

# Comprehensive Git using GitHub

# Welcome!

- Introductions
- Quick Agenda Review
- Class Logistics:
  - Hours: 7:00-3:00 PST
  - Lunch & Breaks
- Teaming for Success / How to Communicate
  - All questions encouraged
  - Realtime feedback is great
  - Chat / raise hand features
- Class Materials
  - Will be emailed to you directly for local use.
  - Class Surveys

# Table of Contents

1. Getting Started with Collaboration
2. Understanding the GitHub Flow
3. Branching with Git
4. Local Git Configuration
5. Working Locally with Git
6. Collaborating on Your Code
7. Merging Pull Requests
8. Viewing Local Project History
9. Streaming Your Workflow with Aliases
10. Workflow Review Project:
11. Resolving Merge Conflicts
12. Working with Multiple Conflicts

# Table of Contents

- 13. Searching for Events in Your Code
- 14. Reverting Commits
- 15. Helpful Git Commands
- 16. Viewing Local Changes
- 17. Creating a New Local Repository
- 18. Fixing Commit Mistakes
- 19. Rewriting History with Git Reset
- 20. Getting it Back
- 21. Merge Strategies: Rebase
- 23. Complex Workflows

# 1. Getting Started with Collaboration

# What is Git?

- Git is a free and open source distributed code management and Version control system that is distributed under the GNU General Public License version 2.
- In addition to software version control, Git is used for other applications including configuration management and content management.

# What is GitHub?

- People use GitHub to build some of the most advanced technologies in the world.
- Whether you're visualizing data or building a new game, there's a whole community and set of tools on GitHub that can help you do it even better.



*GitHub is a collaboration platform that uses Git for versioning. GitHub is a popular place to share and contribute to open-source software.*

# Exploring a GitHub Repository

- A repository is the most basic element of GitHub, They're easiest to imagine as a project's folder.
- A repository contains all of the project files (including documentation), and stores each file's revision history.
- Repositories can have multiple collaborators and can be either public or private.

# Comparison of Git vs GitHub

Aspect	Git	GitHub
Definition	A distributed version control system.	A web-based platform for hosting Git repositories.
Functionality	Tracks changes, manages versions, and branches.	Adds collaboration, hosting, and community features.
Installation	Installed on a local machine.	Accessed via a web browser (no installation needed).
Collaboration	Supports local collaboration through Git.	Enables online collaboration with teams.
Platform	Command-line based tool.	GUI-based online tool (with CLI integration).
Storage	Local repository on your computer.	Remote repositories stored in the cloud.
Usage	For version control of code locally or remotely.	To share and collaborate on Git repositories.
Integration	Works with any hosting platform (e.g., GitHub, GitLab).	Tightly integrated with Git, provides additional services.
Access Control	Local file permissions.	Role-based permissions for collaborators.
Community	No social features.	Social features like stars, forks, and followers.

## 2. Understanding Git Flow

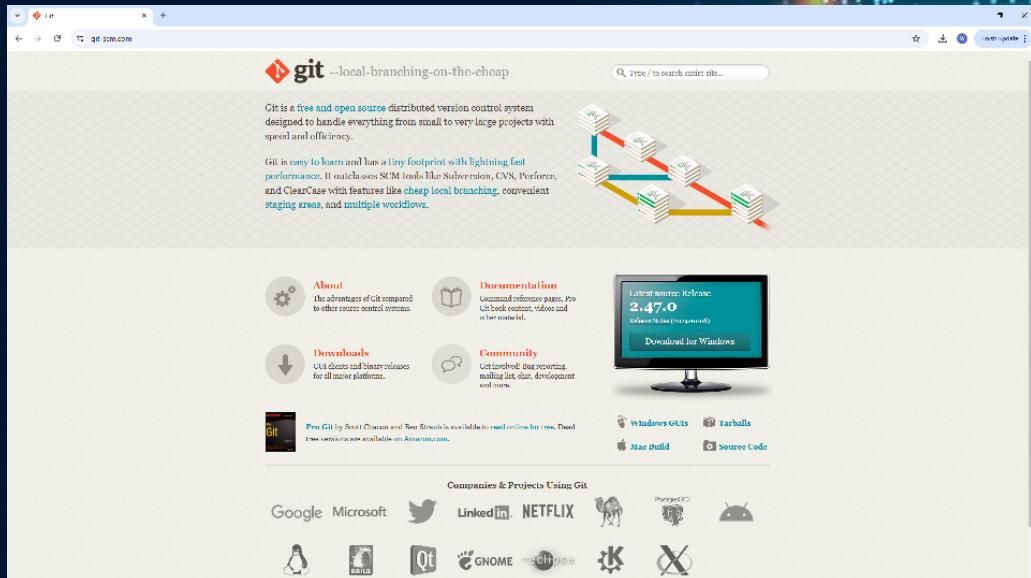
# The Essential Git Workflow

- Git flow is a lightweight, branch-based workflow.
- The Git flow is useful for everyone, not just developers.
- For example, at Git, we use Git flow for our site policy, documentation, and roadmap.

# 4. Local Git Configuration

# Working locally in Git

Working locally with Git involves setting up and managing a Git repository on your computer to handle version control for your projects. This process allows you to develop, test, and commit changes offline, syncing with remote repositories only when needed.



# Checking your Git version

- Once open, run this command:  
`git --version`



```
Command Prompt
Microsoft Windows [Version 10.0.19043.1237]
(c) Microsoft Corporation. All rights reserved.

C:\Users\marsh>git --version
```

```
Command Prompt
Microsoft Windows [Version 10.0.19043.1237]
(c) Microsoft Corporation. All rights reserved.

C:\Users\marsh>git --version
git version 2.33.0.windows.2

C:\Users\marsh>
```

# Git Configuration Levels

- In Git different layers that can be configured. The layers are:

SYSTEM: This layer is system-wide and found in  
`/etc/gitconfig`

GLOBAL: This layer is global for the user and found in  
`~/.gitconfig`

LOCAL: This layer is local to the current repository and found in  
`.git/config`

# Viewing your configurations

- You can use:  
`git config --list`
- or look at your `~/.gitconfig` file. The local configuration will be in your repository's `.git/config` file.

