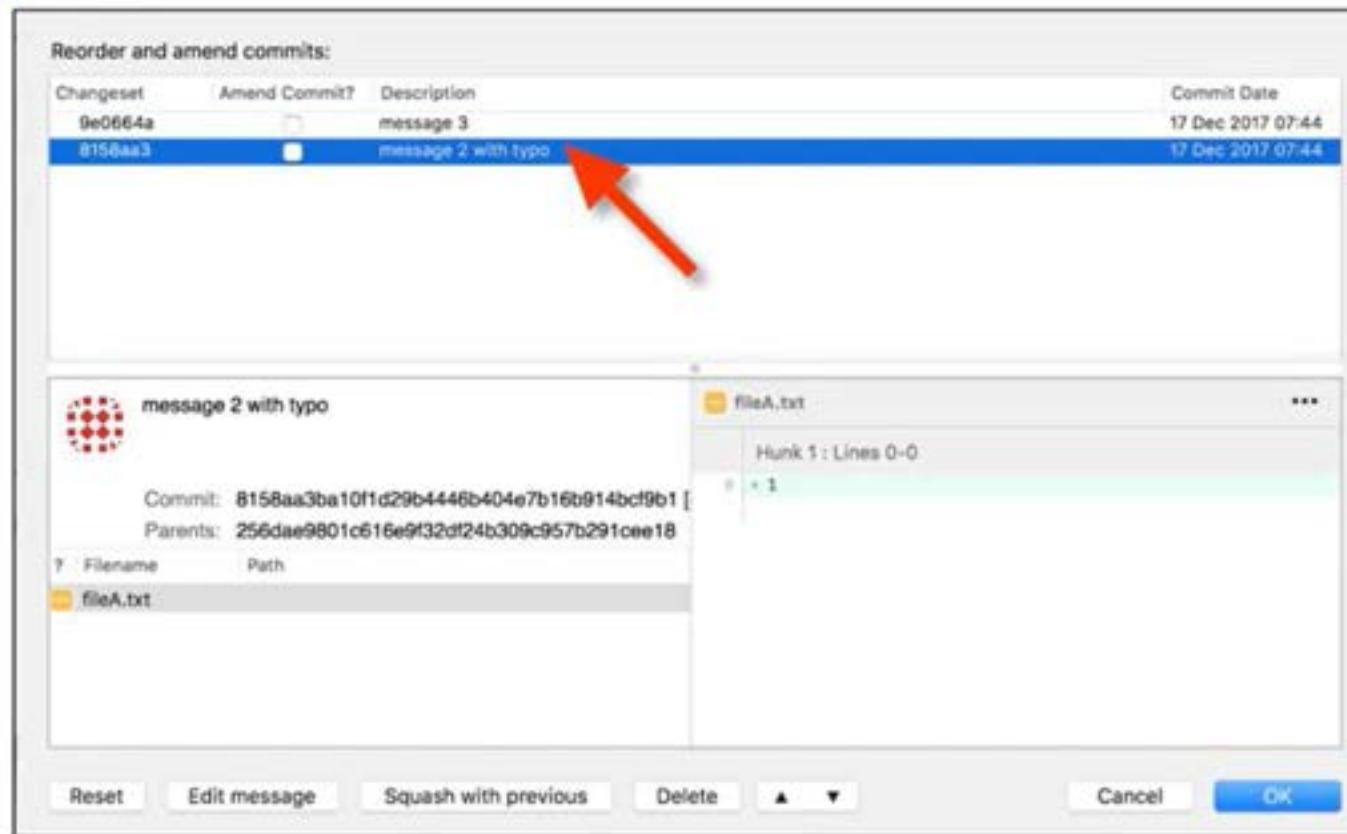
commit	commit message
A	message 1
B'	message 2
C'	message 3

EDIT A COMMIT MESSAGE

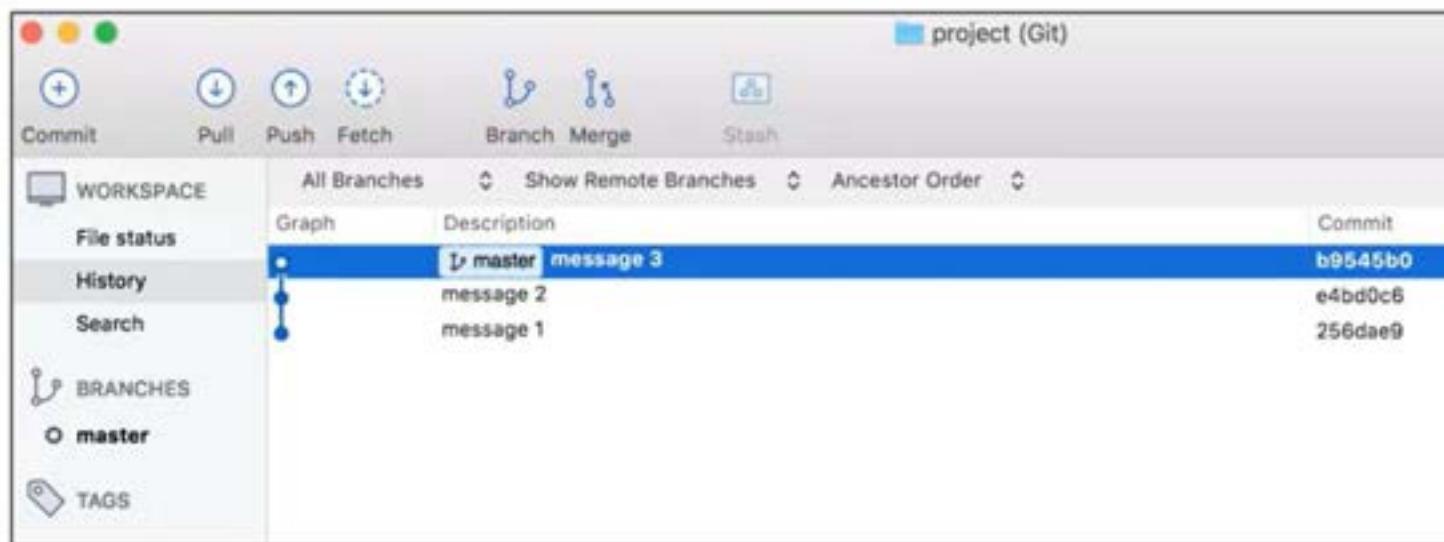
Right-click on the *initial* commit to modify children



EDIT A COMMIT MESSAGE



EDIT A COMMIT MESSAGE



EDIT A COMMIT

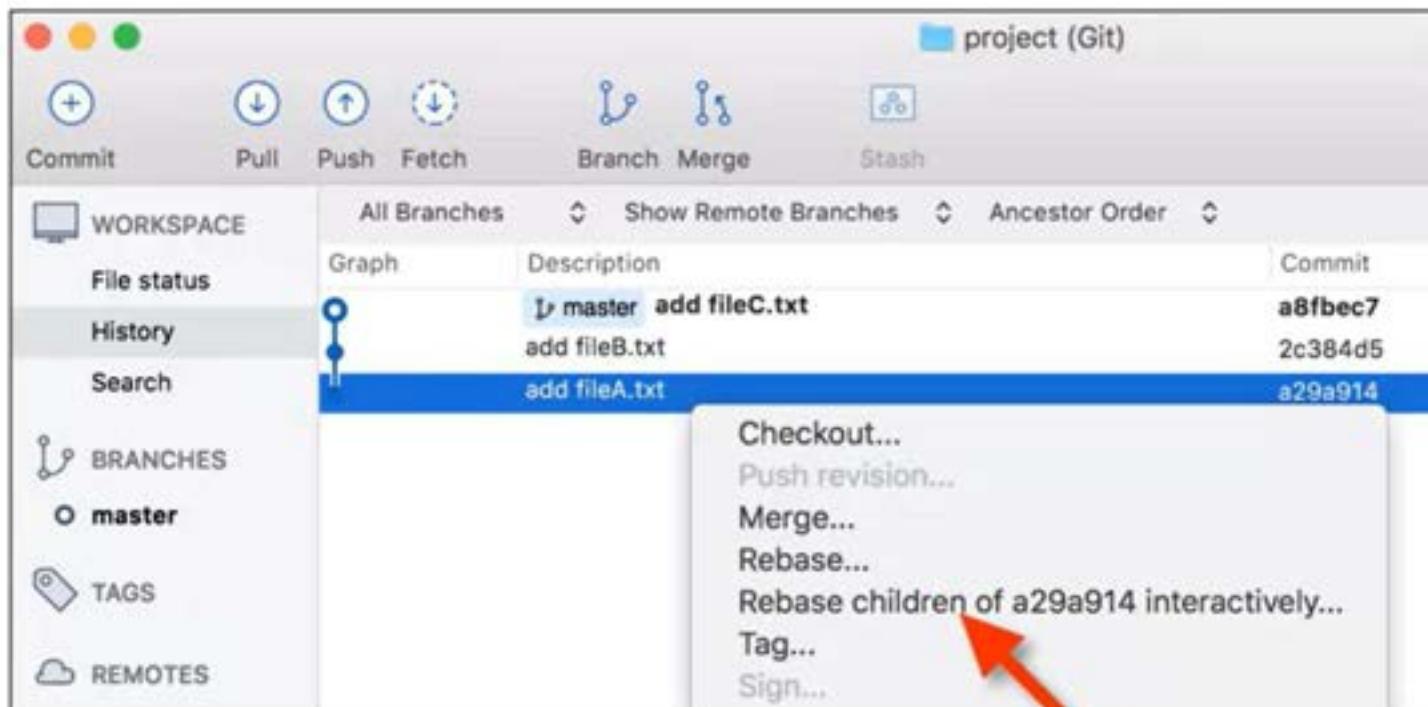
before rebase:

commit	file(s)	commit message
A	fileA.txt	add fileA.txt
B	fileA.txt, fileBB.txt	add fileB.txt
C	fileA.txt, fileBB.txt , fileC.txt	add fileC.txt

after rebase:

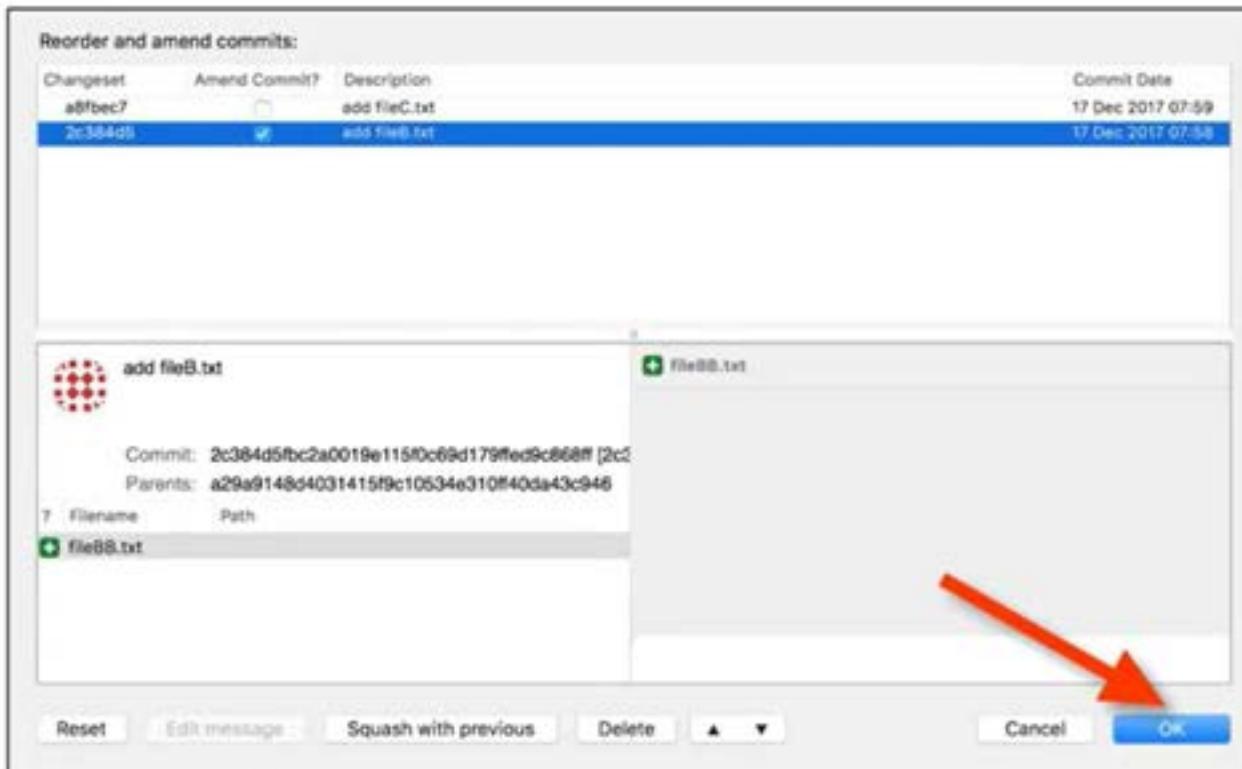
commit	file(s)	commit message
A	fileA.txt	add fileA.txt
B'	fileA.txt, fileB.txt	add fileB.txt
C'	fileA.txt, fileB.txt , fileC.txt	add fileC.txt

EDIT A COMMIT



EDIT A COMMIT

Check the **Amend Commit?** checkbox



EDIT A COMMIT

The amended commit is checked out

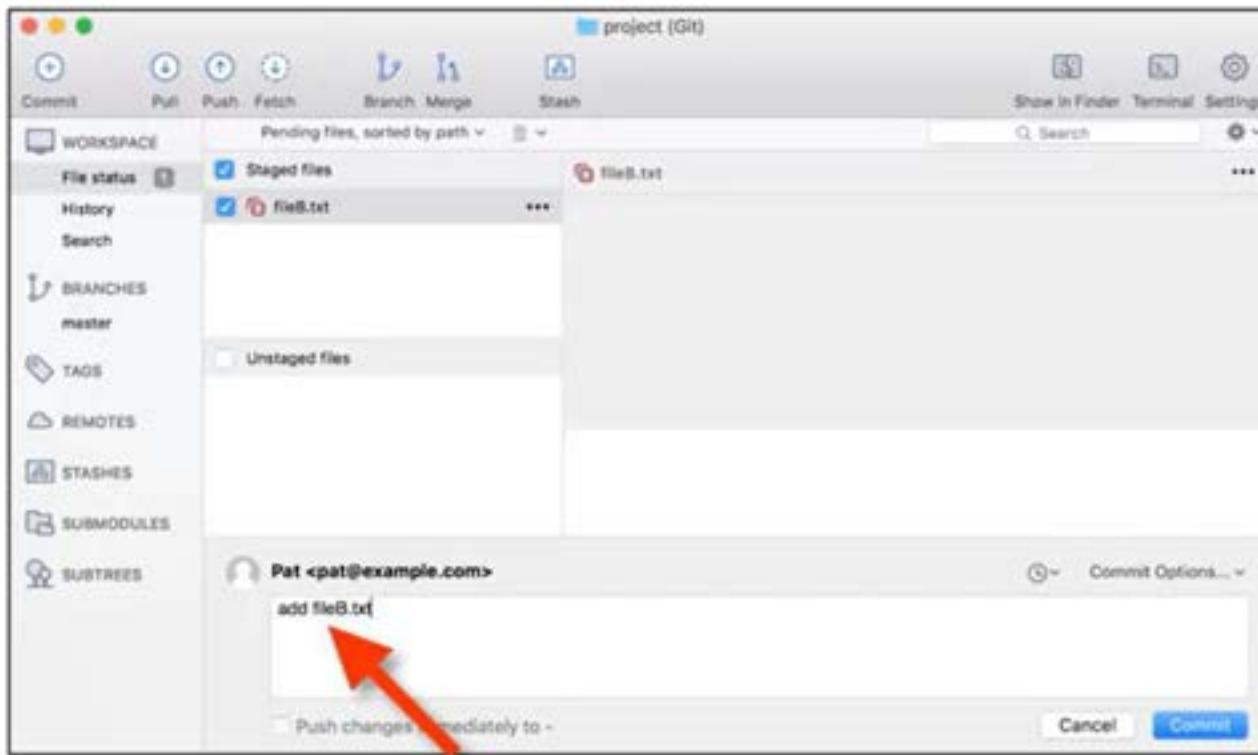
A screenshot of a Git interface showing the commit history for the master branch. The interface includes standard Git operations like Commit, Pull, Push, Fetch, Branch, Merge, and Stash. The commit history table shows three commits:

File status	Description	Commit
Graph	master add fileC.txt	a8fbec7
HEAD	add fileB.txt	2c384d5
	add fileA.txt	a29a914

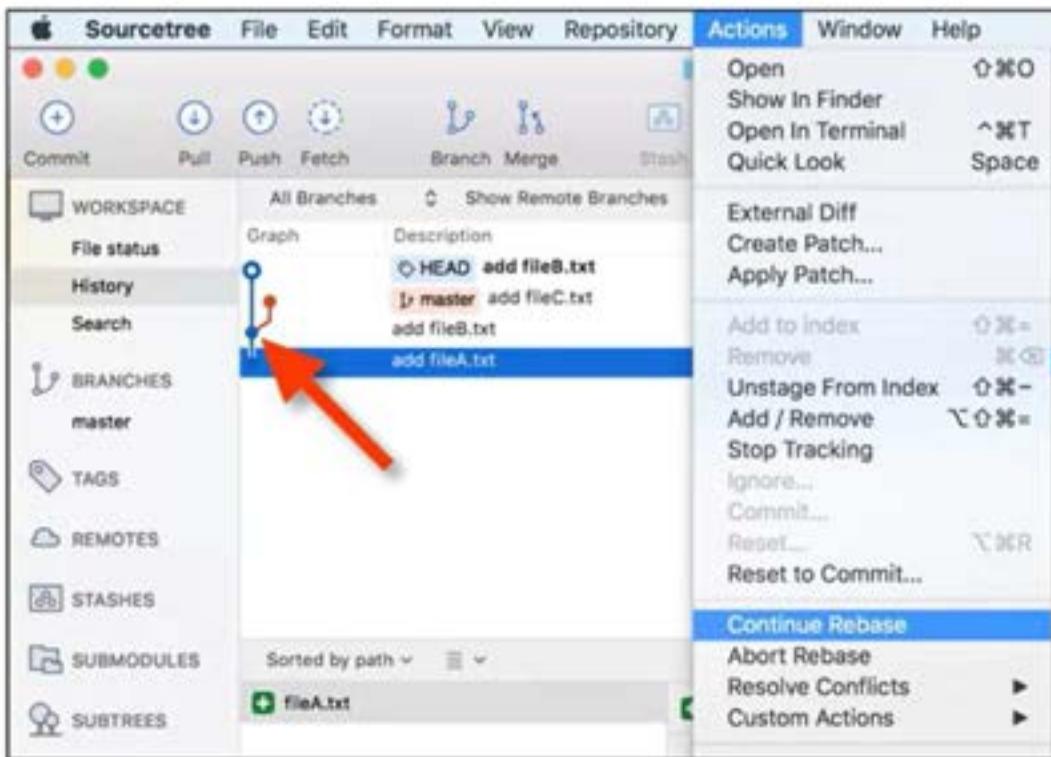
A red arrow points to the HEAD commit, indicating it is the current working copy.

EDIT A COMMIT

Click Commit



EDIT A COMMIT



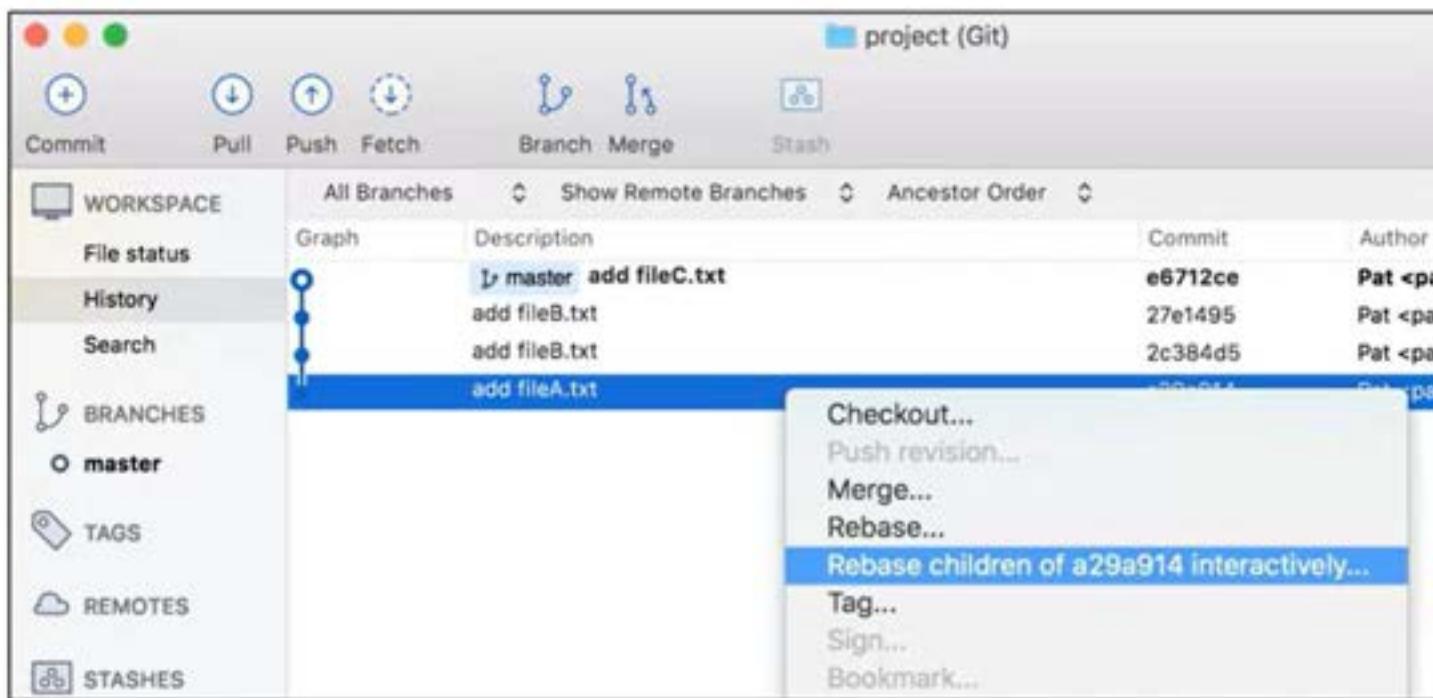
EDIT A COMMIT

The screenshot shows a Git commit history interface with the following details:

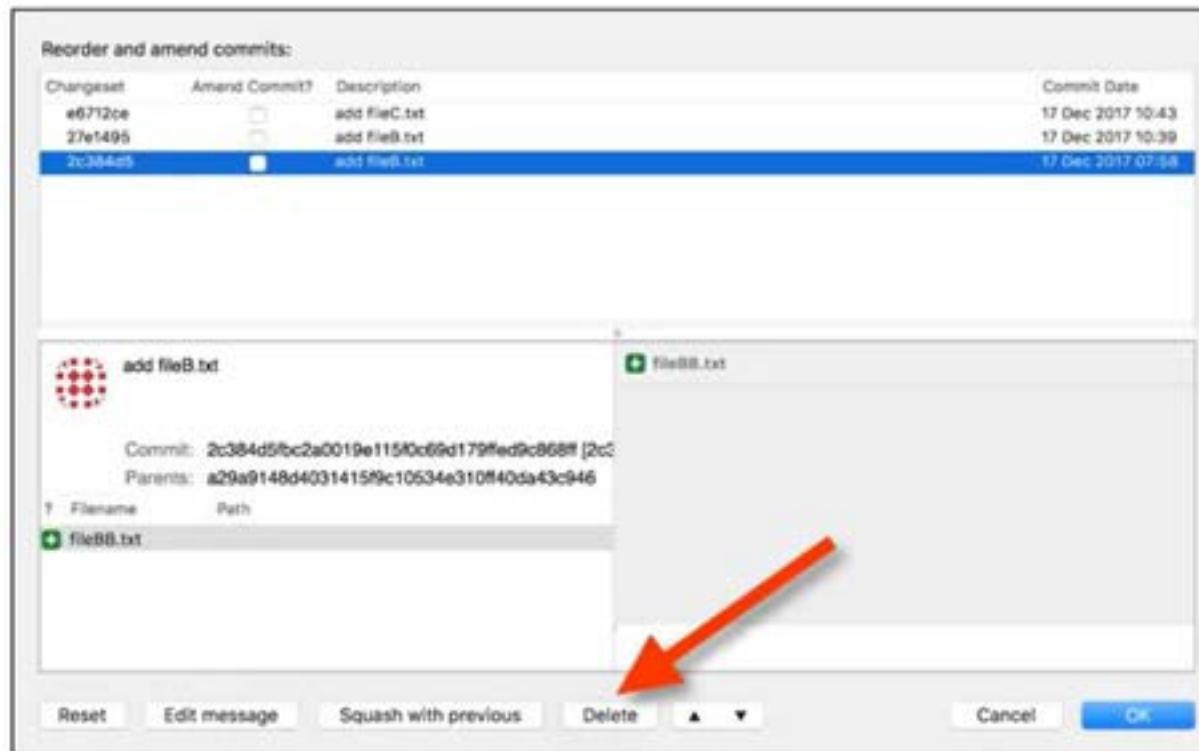
Description	Commit
l master add fileC.txt	e6712ce
add fileB.txt	27e1495
add fileB.txt	2c384d5
add fileA.txt	a29a914

The commit history shows four commits on the master branch. The first commit adds fileC.txt, the second adds fileB.txt, the third adds fileB.txt again, and the fourth adds fileA.txt.

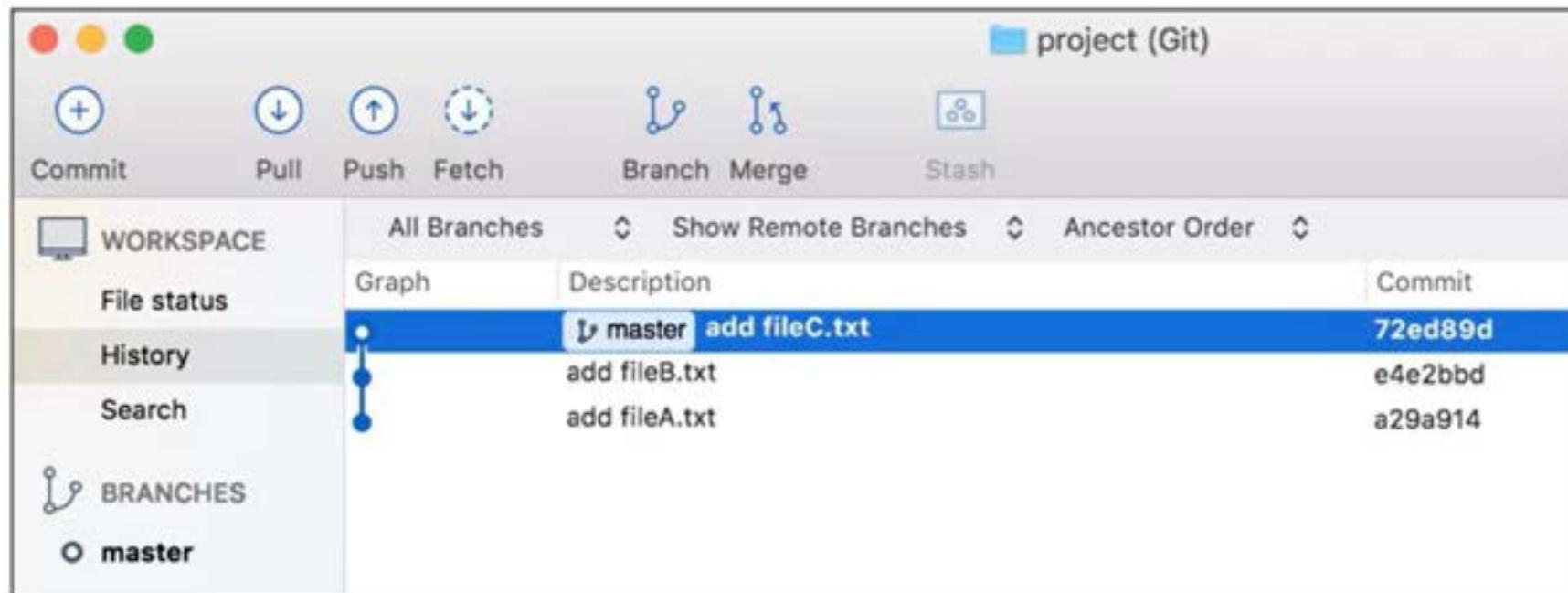
DELETE A COMMIT



DELETE A COMMIT



DELETE A COMMIT



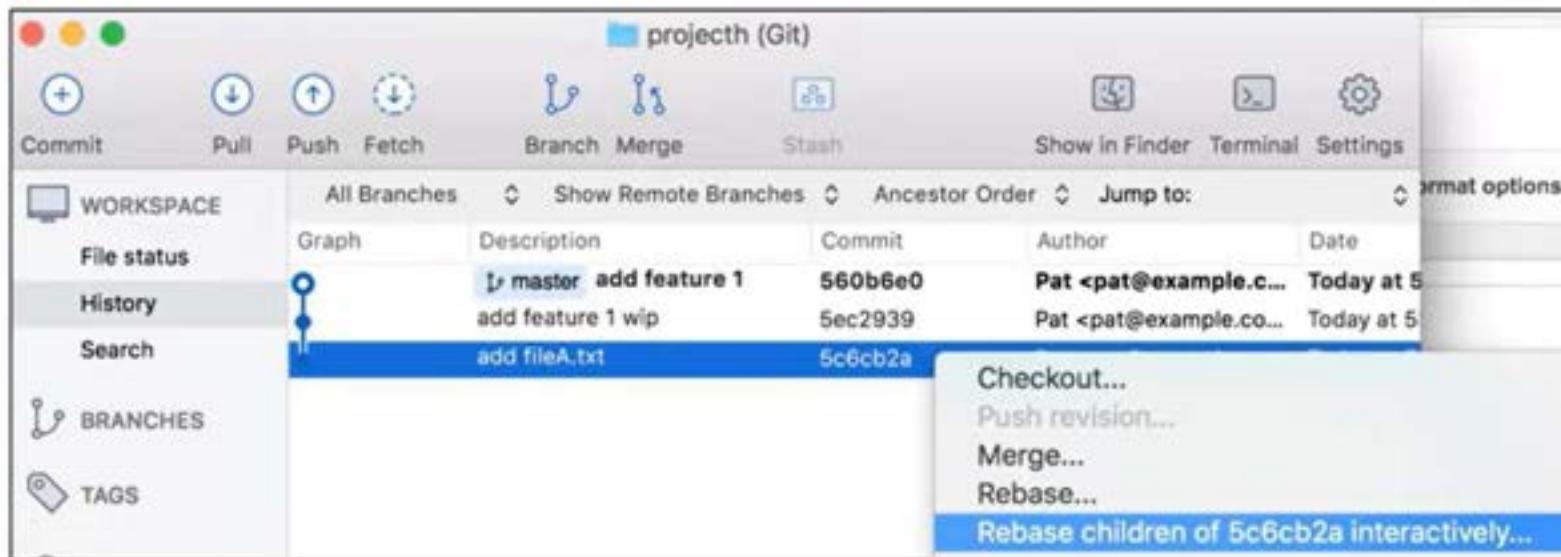
The screenshot shows a Git commit history interface with the following details:

Commit	Description	Commit
72ed89d	master add fileC.txt	72ed89d
e4e2bbd	add fileB.txt	e4e2bbd
a29a914	add fileA.txt	a29a914

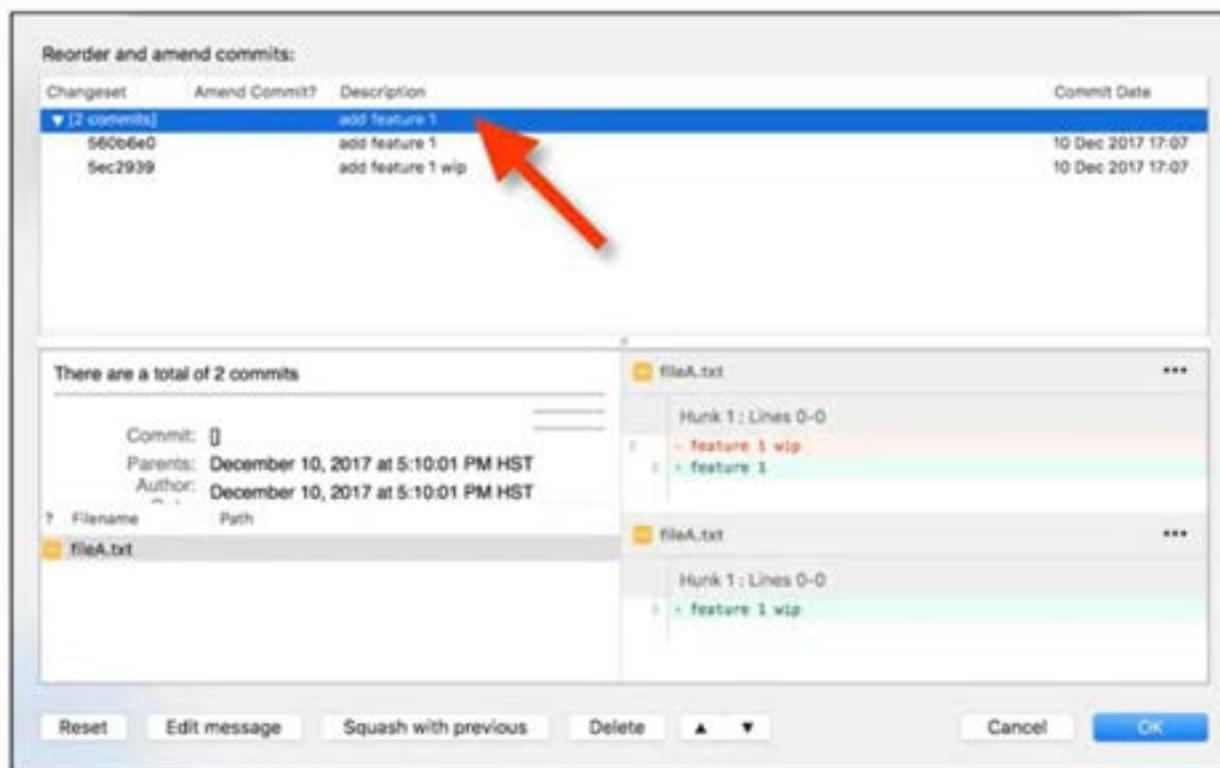
The commit with hash 72ed89d is currently selected. The interface includes a sidebar with options like WORKSPACE, File status, History, and Search, and a top bar with various Git operations like Commit, Pull, Push, Fetch, Branch, Merge, and Stash.