Use:

`git config --list --show-origin`

to see where that setting is defined (global, user, repo, etc...)

# Configuring your username and email

```
# Create a project specific config, you have to execute  
this under the project's directory.  
$ git config user.name "John Doe"
```

```
$ git config --global user.name "John Doe"  
$ git config --global user.email johndoe@example.com
```

# Working Locally

# Git Fundamentals - Working Locally

- In this lesson, we will dive deep into some of the fundamentals of Git.
- It's essential to understand well how Git thinks about files.
- Its way of tracking the history of commits, and all the basic commands that we need to master, in order to become proficient.

# Git objects

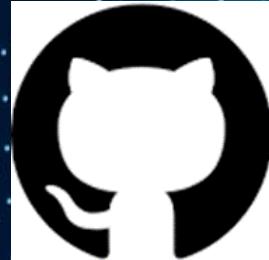
- Let's create a new repository to refresh our memory and then start learning a little bit more about Git.
- In this example, we use Git to track our shopping list before going to the grocery; so, create a new grocery folder, and then initialize a new Git repository:

```
[1] ~  
$ mkdir grocery  
  
[2] ~  
$ cd grocery/  
  
[3] ~/grocery  
$ git init  
Initialized empty Git repository in C:/Users/san/Google Drive/PortableGit/home/grocery/.git/
```

# Git objects

- As we have already seen before, the result of the git init command is the creation of a .git folder, where Git stores all the files it needs to manage our repository:

```
[4] ~/grocery (master)
$ ll
total 8
drwxr-xr-x 1 san 1049089 0 Aug 17 11:11 .
drwxr-xr-x 1 san 1049089 0 Aug 17 11:11 ..
drwxr-xr-x 1 san 1049089 0 Aug 17 11:11 .git/
```



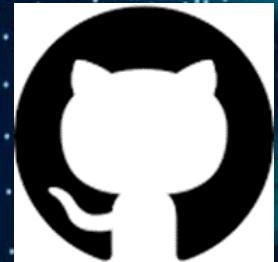
# Git objects

- Go on and create a new README.md file to remember the purpose of this repository:

```
[5] ~/grocery (master)  
$ echo "My shopping list repository" > README.md
```

- Then add a banana to the shopping list:

```
[6] ~/grocery (master)  
$ echo "banana" > shoppingList.txt
```



# Git objects

- At this point, as you already know, before doing a commit, we have to add files to the staging area; add both the files using the shortcut git add :

```
[7] ~/grocery (master)  
$ git add .
```

- With this trick (the dot after the git add command), you can add all the new or modified files in one shot.

# Git objects

- So, let's change these settings and amend our commit:

```
[8] ~/grocery (master)
$ git commit -m "Add a banana to the shopping list"
[master (root-commit) c7a0883] Add a banana to the shopping list
Committer: Santacroce Ferdinando <san@intre.it>

Your name and email address were configured automatically based on your username and hostname.
Please check that they are accurate.

You can suppress this message by setting them explicitly:
git config --global user.name "Your Name"
git config --global user.email you@example.com
After doing this, you may fix the identity used for this commit with:
git commit --amend --reset-author

2 files changed, 2 insertions(+)
create mode 100644 README.md
create mode 100644 shoppingList.txt
```

# Git objects

- For the purpose of this exercise, please leave the message as it is, press Esc, and then input the :wq (or :x) command and press Enter to save and exit:

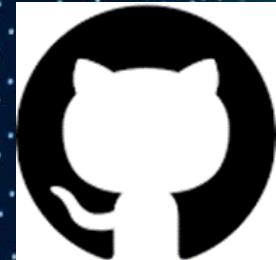
```
[9] ~/grocery (master)
```

```
$ git config user.name "Ferdinando Santacroce"
```

```
[10] ~/grocery (master)
```

```
$ git config user.email
```

```
ferdinando.santacroce@gmail.com
```



# Git objects

- Now it's time to start investigating commits.
- To verify the commit we have just created, we can use the git log command:

```
[12] ~/grocery (master)
```

```
$ git log
```

```
commit a57d783905e6a35032d9b0583f052fb42d5a1308
```

```
Author: Ferdinando Santacroce <ferdinando.santacroce@gmail.com>
```

```
Date: Thu Aug 17 13:51:33 2017 +0200
```

Add a banana to the shopping list

# Git objects

- Just under the author and date, after a blank line, we can see the message we attached to the commit we made; even the message is part of the commit itself but there's something more under the hood; let's try to use the git log command with the --format=fuller option:

```
[13] ~/grocery (master)
$ git log --format=fuller
commit a57d783905e6a35032d9b0583f052fb42d5a1308
Author: Ferdinando Santacroce <ferdinando.santacroce@gmail.com>
AuthorDate: Thu Aug 17 13:51:33 2017 +0200
Commit: Ferdinando Santacroce <ferdinando.santacroce@gmail.com>
CommitDate: Thu Aug 17 13:51:33 2017 +0200
```

Add a banana to the shopping list

# Git objects

- We analyzed a commit, and the information supplied by a simple git log; but we are not yet satisfied, so go deeper and see what's inside.
- Using the git log command again, we can enable x-ray vision using the --format=raw option:

```
[14] ~/grocery (master)
$ git log --format=raw
commit a57d783905e6a35032d9b0583f052fb42d5a1308
tree a31c31cb8d7cc16eeae1d2c15e61ed7382cebf40
author Ferdinando Santacroce <ferdinando.santacroce@gmail.com> 1502970693 +0200
committer Ferdinando Santacroce <ferdinando.santacroce@gmail.com> 1502970693 +0200

Add a banana to the shopping list
```