SQUASH A COMMIT

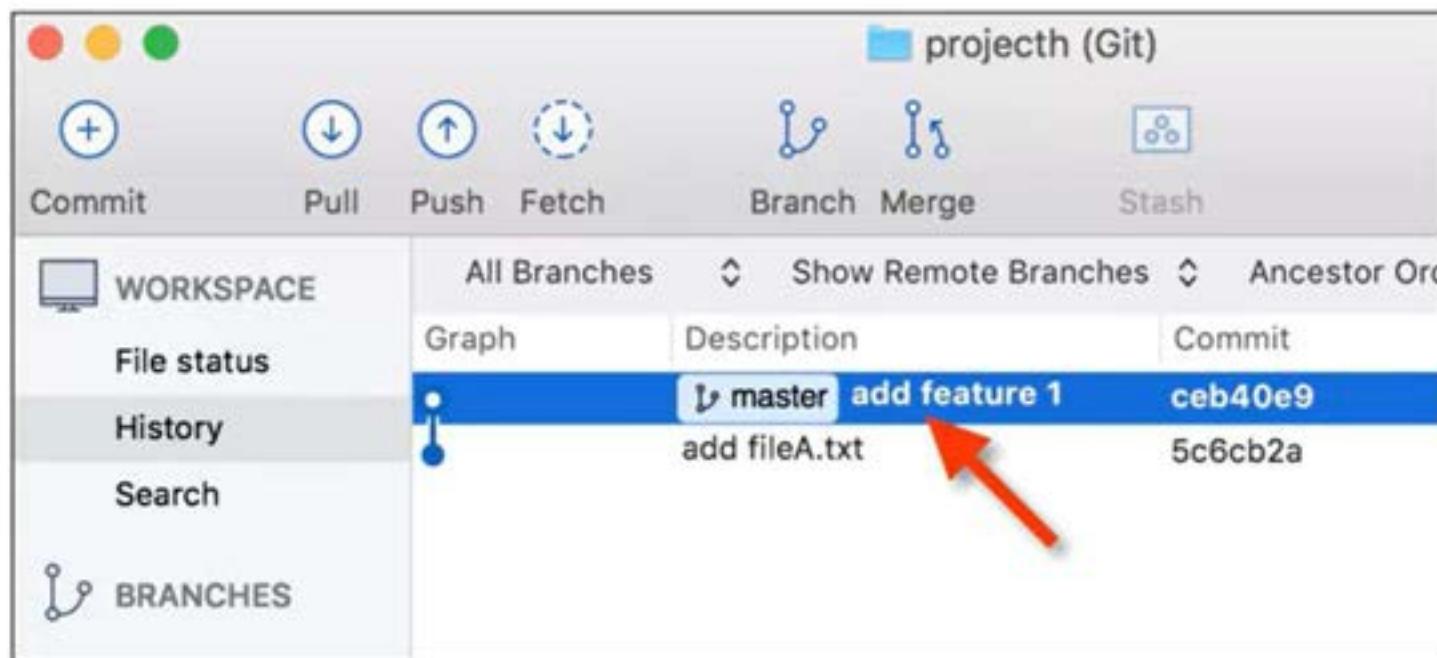
1. Applies a newer (squashed) commit to an older commit
2. Combines the commit messages
3. Removes the newer commit



SQUASH A COMMIT



SQUASH A COMMIT



SQUASH VS. DELETE

Squash- Combine this commit with the older commit, creating a single commit

- The work of both commits is included

Delete- No changes from this commit are applied

- The diff is thrown out
- The work of this commit is lost
- Greater chance of a merge conflict

EXAMPLE: SQUASH VS. DELETE

before rebase:

commit	file(s)
A	fileA.txt
B	fileA.txt, fileB.txt
C	fileA.txt, fileB.txt, fileC.txt

*after interactive rebase with **squash commit C:***

commit	file(s)
A	fileA.txt
B'	fileA.txt, fileB.txt, fileC.txt

*after interactive rebase with **delete commit B:***

commit	file(s)
A	fileA.txt
C'	fileA.txt, fileC.txt

REVIEW

- You can amend the most recent commit's message and/or committed files
 - It creates a new SHA-1
- Interactive rebase allows you to rewrite the history of a branch
- A squash reduces multiple commits into a single commit

Topics

Amending a commit

Interactive rebase

Squash merges

AMENDING A COMMIT

- You can change the most recent commit:
 - change the commit message
 - change the project files
- This creates a new SHA-1 (rewrites history)

```
$ touch fileC.txt
$ git add fileC.txt
$ git commit -m "ad fileC.txt"
[master 43f30b5] ad fileC.txt
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 fileC.txt
$ git log --oneline -1
43f30b5 (HEAD -> master) ad fileC.txt
$ git commit --amend -m "add fileC.txt"
[master d70eb1f] add fileC.txt
Date: ...
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 fileC.txt
$ git log --oneline -1
d70eb1f (HEAD -> master) add fileC.txt
```

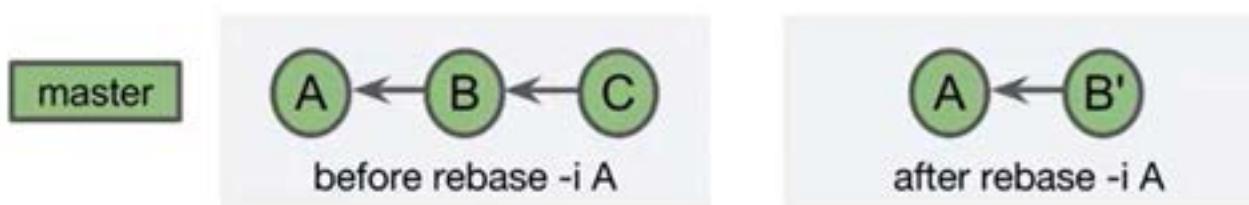
AMENDING A COMMIT- CHANGING COMMITTED FILES

- You can modify the staging area and amend a commit
- Optionally use the --no-edit option to reuse the previous commit message

```
$ git log --oneline -1  
d70eb1f (HEAD -> master) add fileC.txt  
$ echo "some text" > fileC.txt  
$ git add fileC.txt  
$ git commit --amend --no-edit  
[master 9cd5d96] add fileC.txt  
Date: ...  
1 file changed, 1 insertion(+)  
create mode 100644 fileC.txt  
$ git log --oneline -1  
9cd5d96 (HEAD -> master) add fileC.txt
```

INTERACTIVE REBASE

- Interactive rebase lets you edit commits using commands
 - The commits can belong to any branch
 - The commit history is changed- do not use for shared commits
- `git rebase -i <after-this-commit>`
 - Commits in the current branch after `<after-this-commit>` are listed in an editor and can be modified



INTERACTIVE REBASE OPTIONS

- Use the commit as is
- Edit the commit message
- Stop and edit the commit
- Drop/delete the commit
- Squash
- Fixup
- Reorder commits
- Execute shell commands

INTERACTIVE COMMANDS

```
pick cf0ffa7 add fileB.txt
pick 2bd5ca3 add fileC.txt
#
# rebase 0e917b6..2bd5ca3 onto 0e917b6 (2 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
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

EDIT A COMMIT

before rebase:

commit	file(s)	commit message
A	fileA.txt	add fileA.txt
B	fileA.txt, fileBB.txt	add fileB.txt with typo
C	fileA.txt, fileBB.txt , fileC.txt	add fileC.txt

after rebase:

commit	file(s)	commit message
A	fileA.txt	add fileA.txt
B'	fileA.txt, fileB.txt	add fileB.txt
C'	fileA.txt, fileB.txt , fileC.txt	add fileC.txt

EDIT A COMMIT

```
$ git log --oneline --graph
* 2bd5ca3 (HEAD -> master) add fileC.txt
* cf0ffa7 add fileB.txt with typo
* 0e917b6 add fileA.txt
$ git rebase -i 0e91
```

EDIT A COMMIT

```
edit cf0ffa7 add fileB.txt with typo
pick 2bd5ca3 add fileC.txt

# Rebase 0e917b6..2bd5ca3 onto 0e917b6 (2 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
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

EDIT A COMMIT

```
$ git rebase -i 0e91
(change commit cfof to edit in editor)
Stopped at cf0ffa7... add fileB.txt with typo
You can amend the commit now, with
```

```
git commit --amend
```

Once you are satisfied with your changes, run

```
git rebase --continue
$ ls
fileA.txt    fileBB.txt
```

EDIT A COMMIT

```
$ mv fileBB.txt fileB.txt
$ git status
interactive rebase in progress; onto 0e917b6
Last command done (1 command done):
  edit cf0ffa7 add fileB.txt with typo
Next command to do (1 remaining command):
  pick 2bd5ca3 add fileC.txt
  (use "git rebase --edit-todo" to view and edit)
You are currently editing a commit while rebasing branch 'master' on '0e917b6'.
  (use "git commit --amend" to amend the current commit)
  (use "git rebase --continue" once you are satisfied with your changes)

Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    deleted:    fileBB.txt

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    fileB.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

EDIT A COMMIT

```
$ git add .
$ git commit --amend -m "add fileB.txt"
[detached HEAD 2a91c04] add fileB.txt
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 fileB.txt
$ git rebase --continue
Successfully rebased and updated refs/heads/master.
$ git log --oneline --graph
* 815a033 (HEAD -> master) add fileC.txt
* 2a91c04 add fileB.txt
* 0e917b6 add fileA.txt
$ ls
fileA.txt    fileB.txt    fileC.txt
```

DELETE A COMMIT

The commit's work is not used

```
$ git log --oneline --graph
* 815a033 (HEAD -> master) add fileC.txt
* 2a91c04 add fileB.txt
* 0e917b6 add fileA.txt
$ ls
fileA.txt    fileB.txt    fileC.txt
$ git rebase -i e091
```

DELETE A COMMIT

```
drop 2a91c04 add fileB.txt
pick 815a033 add fileC.txt

# Rebase 0e917b6..815a033 onto 0e917b6 (2 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
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

DELETE A COMMIT

```
drop 2a91c04 add fileB.txt
pick 815a033 add fileC.txt

# Rebase 0e917b6..815a033 onto 0e917b6 (2 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
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

DELETE A COMMIT

```
$ git rebase -i 0e91
(change commit 2a91 to drop in editor)
Successfully rebased and updated refs/heads/master.
$ git log --oneline --graph
* 11c9e2b (HEAD -> master) add fileC.txt
* 0e917b6 add fileA.txt
$ ls
fileA.txt    fileC.txt
```

SQUASH A COMMIT

```
pick 8b450d9 add fileB.txt
squash 045de6d add fileC.txt

# Rebase b7fa489..045de6d onto b7fa489 (2 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
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

SQUASH A COMMIT

1. Applies a newer (squashed) commit to an older commit
2. Combines the commit messages
3. Removes the newer commit

```
$ git log --oneline --graph
* 045de6d (HEAD -> master) add fileC.txt
* 8b450d9 add fileB.txt
* b7fa489 add fileA.txt
$ ls
fileA.txt    fileB.txt    fileC.txt
$ git rebase -i b7fa
```

Note: A fixup is like a squash, but the squashed commit's message is discarded

SQUASH A COMMIT

```
# This is a combination of 2 commits.
# This is the 1st commit message:

add fileB.txt

# This is the commit message #2:

add fileC.txt

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# Date:      Sun Dec 17 17:16:28 2017 -0800
#
# interactive rebase in progress; onto b7fa489
# Last commands done (2 commands done):
#   pick 8b450d9 add fileB.txt
#   squash 045de6d add fileC.txt
# No commands remaining.
# You are currently rebasing branch 'master' on 'b7fa489'.
#
# Changes to be committed:
#       new file:  fileB.txt
#       new file:  fileC.txt
```

SQUASH A COMMIT

```
$ git rebase -i b7fa
(edit command file and commit message)
[detached HEAD c781502] add fileB.txt and fileC.txt
 2 files changed, 0 insertions(+), 0 deletions(-)
  create mode 100644 fileB.txt
  create mode 100644 fileC.txt
Successfully rebased and updated refs/heads/master.
$ git log --oneline --graph
* c781502 (HEAD -> master) add fileB.txt and fileC.txt
* b7fa489 add fileA.txt
$ ls
fileA.txt  fileB.txt  fileC.txt
```

SQUASH VS. DELETE

Squash- Combine this commit with the older commit, creating a single commit

- The work of both commits is included

Delete- No changes from this commit are applied

- The diff is thrown out
- The work of this commit is lost
- Greater chance of a merge conflict

EXAMPLE: SQUASH VS. DELETE

before rebase:

commit	file(s)
A	fileA.txt
B	fileA.txt, fileB.txt
C	fileA.txt, fileB.txt, fileC.txt

*after interactive rebase with **squash commit C:***

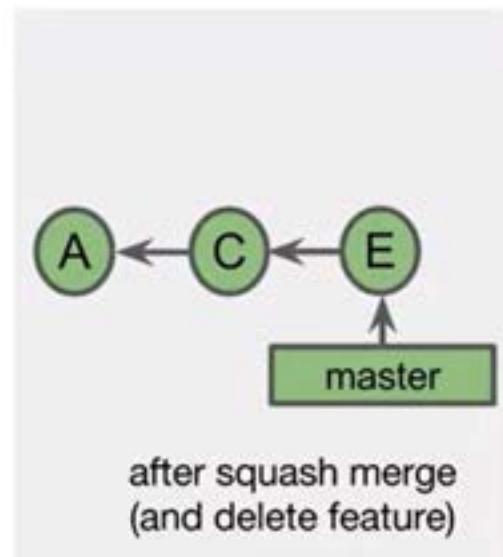
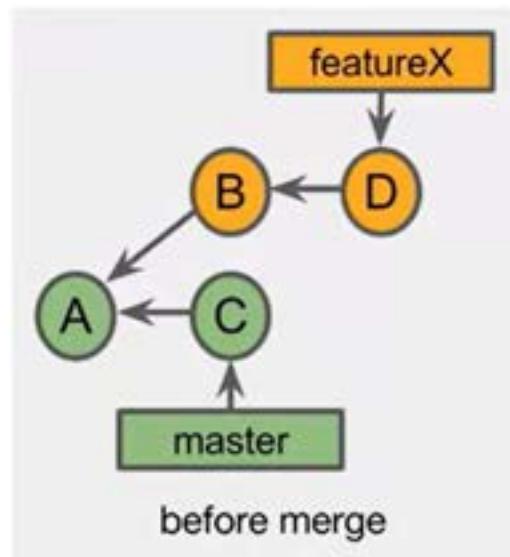
commit	file(s)
A	fileA.txt
B'	fileA.txt, fileB.txt, fileC.txt

*after interactive rebase with **delete commit B:***

commit	file(s)
A	fileA.txt
C'	fileA.txt, fileC.txt

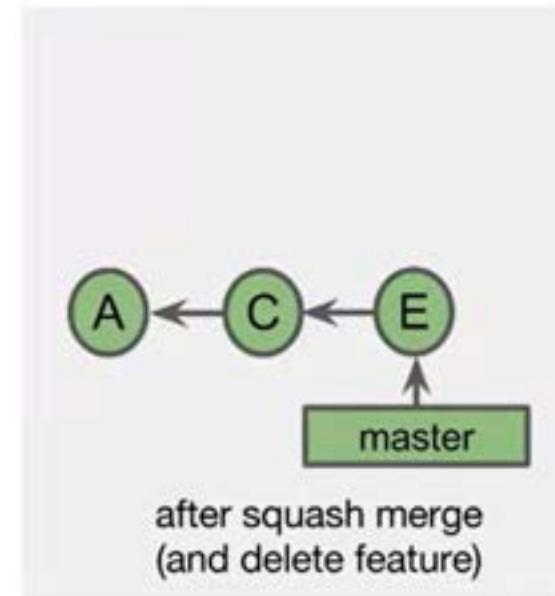
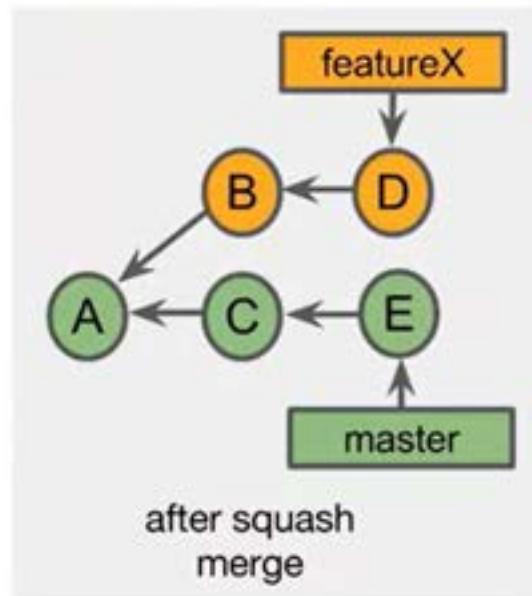
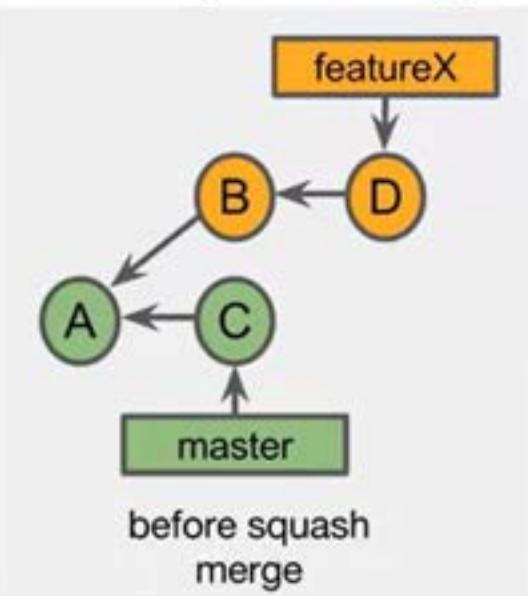
SQUASH MERGE

1. Merges the tip of the feature branch (D) onto the tip of the base branch (C)
 - o There is a chance of a merge conflict
2. Places the result in the **staging area**
3. The result can then be committed (E)



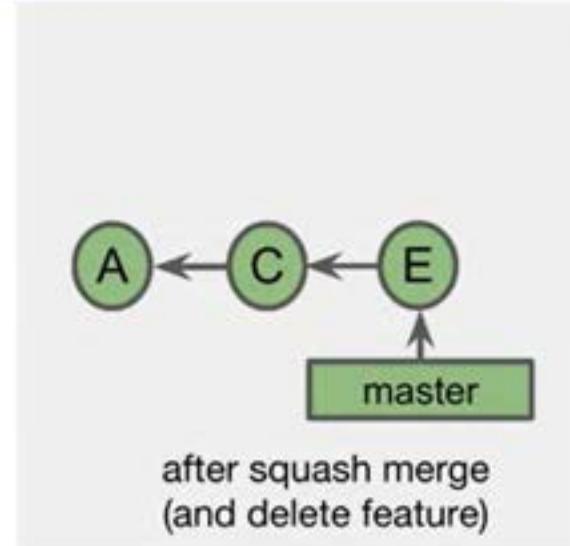
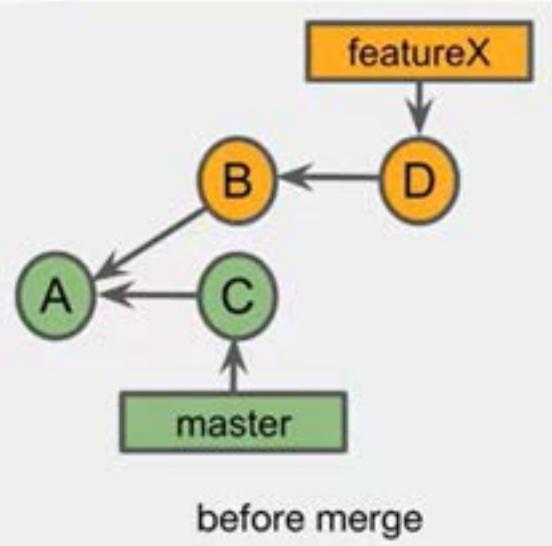
WHAT HAPPENS TO THE FEATURE COMMITS?

- After the **featureX** label is deleted, commits B and D are no longer part of any named branch
 - Commits B and D will eventually be garbage collected
- A squash merge *rewrites the commit history*



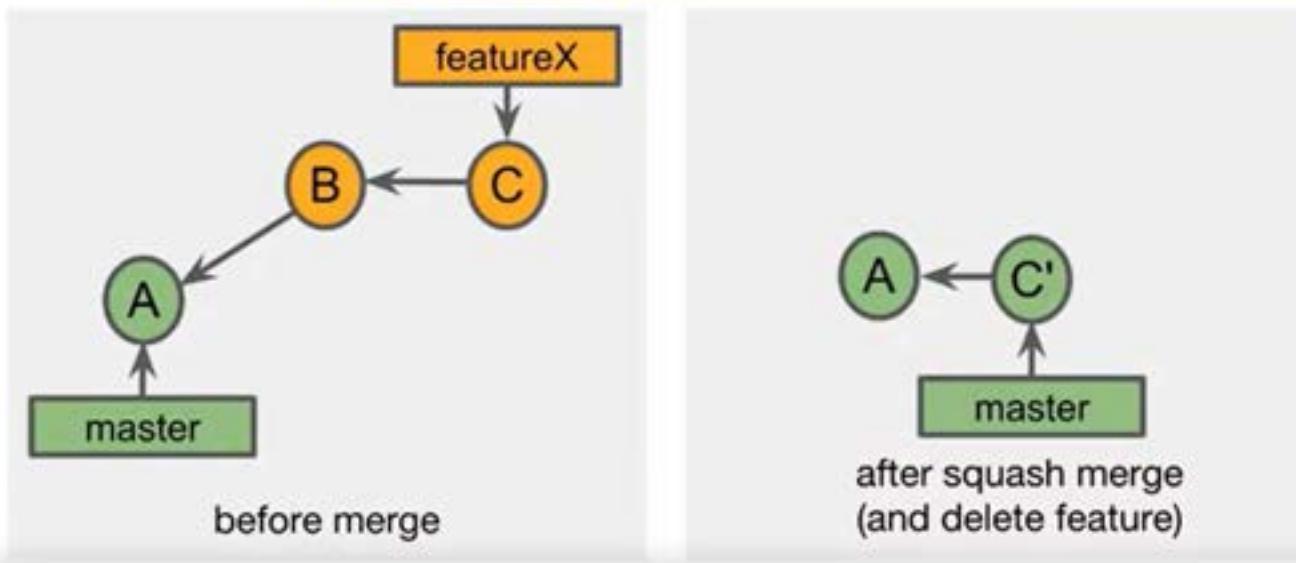
PERFORMING A SQUASH MERGE

1. git checkout **master**
2. git merge **--squash** featureX
3. git **commit**
 - a. accept or modify the **squash message**
4. git branch -D featureX



SQUASH MERGE WITH FAST-FORWARD

1. git checkout **master**
2. git merge --squash featureX
3. git **commit**
 - a. accept or modify the **squash message**
4. git branch -D featureX



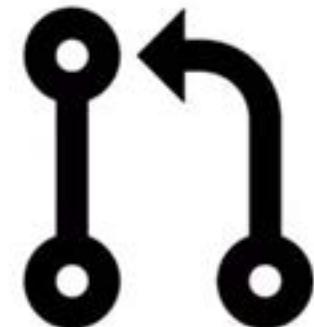
Topics

Pull request overview

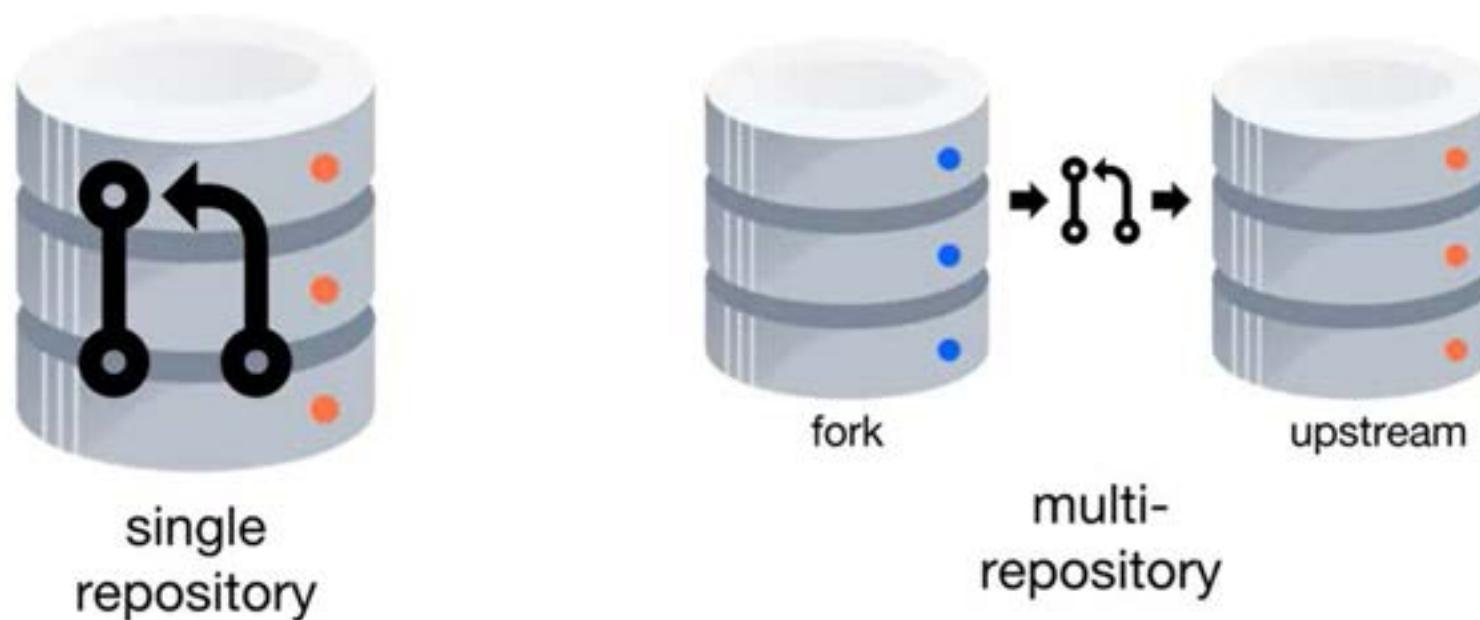
Single repository pull requests

PULL REQUESTS

- A feature of Git hosting sites
- The ultimate goal is to merge a branch into the project
- Enable team communication related to the work of the branch
 - Notifications sent to team members
 - Feedback or comments
 - Approval of the content (code review)



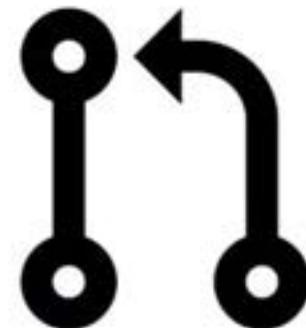
TWO REPOSITORY CONFIGURATIONS



We will cover multi-repository pull requests later

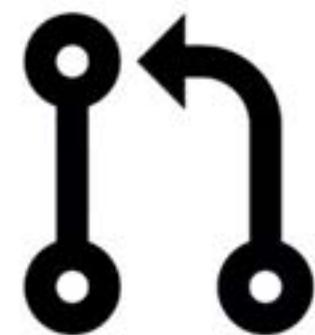
WHEN DO YOU OPEN A PULL REQUEST?

- When the branch is created
- When you want comments on the branch
- When the branch is ready for review/merging



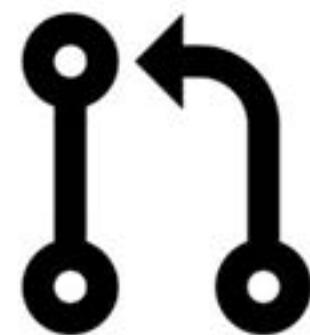
PREPARING FOR PULL REQUEST (SINGLE REPOSITORY)

1. Create a feature branch
2. Optionally work on the feature branch
3. Push the branch to the remote repository



PREPARING FOR PULL REQUEST (SINGLE REPOSITORY)

1. Create a feature branch
2. Optionally work on the feature branch
3. Push the branch to the remote repository



CREATE A PULL REQUEST

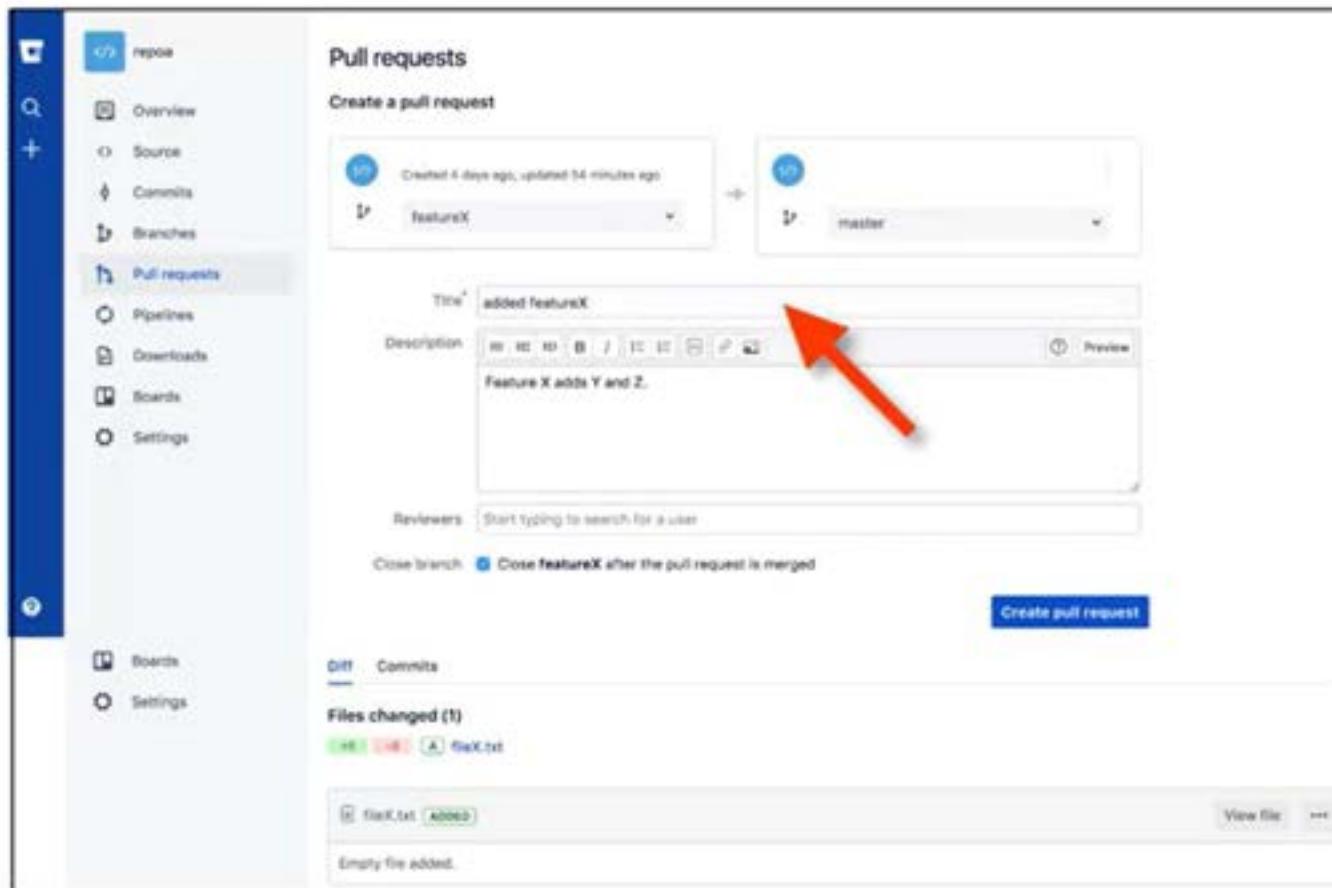
The screenshot shows a software interface for managing pull requests. On the left is a vertical sidebar with a blue header bar containing icons for repository management, search, and creation. Below this, a white sidebar lists various project management features: Overview, Source, Commits, Branches, Pull requests (which is selected and highlighted in blue), Pipelines, Downloads, Boards, and Settings.

The main content area has a header "Pull requests" and a "Create pull request" button. It includes a "FILTER BY:" section with dropdown menus for "Open", "Author", "Target branch", and "I'm reviewing", along with a search icon. The central message states "No open pull requests" and provides instructions: "When a branch is ready to review, [create a pull request](#)".

CREATE A PULL REQUEST

The screenshot shows a GitHub repository interface. On the left, there's a sidebar with icons for Overview, Source, Commits, Branches, Pull requests (which is selected and highlighted in blue), Pipelines, Downloads, Boards, and Settings. The main area is titled "Pull requests" and includes a "Create pull request" button. Below it, there are filter options: "FILTER BY: Open", "Author", "Target branch", and "I'm reviewing". A search bar is also present. The central message says "No open pull requests" and provides instructions: "When a branch is ready to review, [create a pull request](#)". A red arrow points to the "Create pull request" button.

FILL OUT AND CREATE THE PULL REQUEST



PULL REQUEST NOTIFICATIONS

The screenshot shows the Bitbucket dashboard. On the left, there's a sidebar with icons for Overview, Repositories, Projects, Pull requests, Issues, and Snippets. The 'Overview' item is highlighted with a dark blue background. The main area is titled 'Dashboard' and contains sections for 'Pull requests' and 'Created by you'. The 'Pull requests' section has tabs for 'Waiting on your review', 'Activity', 'Reviewers', and 'Builds'. It displays a message: 'No pull requests are waiting on your review. View the complete list'. The 'Created by you' section shows a single pull request: 'added featureX → master #1, last updated 16 hours ago in repos'.