# Git objects

- Back on topic; type the command, specifying the first characters of the commit's hash (a57d7 in my case):

```
[15] ~/grocery (master)
$ git cat-file -p a57d7
tree a31c31cb8d7cc16eeae1d2c15e61ed7382cebf40
author Ferdinando Santacroce <ferdinando.santacroce@gmail.com> 1502970693 +0200
committer Ferdinando Santacroce <ferdinando.santacroce@gmail.com> 1502970693 +0200

Add a banana to the shopping list
```

# Git objects

## **Porcelain commands and plumbing commands**

- Git, as we know, has a myriad of commands, some of which are practically never used by the average user; as by example, the previous git cat-file.
- These commands are called plumbing commands, while those we have already learned about, such as git add, git commit, and so on, are among the so-called porcelain commands.

# Git objects

- Here, for convenience, there is the output of the git cat-file -p command typed before:

```
[15] ~/grocery (master)
$ git cat-file -p a57d7
tree a31c31cb8d7cc16eeae1d2c15e61ed7382cebf40
author Ferdinando Santacroce <ferdinando.santacroce@gmail.com> 1502970693 +0200
committer Ferdinando Santacroce <ferdinando.santacroce@gmail.com> 1502970693 +0200
```

# Trees

- The tree is a container for blobs and other trees.
- The easiest way to understand how it works is to think about folders in your operating system, which also collect files and other subfolders inside them.
- Let's try to see what this additional Git object holds, using again the git cat-file -p command:

```
[16] ~/grocery (master)
$ git cat-file -p a31c3
100644 blob 907b75b54b7c70713a79cc6b7b172fb131d3027d README.md
100644 blob 637a09b86af61897fb72f26fb874f2ae726db82 shoppingList.txt
```

# Blobs

```
[17] ~/grocery (master)
$ git cat-file -p 637a0
banana
```

- Wow! Its content is exactly the content of our shoppingFile.txt file.
- To confirm, we can use the cat command, which on \*nix systems allows you to see the contents of a file:

```
[18] ~/grocery (master)
$ cat shoppingList.txt
banana
```

- As you can see, the result is the same.

# Blobs

- Blobs are binary files, nothing more and nothing less.
- These byte sequences, which cannot be interpreted with the naked eye, retain inside information belonging to any file, whether binary or textual, images, source code, archives, and so on.
- Everything is compressed and transformed into a blob before archiving it into a Git repository.

# Blobs

- Let's try to understand it better with an example.
- Open a shell and try to play a bit with another plumbing command, git hash-object:

```
[19] ~/grocery (master)
$ echo "banana" | git hash-object --stdin
637a09b86af61897fb72f26fb874f2ae726db82
```

# Even deeper - the Git storage object model

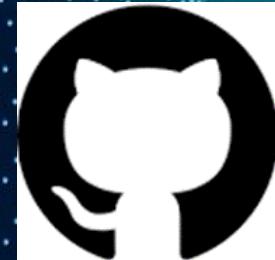
- Do you remember the .git folder? Let's put our nose inside it:

```
[20] ~/grocery (master)
$ ll .git/
total 13
drwxr-xr-x 1 san 1049089 0 Aug 18 17:22 .
drwxr-xr-x 1 san 1049089 0 Aug 18 17:15 ..
-rw-r--r-- 1 san 1049089 294 Aug 17 13:52 COMMIT_EDITMSG
-rw-r--r-- 1 san 1049089 208 Aug 17 13:51 config
-rw-r--r-- 1 san 1049089 73 Aug 17 11:11 description
-rw-r--r-- 1 san 1049089 23 Aug 17 11:11 HEAD
drwxr-xr-x 1 san 1049089 0 Aug 18 17:15 hooks/
-rw-r--r-- 1 san 1049089 217 Aug 18 17:22 index
drwxr-xr-x 1 san 1049089 0 Aug 18 17:15 info/
drwxr-xr-x 1 san 1049089 0 Aug 18 17:15 logs/
drwxr-xr-x 1 san 1049089 0 Aug 18 17:15 objects/
drwxr-xr-x 1 san 1049089 0 Aug 18 17:15 refs/
```

# Even deeper - the Git storage object model

- Within it, there is an objects subfolder; let's take a look:

```
[21] ~/grocery (master)
$ ll .git/objects/
total 4
drwxr-xr-x 1 san 1049089 0 Aug 18 17:15 .
drwxr-xr-x 1 san 1049089 0 Aug 18 17:22 ..
drwxr-xr-x 1 san 1049089 0 Aug 18 17:15 63/
drwxr-xr-x 1 san 1049089 0 Aug 18 17:15 90/
drwxr-xr-x 1 san 1049089 0 Aug 18 17:15 a3/
drwxr-xr-x 1 san 1049089 0 Aug 18 17:15 a5/
drwxr-xr-x 1 san 1049089 0 Aug 18 17:15 c7/
drwxr-xr-x 1 san 1049089 0 Aug 17 11:11 info/
drwxr-xr-x 1 san 1049089 0 Aug 18 17:12 pack/
```



# Even deeper - the Git storage object model

- Other than info and pack folders, which are not interesting for us right now, as you can see there are some other folders with a strange two-character name; let's go inside the 63 folder:

```
[22] ~/grocery (master)
$ ll .git/objects/63/
total 1
drwxr-xr-x 1 san 1049089 0 Aug 18 17:15 ../
drwxr-xr-x 1 san 1049089 0 Aug 18 17:15 ./
-r--r--r-- 1 san 1049089 20 Aug 17 13:34 7a09b86af61897fb72f26fb874f2ae726db82
```