REVIEWING A PULL REQUEST

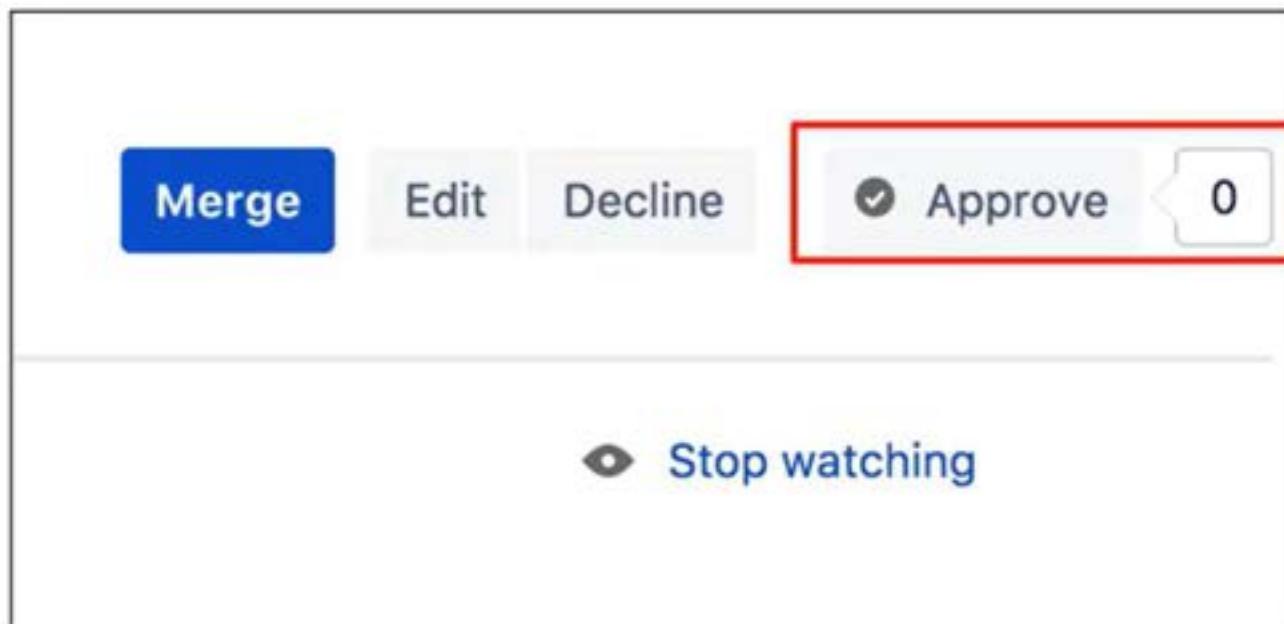
All comments and new commits are visible to the reviewer

The screenshot shows a pull request interface for a repository. At the top, it says "added featureX" and has a status bar with "#1 OPEN" and branches "featureX" and "master". Below this are buttons for "Merge", "Edit", "Decline", "Approve", and a comment count of "0". There is also a "Stop watching" link.

The main area includes sections for "Reviewers" (No reviewers), "Description" (Feature X adds Y and Z.), and "Comments (0)" with a text input field. The "Files changed (1)" section shows a single file, "fileX.txt", with a status of "ADDED". Below this, a note says "Empty file added."

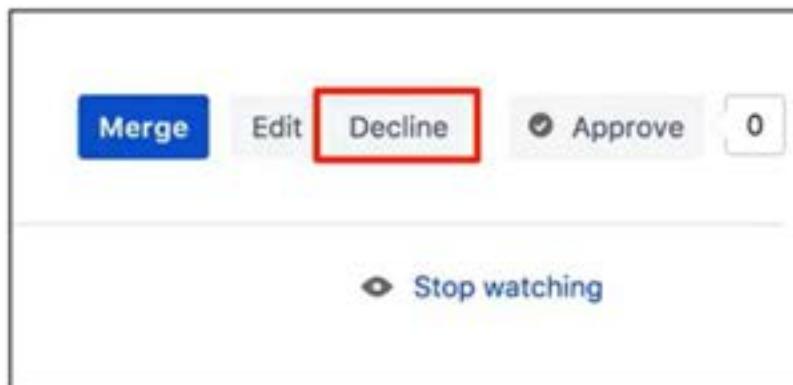
APPROVING A PULL REQUEST

Click **Approve** to add to the count of approvers of the pull request



DECLINE THE PULL REQUEST

Click **Decline** to reject and remove the pull request



Decline pull request

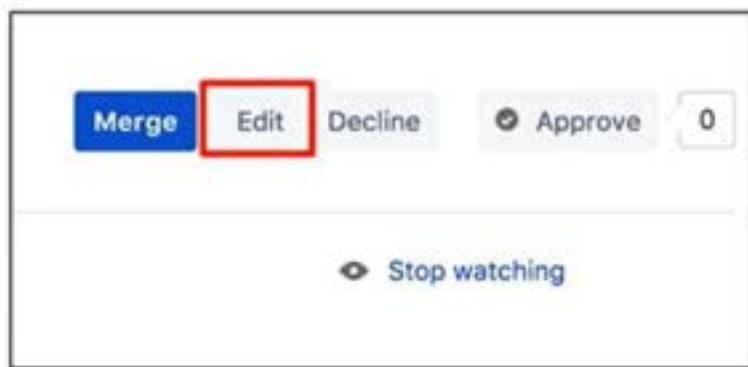
⚠ Declining a pull request is permanent and cannot be undone. If you have feedback for the author, we recommend leaving comments on the pull request instead of declining.

Reason

Why are you declining this pull request?

Decline **Cancel**

EDIT A PULL REQUEST



Pull requests

Update pull request #1

Created 5 days ago, updated an hour ago

featureX → master

Title

Description

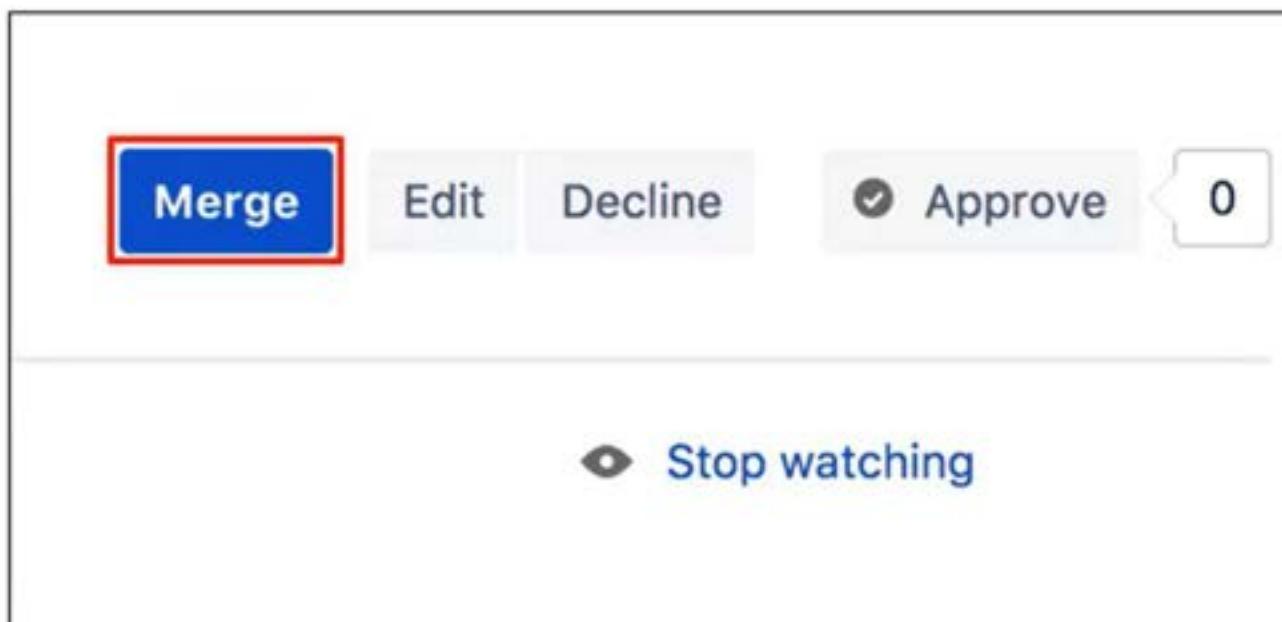
Feature X adds Y and Z.

Reviewers

Close branch Close featureX after the pull request is merged

MERGE A PULL REQUEST

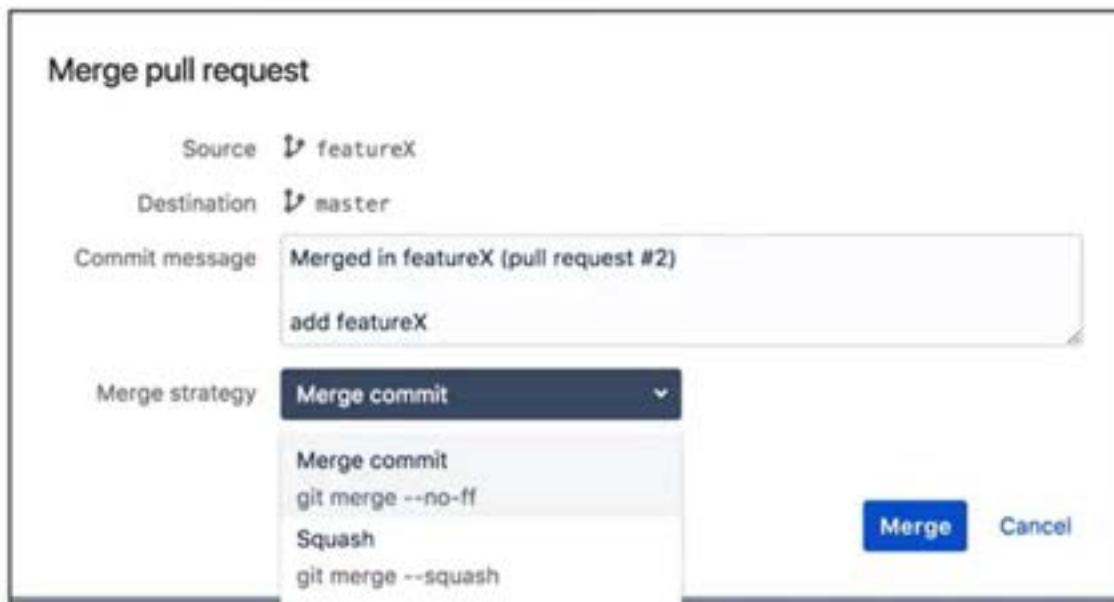
Click **Merge** to begin the process of merging the branch



You can also merge from the command line or Sourcetree and the pull request will be closed

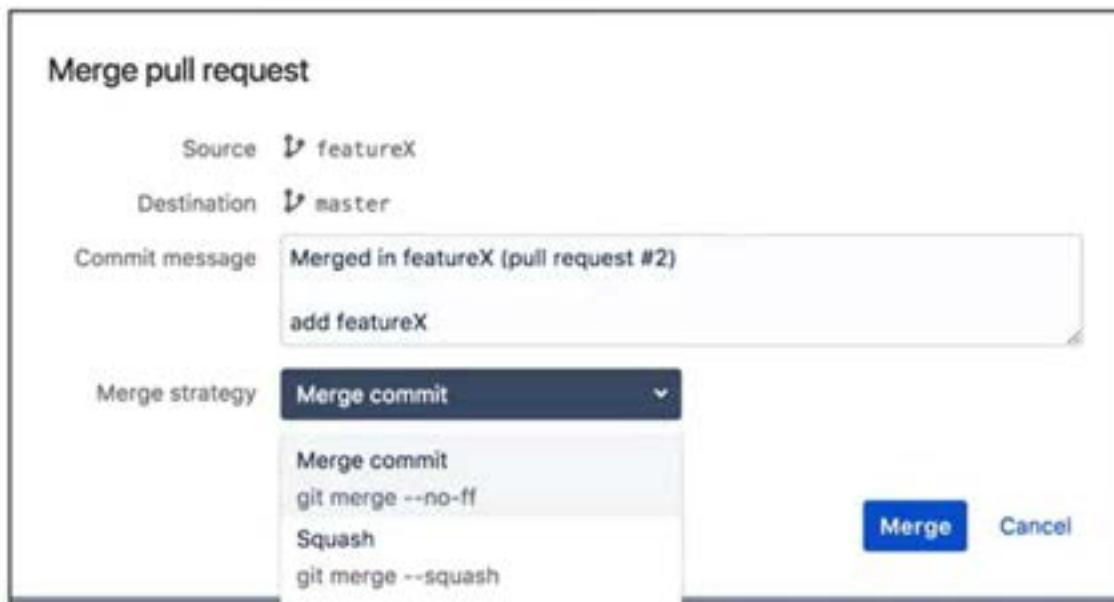
MERGE STRATEGY

- Merge commit- the merge creates a separate commit object (`git merge --no-ff`)
- Squash- the entire branch is condensed to one linear commit (`git merge --squash`)



MERGE STRATEGY

- Merge commit- the merge creates a separate commit object (`git merge --no-ff`)
- Squash- the entire branch is condensed to one linear commit (`git merge --squash`)



REVIEW

- Pull requests are opened using an online Git host such as Bitbucket or GitHub
- The ultimate goal of a pull request is to merge a branch, but they also facilitate team discussion and approval
- You can open a pull request any time after creating the branch
- You do not need to edit the pull request if you add a commit to the branch

REVIEW

- Pull requests are opened using an online Git host such as Bitbucket or GitHub
- The ultimate goal of a pull request is to merge a branch, but they also facilitate team discussion and approval
- You can open a pull request any time after creating the branch
- You do not need to edit the pull request if you add a commit to the branch
- You can merge the pull request using an online Git host or by pushing the merge from your local client

PREPARING FOR PULL REQUEST (SINGLE REPOSITORY)

- Create a feature branch
- Optionally work on the feature branch
- Push the branch to the remote repository

```
$ git checkout -b "featureX"
Switched to a new branch 'featureX'
$ touch fileA.txt
$ git add fileA.txt
$ git commit -m "added featureX"
[featureX 52c3153] added featureX
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 fileX.txt
$ git push --set-upstream origin featureX
(snip)
remote: Create pull request for featureX: 
remote: https://bitbucket.org/me/repoa/pull-requests/new?source=featureX&t=1
To https://bitbucket.org/me/repoa.git
 * [new branch]      featureX -> featureX
Branch featureX set up to track remote branch featureX from origin.
```

CREATE A PULL REQUEST

The screenshot shows a software interface for managing pull requests. On the left is a vertical sidebar with icons for Overview, Source, Commits, Branches, Pull requests (which is selected), Pipelines, Downloads, Boards, and Settings. The main area is titled "Pull requests" and displays a message: "No open pull requests". Below this, it says "When a branch is ready to review, [create a pull request](#)". A red arrow points to this "create a pull request" link.

repoa

Pull requests

FILTER BY: Open Author Target branch I'm reviewing

No open pull requests

When a branch is ready to review, [create a pull request](#).

Create pull request

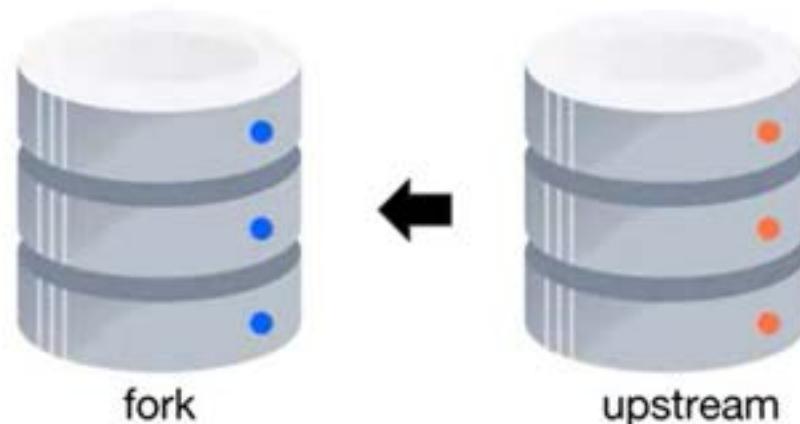
DELETING REMOTE BRANCH LABELS

git push -d <remote> <branch>

```
$ git push -d origin featureX
To https://bitbucket.org/me/repoa.git
  - [deleted]          featureX
```

FORKING

- *Forking*- copying a remote repository to your own online account
- Both repositories are remote repositories
- The *upstream* repository is usually the "source of truth"

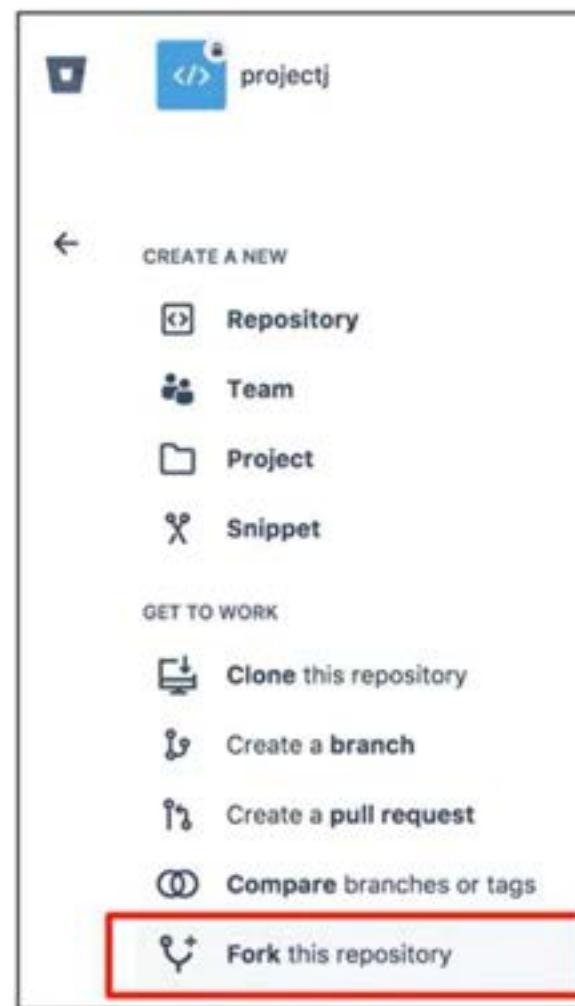
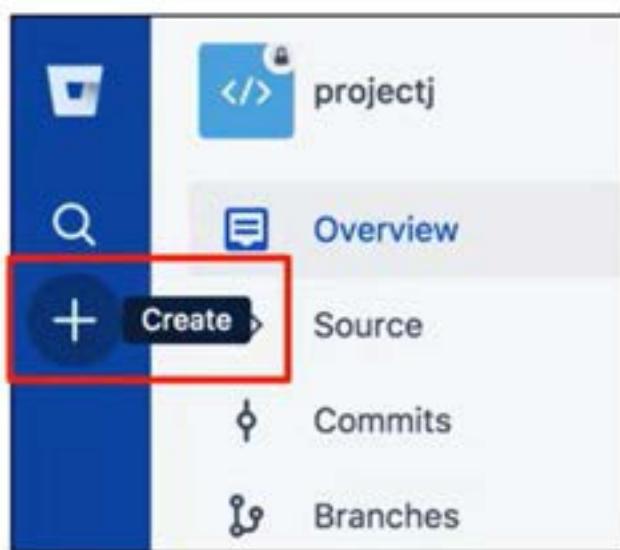


WHAT IS A FORK USED FOR?

- Experiment with/learn from the upstream repository
- Issue pull requests to the upstream repository
- Create a different source of truth

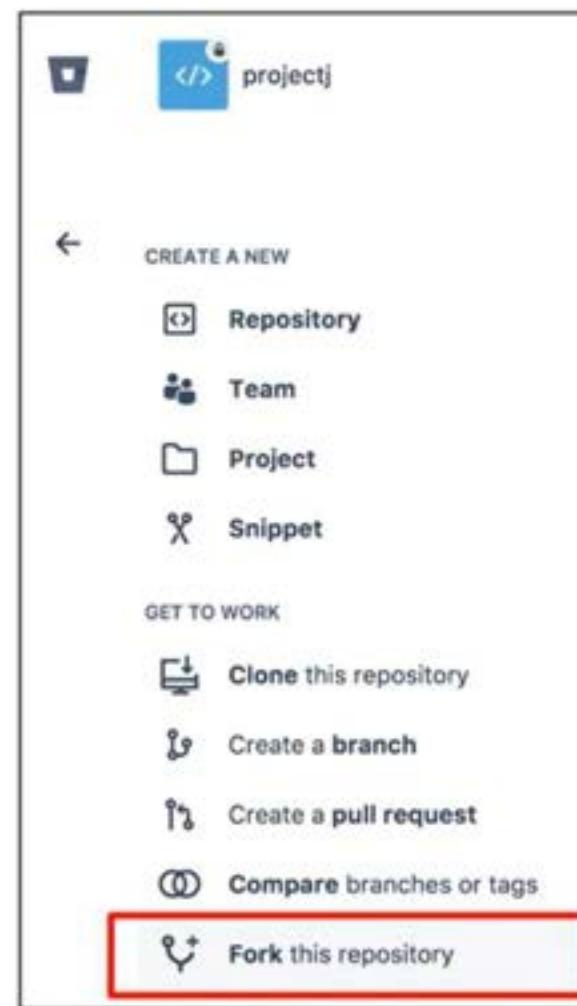
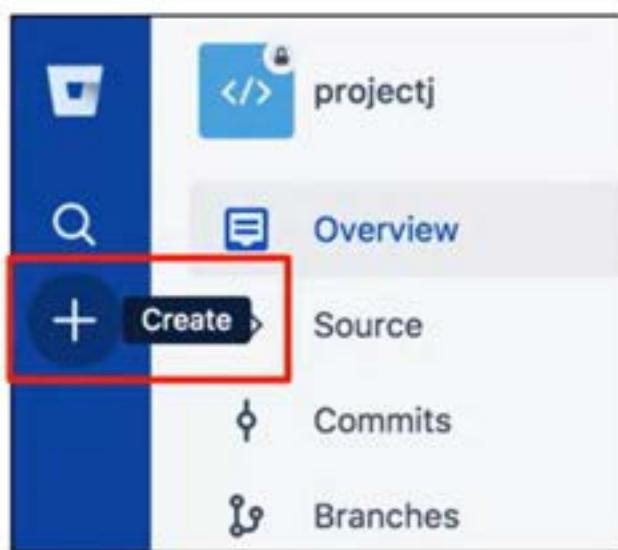
FORKING A REPOSITORY

1. In Bitbucket, navigate to the repository that you want to fork
2. Click +
3. Select **Fork this repository**



FORKING A REPOSITORY

1. In Bitbucket, navigate to the repository that you want to fork
2. Click +
3. Select **Fork this repository**



NAME YOUR FORKED REPOSITORY

The screenshot shows a web browser window with the URL <https://bitbucket.org/pat/projectj/fork>. A modal dialog box titled "Fork projectj" is open. Inside the dialog, there is a "Name" field containing "projectjfork". Below it, an "Access level" section has a checked checkbox labeled "This is a private repository". A red arrow points from the text "This repository does not allow public forks." to the checkbox. At the bottom of the dialog are two buttons: "Fork repository" (in blue) and "Cancel".

https://bitbucket.org/pat/projectj/fork

Fork projectj

Name

Access level This is a private repository

This repository does not allow public forks.

› Advanced settings

Fork repository Cancel

RESULT OF A FORK

Repositories

FILTER BY: Owner ▾ Project ▾

Repository
 projectj
 projectifork

SYNCING A FORK- FROM BITBUCKET (1 of 2)

Overview

Last updated 12 minutes ago

Fork of projectj

Access level Admin

0	1
Open PRs	Watcher
1	0
Branch	Forks

Edit README

Share

Invite users to this repo

Send invitation

This fork is 1 commit behind projectj. Sync now.

The screenshot shows the 'Overview' section of a Bitbucket fork. It includes a download link, HTTPS dropdown, share button, and a 'Send invitation' button. The main stats table shows 0 Open PRs, 1 Watcher, 1 Branch, and 0 Forks. A message box at the bottom right says 'This fork is 1 commit behind projectj. Sync now.' with a red border around it. The URL in the address bar is https://bitbucket.org/username/projectj/fork.

SYNCING A FORK- FROM BITBUCKET (2 of 2)

- Syncing via Bitbucket creates a merge commit on the forked repository
- This commit is not in the upstream repository



VIEWING THE MERGE COMMIT

Commit

5bc2f3a MERGE

Merged projectj into master

Comments (0)

What do you want to say?

Files changed (1)

+2 -0 M README.md

README.md MODIFIED

```
1 1 # PROJECTJ README #
2 +
3 +fun with forks
```

projectjfork (Git)

Push Fetch Branch Merge Stash

All Branches Show Remote Branches Ancestor Order

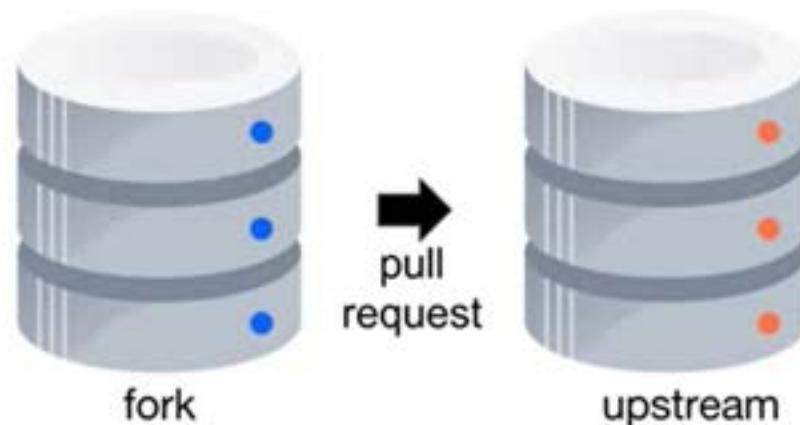
Graph Description

master origin/master origin/HEAD Merged projectj into master

add line to README.md
add README.md

MULTI-REPOSITORY PULL REQUESTS

1. Fork the upstream repository
2. Create a branch
3. Create a pull request



CREATE THE PULL REQUEST

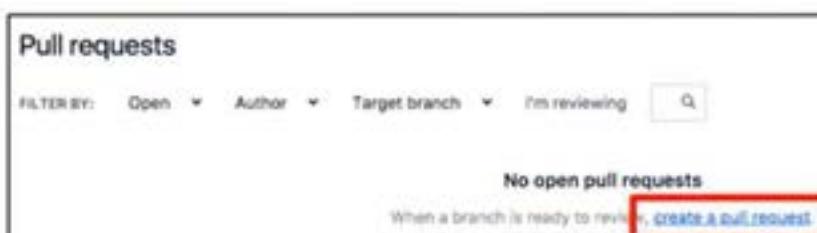
Most of the information is added for you

Pull requests

FILTER BY: Open Author Target branch I'm reviewing Q

No open pull requests

When a branch is ready to review, [CREATE A PULL REQUEST](#)



Create a pull request

From: project/fork
Created an hour ago, updated 3 minutes ago

To: project
master

Title:

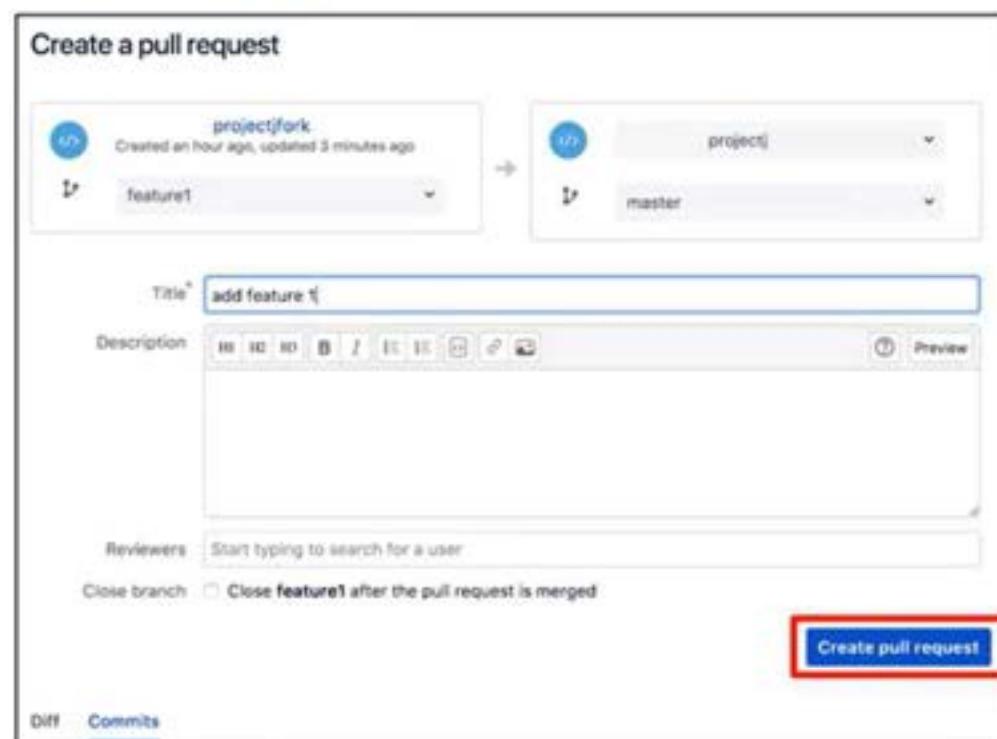
Description:

Reviewers: Start typing to search for a user

Close branch Close feature1 after the pull request is merged

[Create pull request](#)

Diff Commits



THE PULL REQUEST ON UPSTREAM

Pull requests

FILTER BY: Open ▾ Author ▾ Target branch ▾

Summary

add feature 1 → master

#1, last updated a moment ago

MERGING MULTI-REPOSITORY PULL REQUESTS

Use the Bitbucket interface

or

1. Add the forked repository as a remote
2. Perform and push the merge

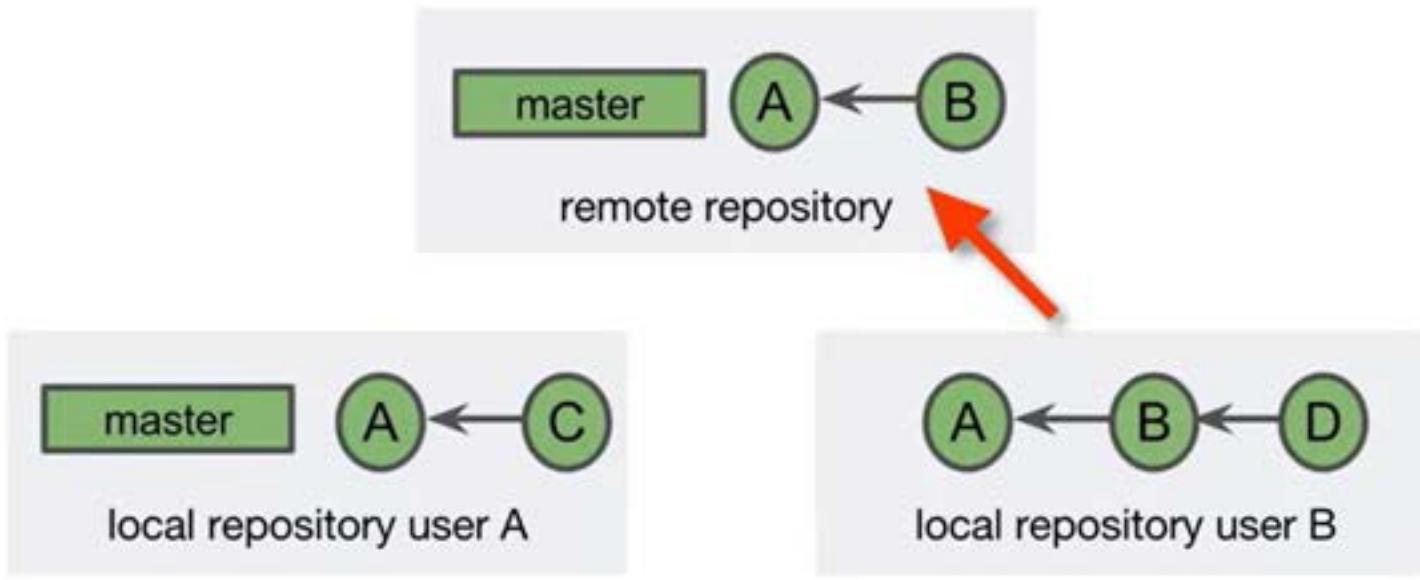
REVIEW

- A fork is a remote copy of an upstream remote repository
- A fork is created using an online Git hosting provider
- Forks and upstream repositories may become out of sync
- Pull requests can be made from forks and merged into the upstream repository

Git Workflows

Copyright © 2018 Atlassian

CENTRALIZED WORKFLOW



- only a single branch
- no pull requests/discussion

Topics

Centralized workflow

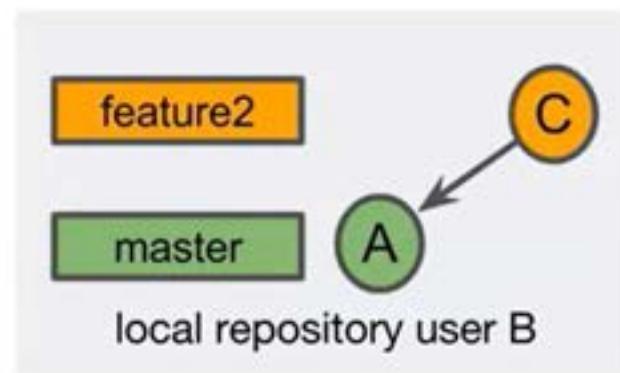
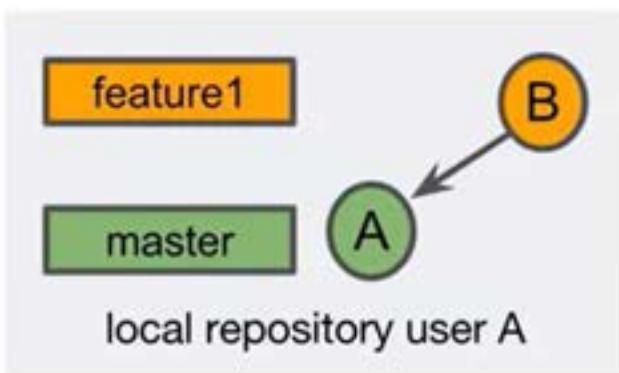
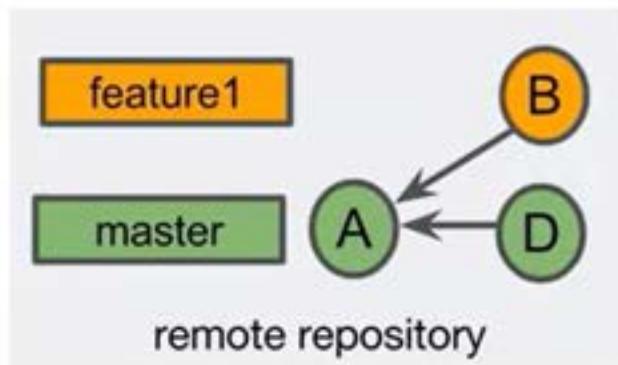
Feature branch workflow