# Even deeper - the Git storage object model

- To become aware of this, we need a new commit. So, let's now proceed modifying the shoppingList.txt file:

```
[23] ~/grocery (master)
$ echo "apple" >> shoppingList.txt
```

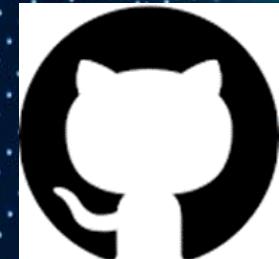
```
[24] ~/grocery (master)
$ git add shoppingList.txt
```

```
[25] ~/grocery (master)
$ git commit -m "Add an apple"
[master e4a5e7b] Add an apple
 1 file changed, 1 insertion(+)
```

# Even deeper - the Git storage object model

- Use the git log command to check the new commit; the --oneline option allows us to see the log in a more compact way:

```
[26] ~/grocery (master)
$ git log --oneline
e4a5e7b Add an apple
a57d783 Add a banana to the shopping list
```



# Even deeper - the Git storage object model

- Okay, we have a new commit, with its hash.
- Time to see what's inside it:

```
[27] ~/grocery (master)
$ git cat-file -p e4a5e7b
tree 4c931e9fd8ca4581ddd5de9efd45daf0e5c300a0
parent a57d783905e6a35032d9b0583f052fb42d5a1308
author Ferdinando Santacroce <ferdinando.santacroce@gmail.com> 1503586854 +0200
committer Ferdinando Santacroce <ferdinando.santacroce@gmail.com> 1503586854 +0200
```

Add an apple

# Git doesn't use deltas

- In Git even if you change only a char in a big text file, it always stores a new version of the file.
- Git doesn't do deltas (at least not in this case), and every commit is actually a snapshot of the entire repository.
- At this point, people usually exclaim:

"Gosh, Git waste a large amount of disk space in vain!"

- Well, this is simply untrue.

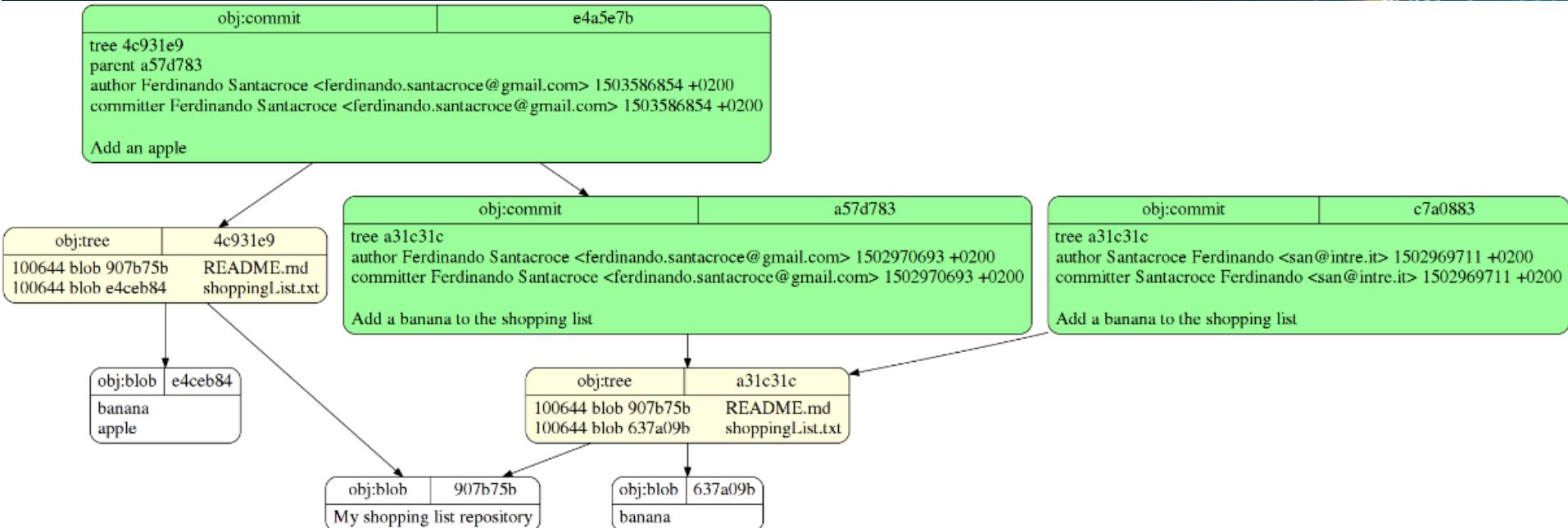
# Git doesn't use deltas

- Furthermore, Git has a clever way to deal with files; let's take a look again at the last commit:

```
[28] ~/grocery (master)
$ git cat-file -p e4a5e7b
tree 4c931e9fd8ca4581ddd5de9efd45daf0e5c300a0
parent a57d783905e6a35032d9b0583f052fb42d5a1308
author Ferdinando Santacroce <ferdinando.santacroce@gmail.com> 1503586854 +0200
committer Ferdinando Santacroce <ferdinando.santacroce@gmail.com> 1503586854 +0200
```

Add an apple

# Wrapping up



# Wrapping up

- In this graphic representation of previous slide, you will find a detailed diagram that represents the current structure of the newly created repository.
- You can see trees (yellow), blobs (white), commits (green), and all relationships between them, represented by oriented arrows.

# Wrapping up

- Commit
  - |
    - └ Tree (root directory)
      - ┌─ Blob (file1.txt content)
      - ┌─ Blob (file2.txt content)
      - └ Tree (subdir)
        - └─ Blob (file3.txt content)

# Git Aliases

# Git aliases

## Shortcuts to common commands

- One thing you may find useful is to shorten common commands such as git checkout and so on; therefore, useful aliases can include the following:

```
[1] ~/grocery (master)
$ git config --global alias.co checkout
[2] ~/grocery (master)
$ git config --global alias.br branch
[3] ~/grocery (master)
$ git config --global alias.ci commit
[4] ~/grocery (master)
$ git config --global alias.st status
```

# Git aliases

- Another common practice is to shorten a command, adding one or more options you use all the time; for example, set a git cm <commit message> command shortcut to the alias git commit -m <commit message>:

```
[5] ~/grocery (master)
```

```
$ git config --global alias.cm "commit -m"
```

```
[6] ~/grocery (master)
```

```
$ git cm "My commit message"
```

```
On branch master
```

```
nothing to commit, working tree clean
```

# Git aliases

- The classic example is the git unstage alias:  
[1] ~/grocery (master)  
\$ git config --global alias.unstage 'reset HEAD --'
- With this alias, you can remove a file from the index in a more meaningful way, compared to the equivalent git reset HEAD -- <file> syntax:

[2] ~/grocery (master)  
\$ git unstage myFile.txt

# Git aliases

- Now behaves the same as:

```
[3] ~/grocery (master)
```

```
$ git reset HEAD -- myFile.txt
```

**git undo**

- Want a fast way to revert the last ongoing commit? Create a git undo alias:

```
[1] ~/grocery (master)
```

```
$ git config --global alias.undo 'reset --soft HEAD~1'
```

# Git aliases

## git last

- A git last alias is useful to read about your last commit:

[1] ~/grocery (master)

```
$ git config --global alias.last 'log -1 HEAD'
```

[2] ~/grocery (master)

```
$ git last
```

commit b25ffa60f44f6fc50e81181cab87ed3dbf3b172c

Author: Ernesto Lee <socrates73@gmail.com>

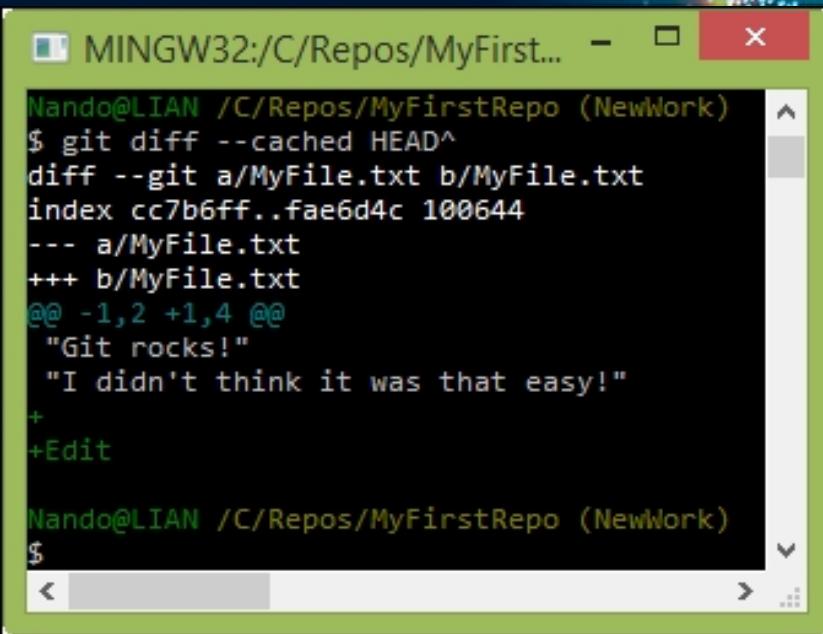
Date: Thu Jul 27 15:12:48 2017 +0200Add an apricot

# Git aliases

## git difflast

- With the git difflast alias, you can see a diff against your last commit:

```
$ git config --global alias.difflast  
'diff --cached HEAD^'
```



The screenshot shows a terminal window titled "MINGW32:/C/Repos/MyFirst...". The command entered is \$ git diff --cached HEAD^, which displays a diff between the current state and the previous commit. The output shows a file named MyFile.txt with changes made in the second commit. The terminal window has a green border and a red header bar.

```
Nando@LIAN /C/Repos/MyFirstRepo (NewWork)  
$ git diff --cached HEAD^  
diff --git a/MyFile.txt b/MyFile.txt  
index cc7b6ff..fae6d4c 100644  
--- a/MyFile.txt  
+++ b/MyFile.txt  
@@ -1,2 +1,4 @@  
 "Git rocks!"  
 "I didn't think it was that easy!"  
+  
+Edit  
  
Nando@LIAN /C/Repos/MyFirstRepo (NewWork)  
$
```

# Getting ready

- In this example, we will use the jgit repository, Navigating Git, with the master branch pointing at b14a93971837610156e815ae2eee3baaa5b7a44b. Navigating Git, or clone the repository again, as follows:

```
$ git clone https://git.eclipse.org/r/jgit/jgit  
$ cd jgit  
$ git checkout master && git reset --hard b14a939
```

# How to do it...

```
$ git config --global alias.co checkout  
$ git config --global alias.br branch  
$ git config --global alias.ci commit  
$ git config --global alias.st status
```

- Now, try to run git st in the jgit repository as follows:

```
$ git st  
# On branch master  
nothing to commit, working directory clean
```

# How to do it...

- The alias method is also good to create the Git commands you think are missing in Git.
- One of the common Git aliases is `unstage`, which is used to move a file out of the staging area, as shown in the following command:

```
$ git config --global alias.unstage 'reset HEAD --'
```

# How to do it...

- Try to edit the README.md file in the root of the jgit repository and add it in the root.
- Now, git status/git st should display something like the following:

```
$ git st
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified: README.md
```