Forking workflow

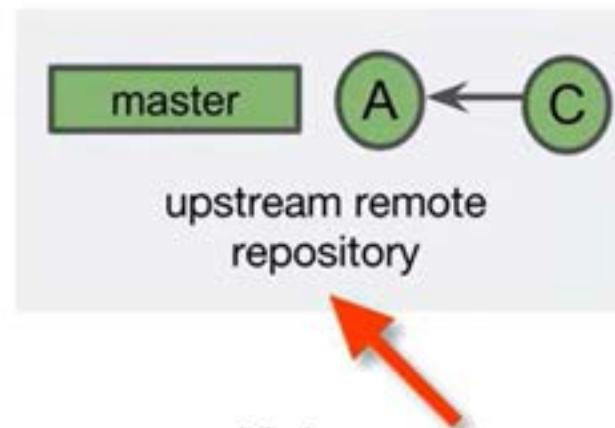
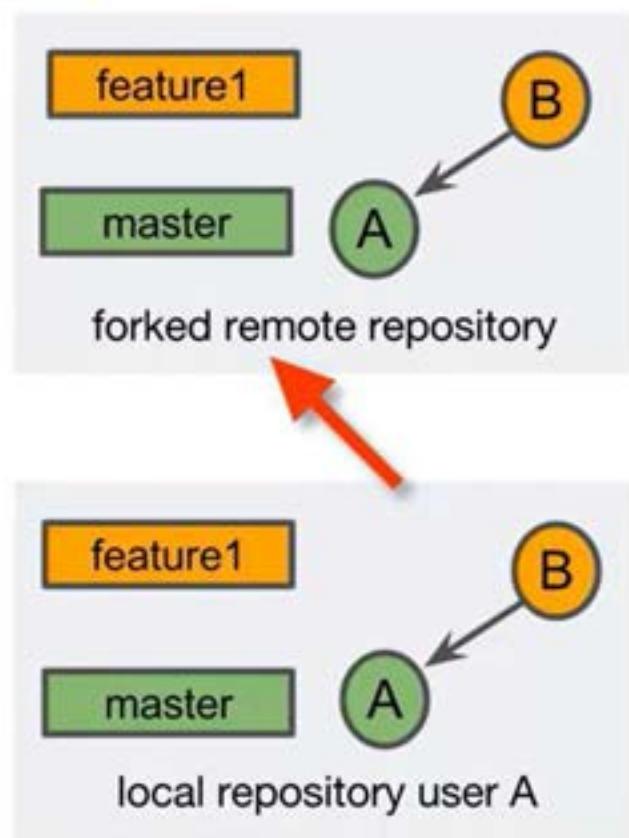
Gitflow workflow

FEATURE BRANCH WORKFLOW

- work done on feature/topic branches
- single remote repository
- pull requests/discussion

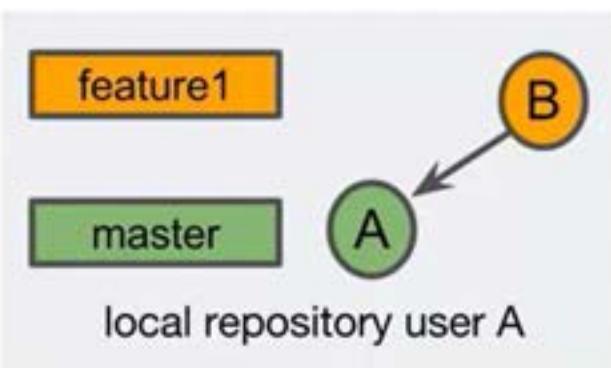
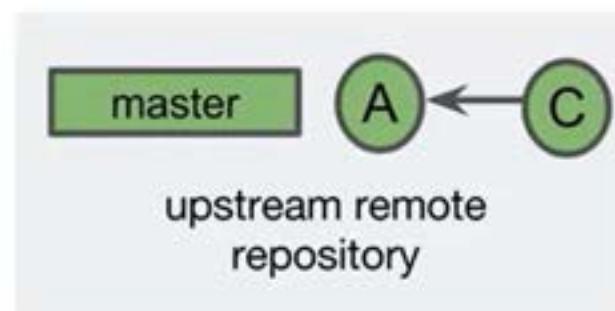
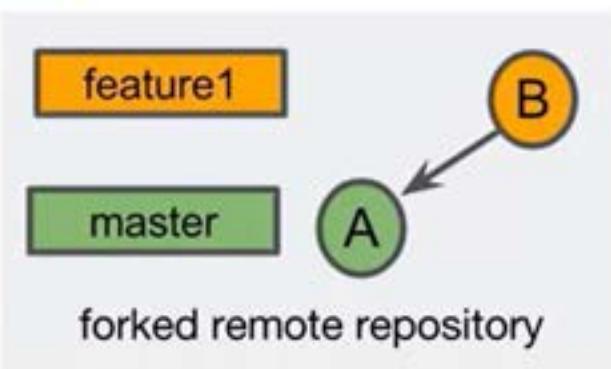


FORKING WORKFLOW



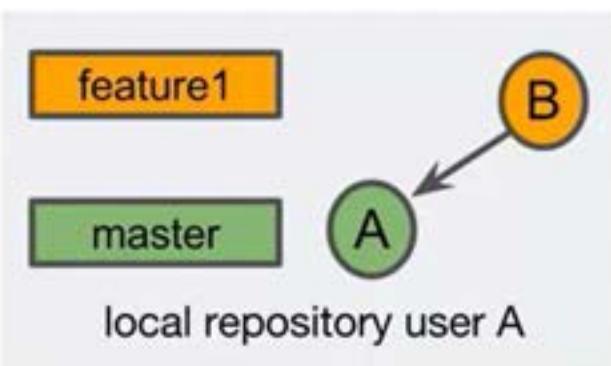
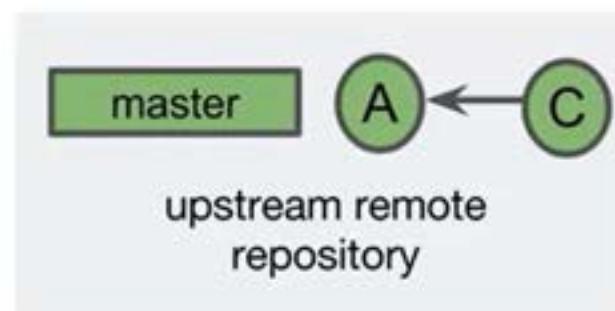
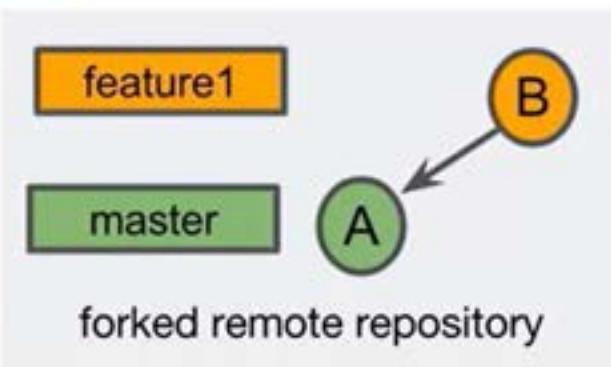
- multiple remote repositories
- pull requests/discussion
- don't need write access on upstream
- backs up your work in progress
- can rebase your forked branch
- must synchronize with upstream

FORKING WORKFLOW



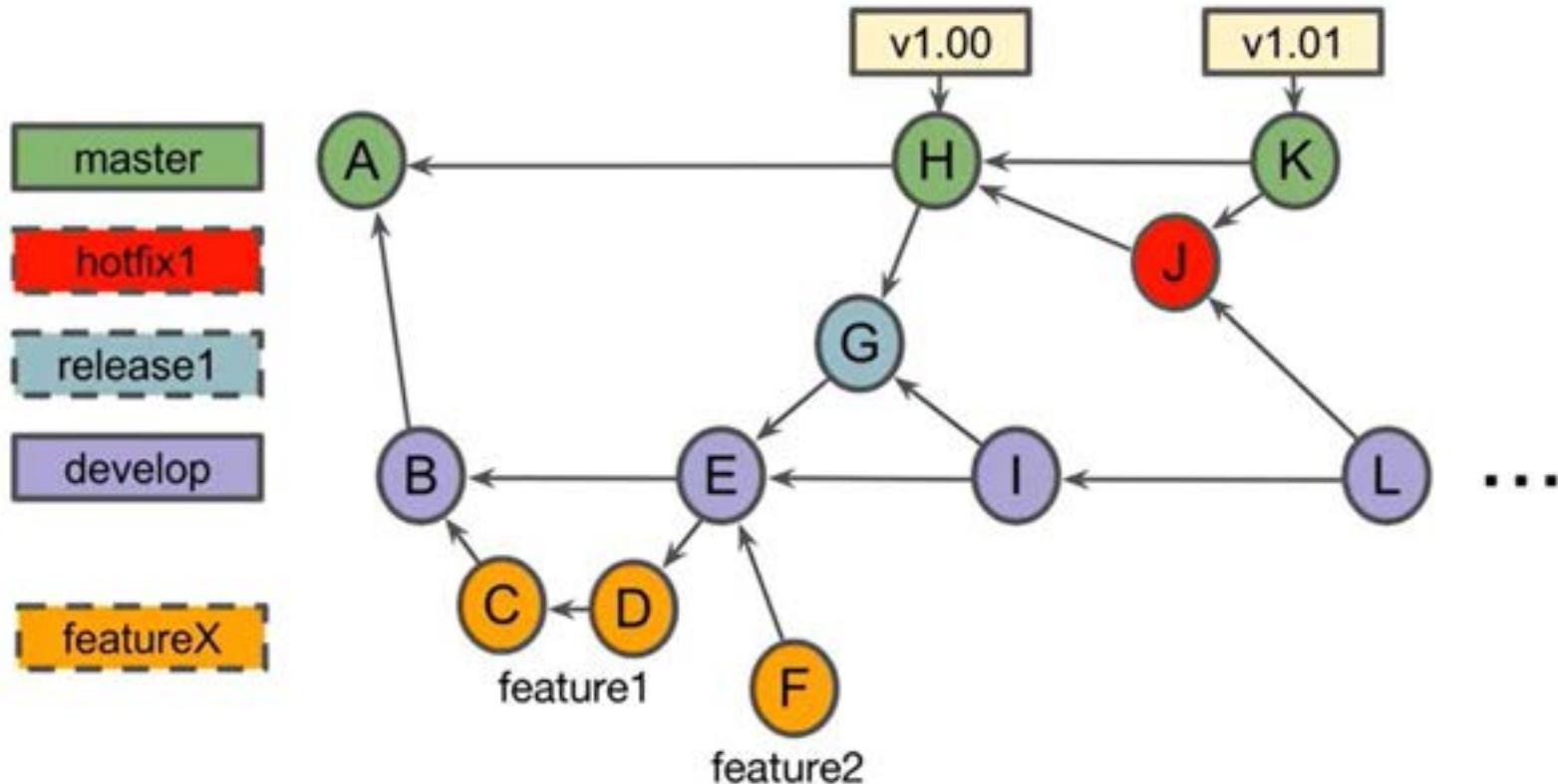
- multiple remote repositories
- pull requests/discussion
- don't need write access on upstream
- backs up your work in progress
- can rebase your forked branch
- must synchronize with upstream

FORKING WORKFLOW



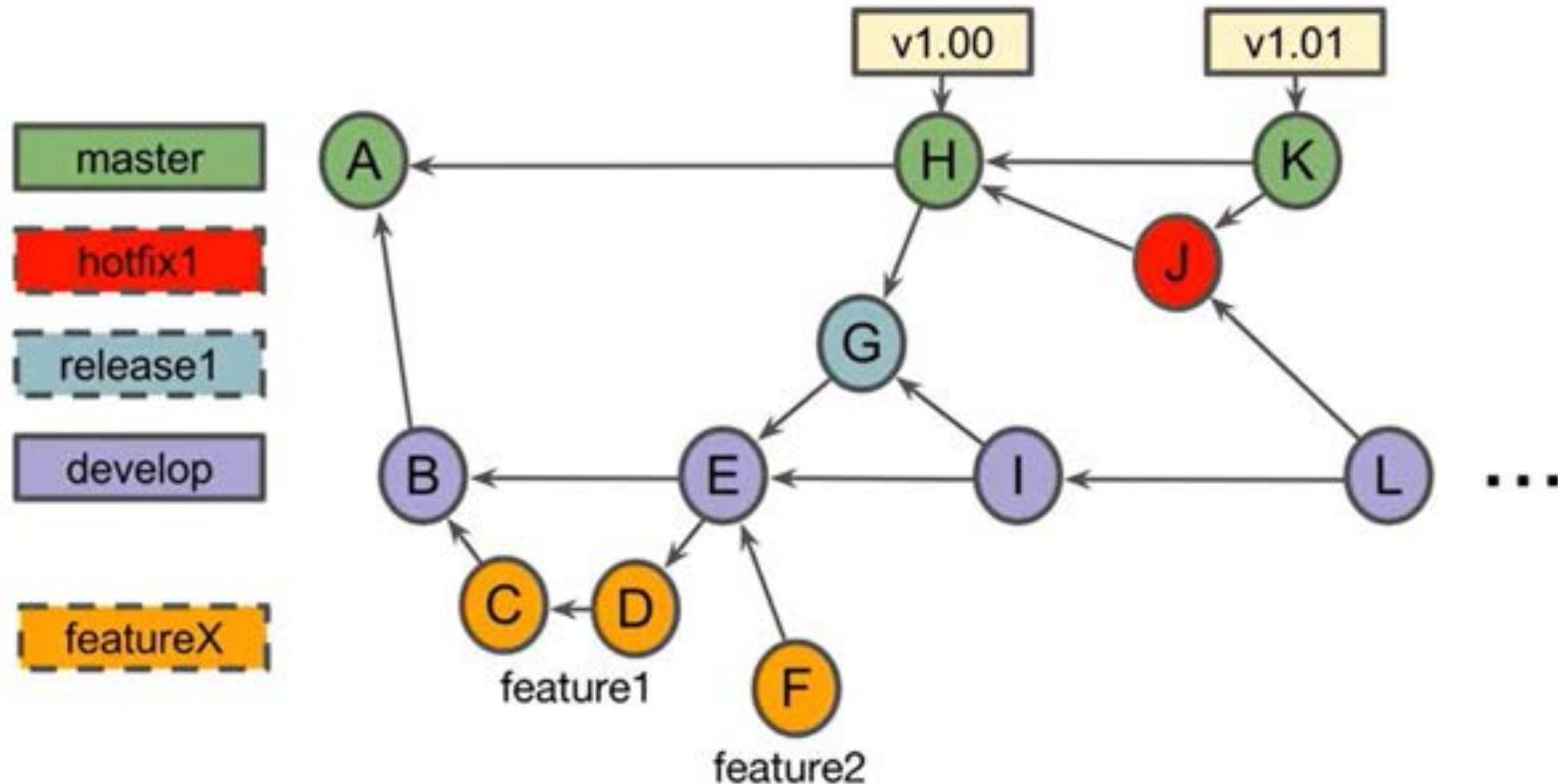
- multiple remote repositories
- pull requests/discussion
- don't need write access on upstream
- backs up your work in progress
- can rebase your forked branch
- must synchronize with upstream

GITFLOW



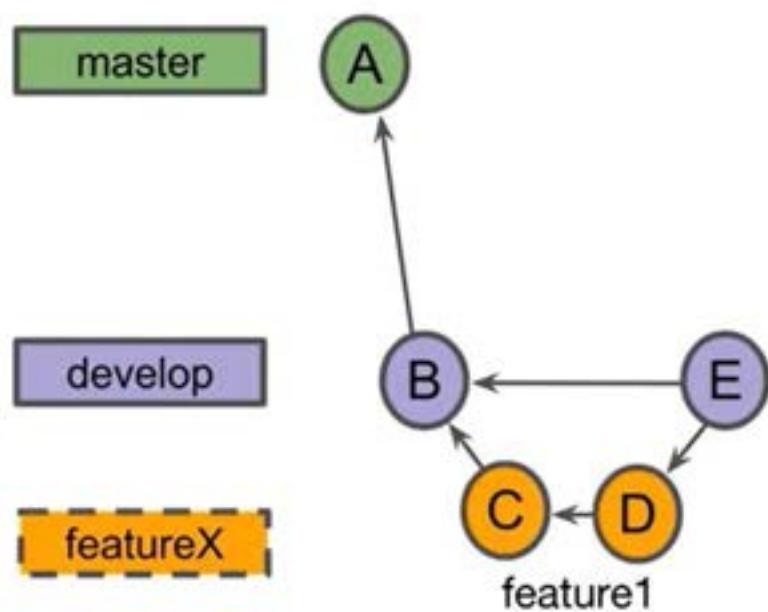
<https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow>