# How to do it...

```
$ git unstage README.md
Unstaged changes after reset:
M      README.md

$ git st
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   README.md
#
no changes added to commit (use "git add" and/or "git commit -a")
```

# How to do it...

- Let's say you want the number of lines added and deleted for each file in the commit displayed along with some common commit data.
- For this, we can create the following alias so we don't have to type everything each time:

```
$ git config --global alias.ll "log --  
pretty=format:\"%C(yellow)%h%Cred%d %Creset%s  
%Cgreen(%cr) %C(bold blue)<%an>%Creset\" --numstat"
```

# How to do it...

```
$ git ll  
b14a939 (HEAD, master) Prepare 3.3.0-SNAPSHOT builds (8 days ago) <Matthias Sohn>  
6      6      org.eclipse.jgit.ant.test/META-INF/MANIFEST.MF  
1      1      org.eclipse.jgit.ant.test/pom.xml  
3      3      org.eclipse.jgit.ant/META-INF/MANIFEST.MF  
1      1      org.eclipse.jgit.ant/pom.xml  
4      4      org.eclipse.jgit.archive/META-INF/MANIFEST.MF  
2      2      org.eclipse.jgit.archive/META-INF/SOURCE-MANIFEST.MF  
1      1      org.eclipse.jgit.archive/pom.xml  
6      6      org.eclipse.jgit.console/META-INF/MANIFEST.MF  
1      1      org.eclipse.jgit.console/pom.xml  
12     12     org.eclipse.jgit.http.server/META-INF/MANIFEST.MF  
...  
...
```

# Git aliases

## Advanced aliases with external commands

- If you want the alias to run external shell commands, instead of a Git sub-command, you have to prefix the alias with a !:

```
$ git config --global alias.echo !echo
```

- Suppose you are annoyed by the canonical git add <file> plus git commit <file> sequence of commands, and you want to do it in a single shot; you can call the git command twice in sequence by creating this alias:

```
$ git config --global alias.cm '!git add -A && git commit -m'
```

# Git aliases

## Removing an alias

- Removing an alias is quite easy; you have to use the `--unset` option, specifying the alias to remove.
- For example, if you want to remove the `cm` alias, you have to run:

```
$ git config --global --unset alias.cm
```

# Git aliases

## Aliasing the git command itself

- I've already said I'm a bad typist; if you are too, you can alias the git command itself (using the default alias command in Bash):

```
$ alias gti='git'
```