GITFLOW



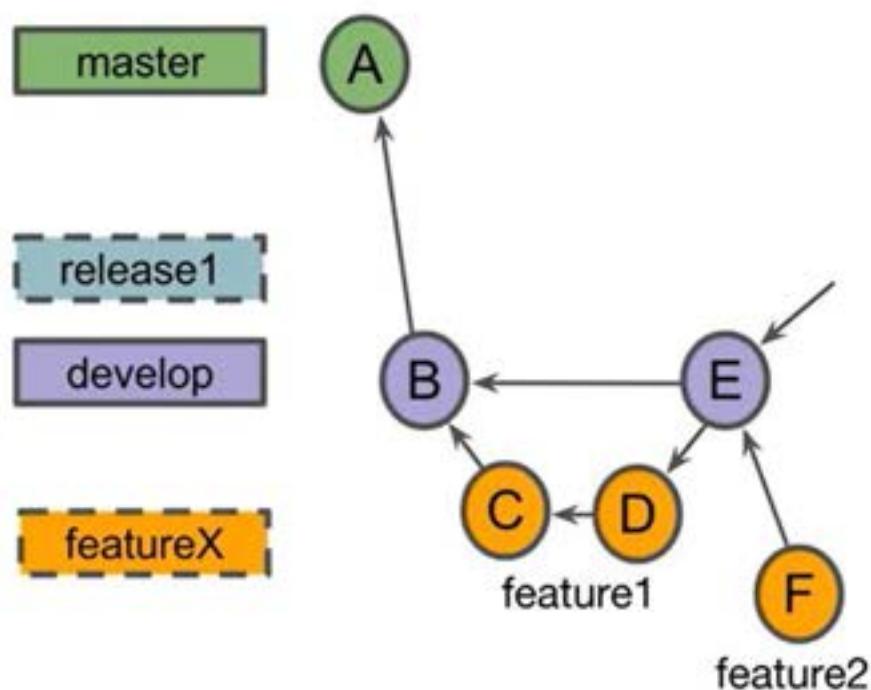
<https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow>

GITFLOW



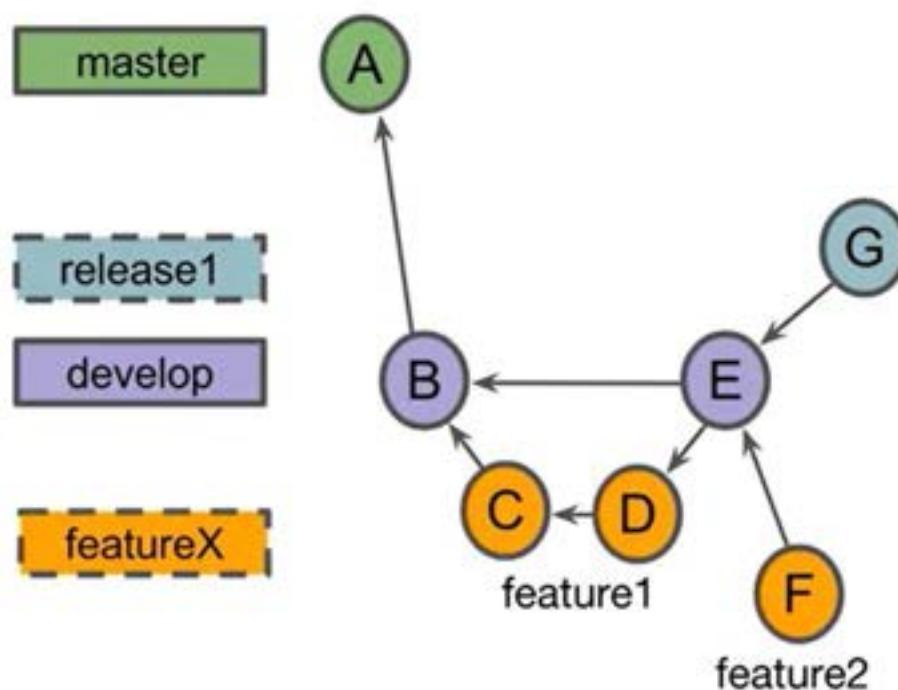
1. Final work on feature 1 is done in commit D
2. Team decides feature 1 is ready
3. Merge commit E is created, adding feature 1 to the project
4. **feature1** branch label can be deleted

GITFLOW



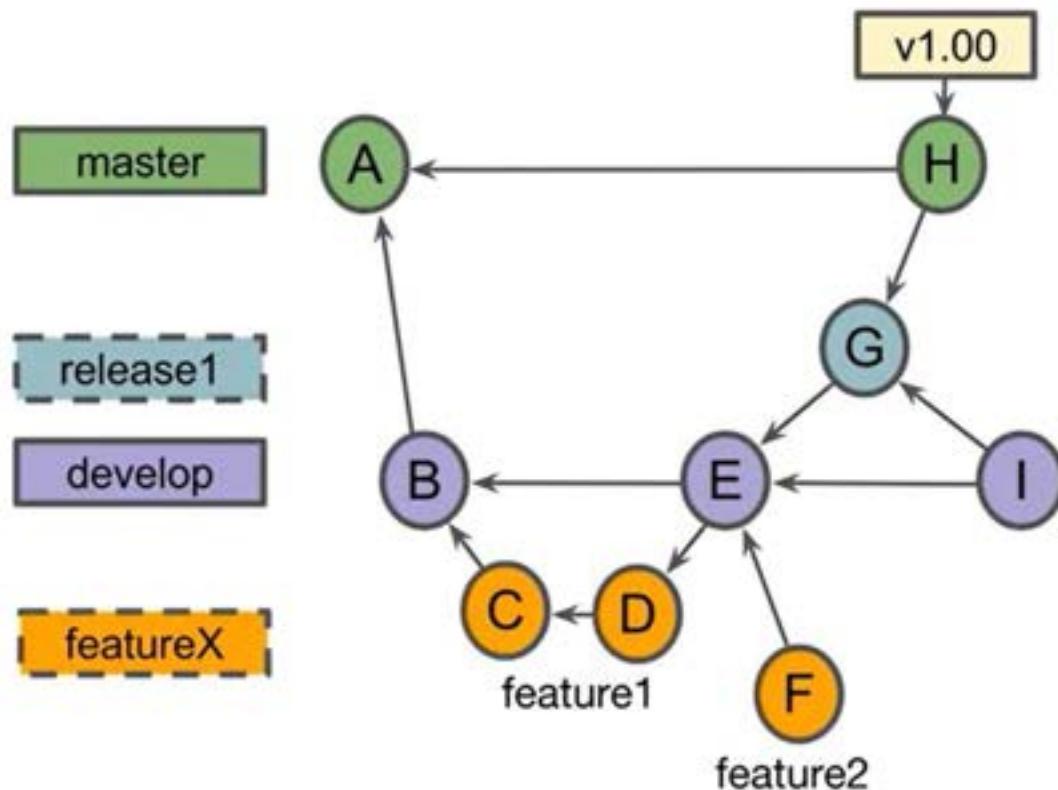
1. Team decides commit E is a release candidate
2. Creates a **release1** branch off commit E (no commits yet)
3. Developer creates **feature2** branch and creates commit F

GITFLOW



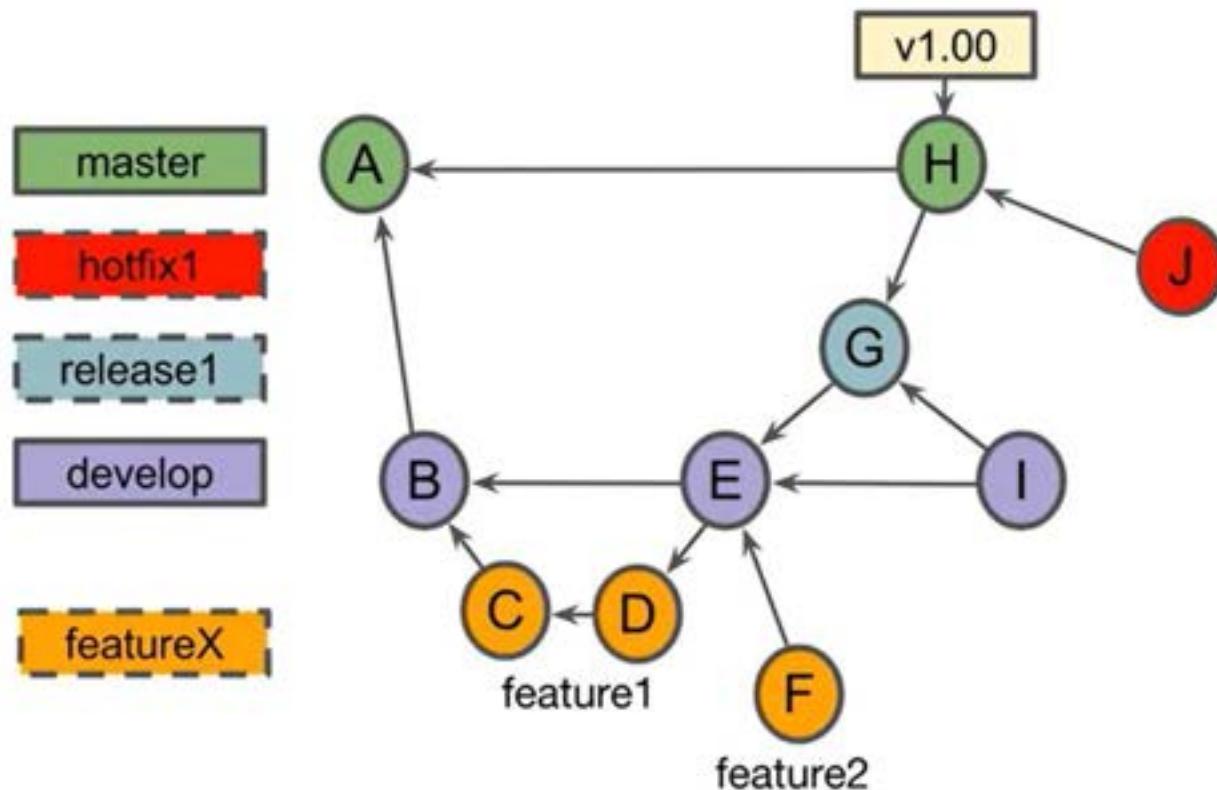
1. The team discovers a bug in commit E
2. Creates commit G on the **release1** branch

GITFLOW



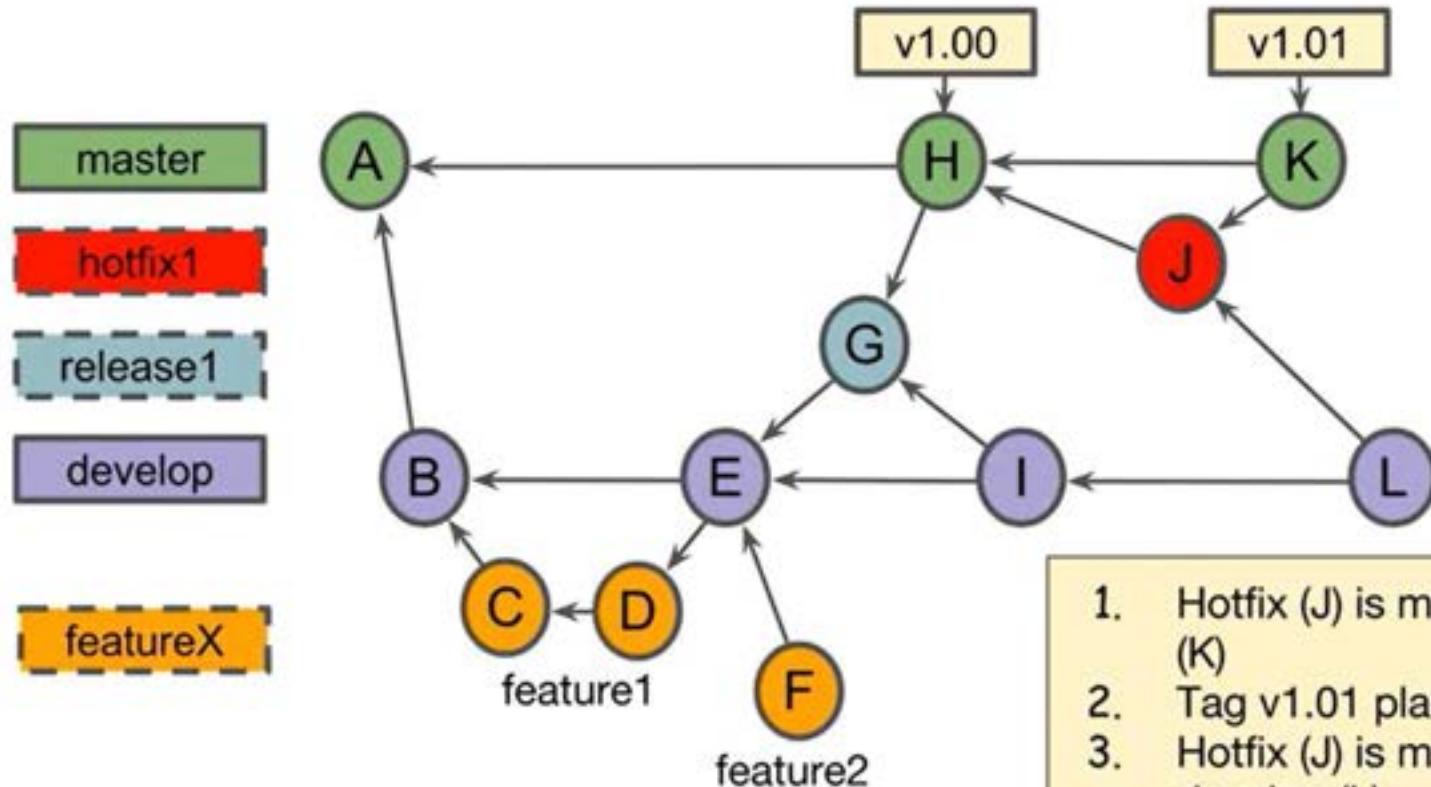
1. Commit G is approved
2. Create merge commit H
3. Tag commit H with "v1.00"
4. Create merge commit I on develop to incorporate bug fix from commit G
5. **release1** branch label can be removed

GITFLOW



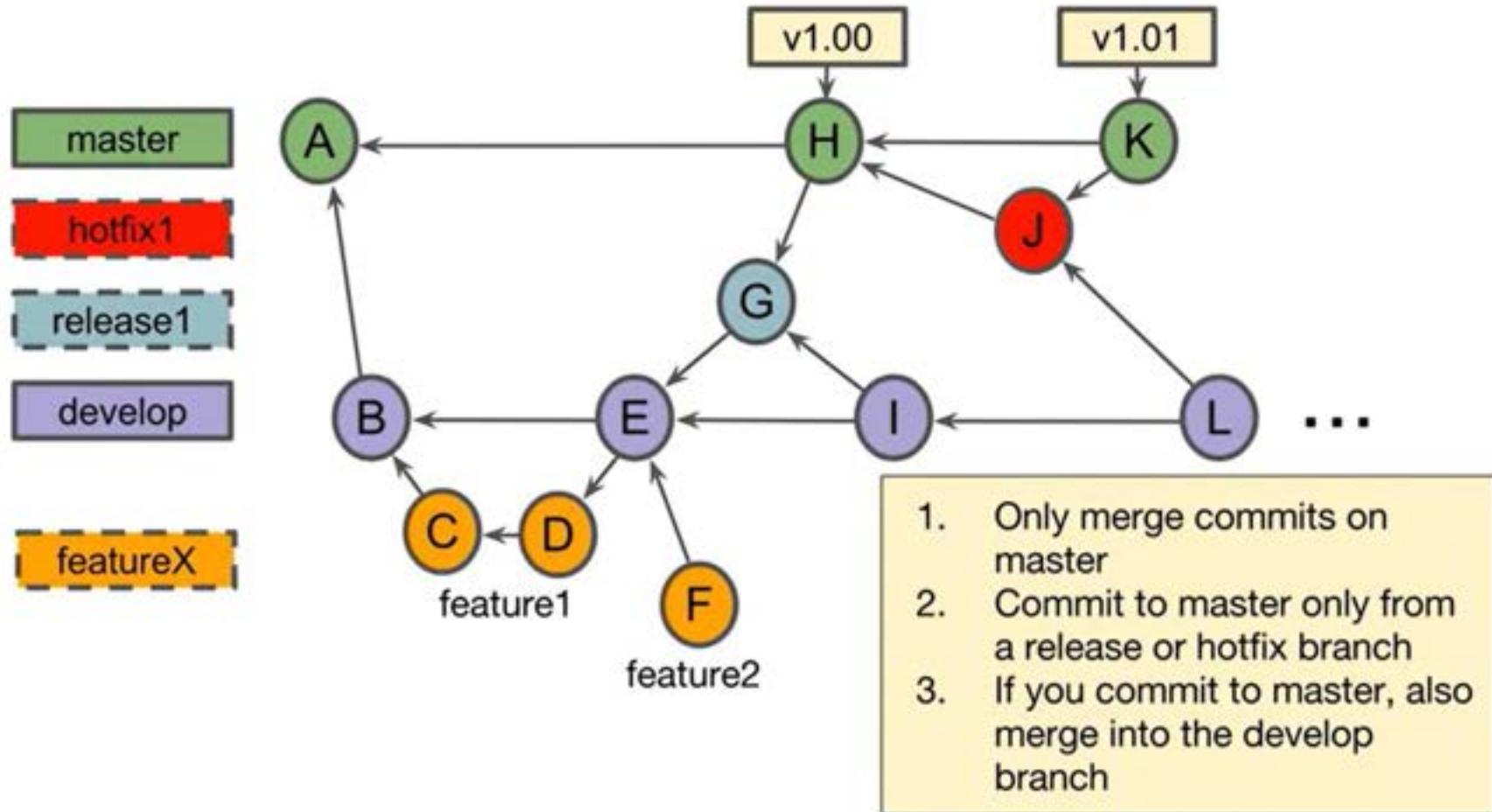
1. Uh oh, problem with v1.00
2. Create **hotfix1** branch
3. Create commit J to fix the issue

GITFLOW



1. Hotfix (J) is merged into master (K)
 2. Tag v1.01 placed on commit K
 3. Hotfix (J) is merged into develop (L)
 4. **hotfix1** branch label can be deleted

GITFLOW- MERGING "RULES"



GITFLOW- MERGING "RULES"