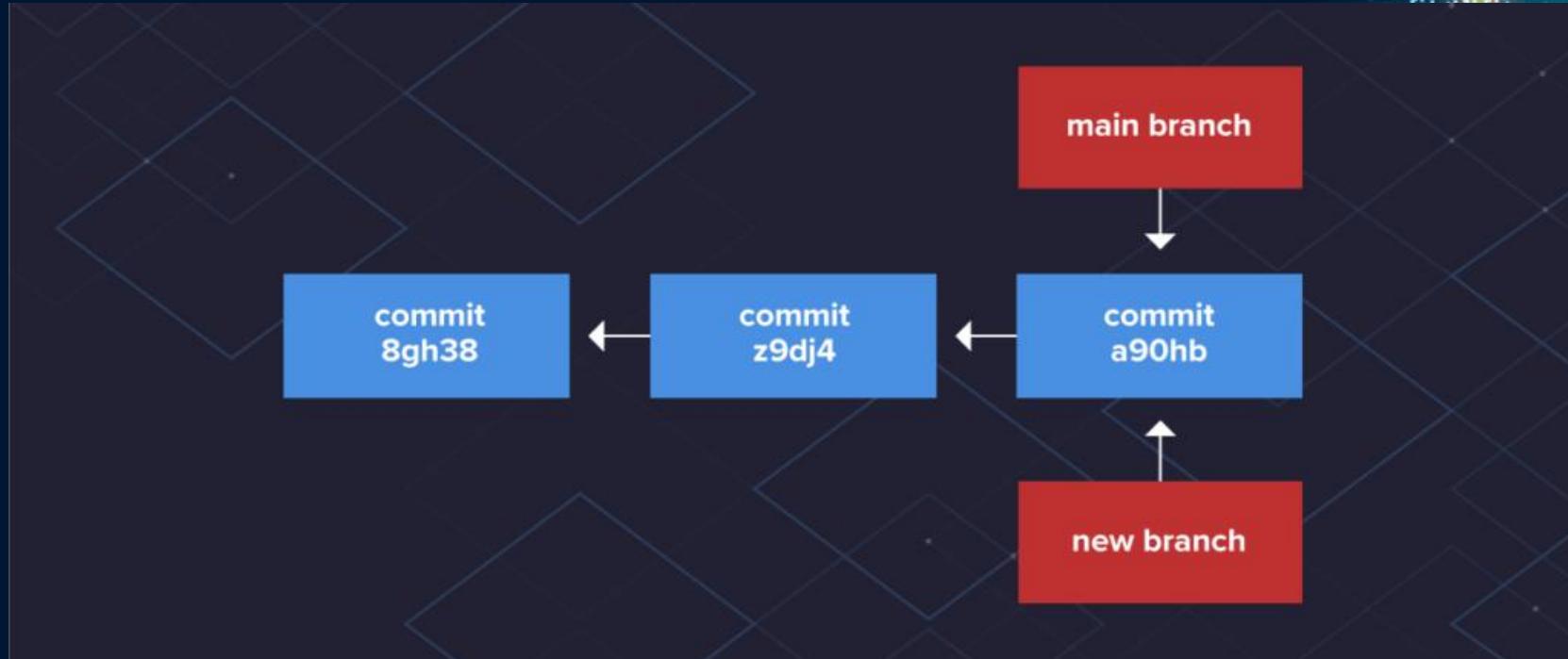
# 3. Branching with Git



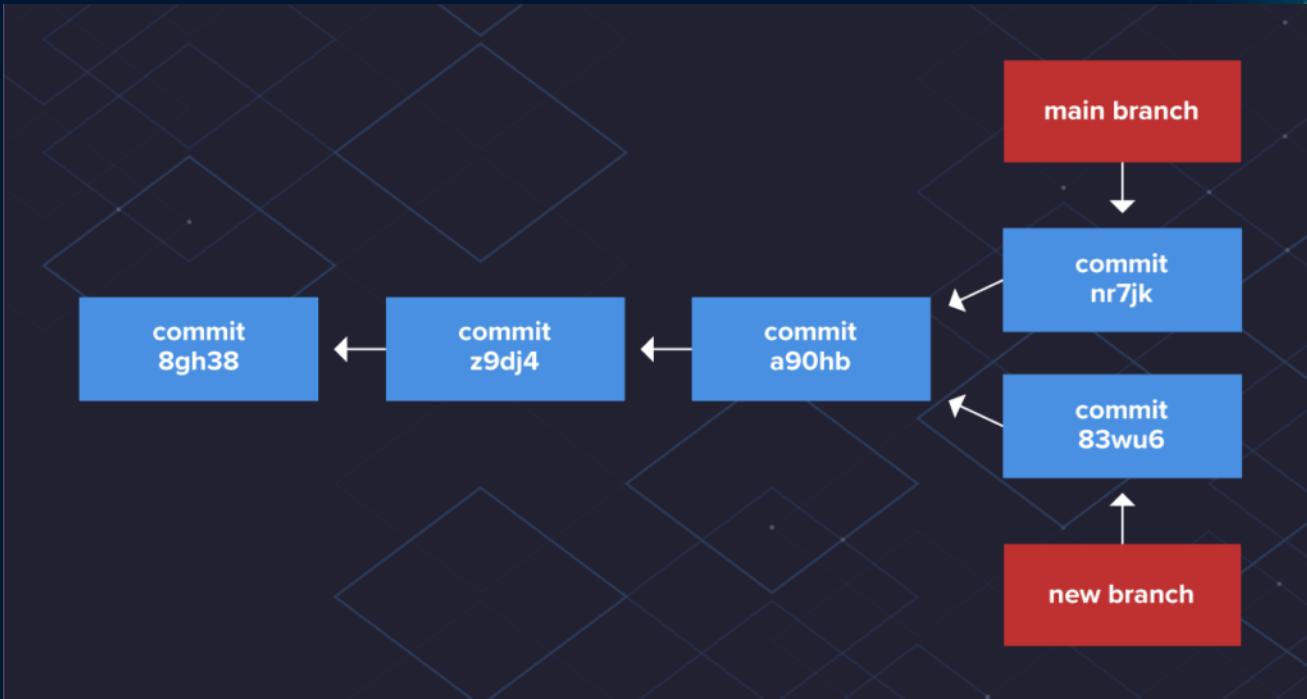
# Git branching

- Branches are not just exclusive to Git. However, in this course we focus on Git due to the many advantages this model of branching offers.
- Consequently, before we delve into the various branching strategies out there, including Git branching strategies.
- Put simply, Git and other version control tools allow developers to track, manage and organize their code.

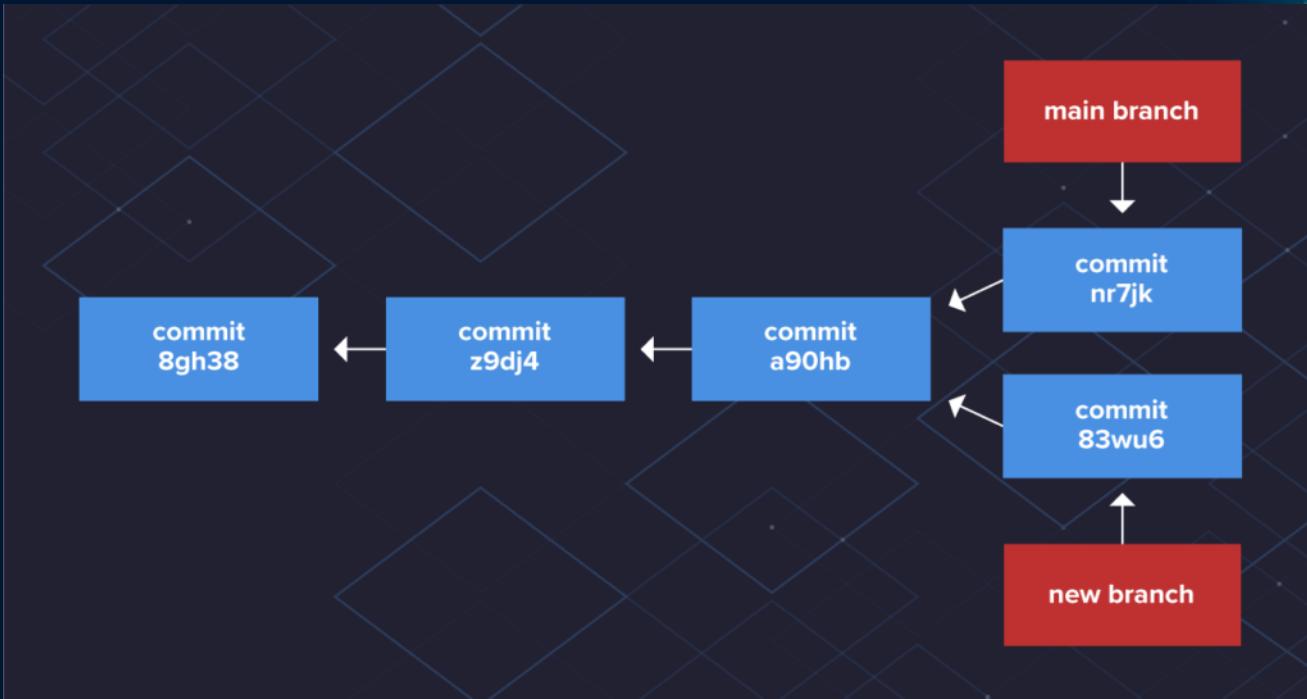
# Git Branching



# Git Branching



# Git Branching



# Branch Naming Strategies

- Branch names can be anything you'd like.
- Your organization or project may have standards outlined for branch naming.
- For example, naming the branch based on the person responsible for working on the branch and a description or work item:
  - username/description
  - username/workitem

# Branch Naming Strategies

- You can name a branch to indicate the branch's function, like a feature, bug fix, or hotfix:
  - bugfix/description
  - feature/feature-name
  - hotfix/description

# COMPLETE

## Activity: Creating a Branch with Git

# Branching with Git

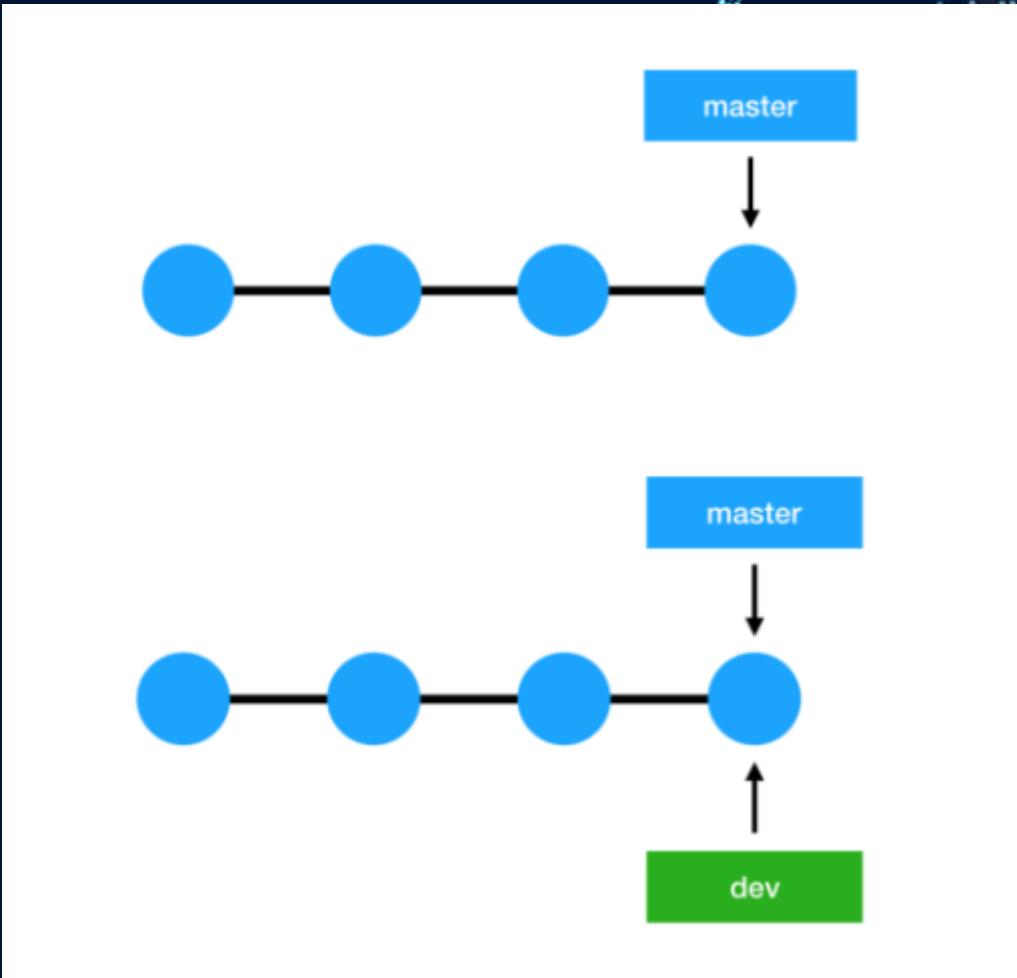
# Create a branch

- Create a branch in your repository, A short, descriptive branch name enables your collaborators to see ongoing work at a glance.
- For example, increase-test-timeout or add-code-of-conduct
- By creating a branch, you create a space to work without affecting the default branch, Additionally, you give collaborators a chance to review your work.

# Make changes

- On your branch, make any desired changes to the repository.
- Your branch is a safe place to make changes, If you make a mistake, you can revert your changes or push additional changes to fix the mistake.
- Your changes will not end up on the default branch until you merge your branch.
- Commit and push your changes to your branch.
- Give each commit a descriptive message to help you and future contributors understand what changes the commit contains.

# Introduction



# Introduction

- If you are developing a small application in a big corporation as a developer.
- Or you are trying to wrap your head around an open source project from GitHub, you have already been using branches with Git.

# Managing your local branches

- Suppose you are just having your local Git repository, and you have no intentions at the moment to share the code with others.
- However, you can easily share this knowledge while working with a repository with one or more remotes

# Getting ready

- Use the following command to clone the jgit repository to match:

```
$ git clone https://git.eclipse.org/r/jgit/jgit  
$ cd jgit
```

# How to do it...

- Whenever you start working on a bug fix or a new feature in your project, you should create a branch.
- You can do so using the following code:

```
$ git branch newBugFix  
$ git branch  
* master  
newBugFix
```

# How to do it...

- The newBugFix branch points to the current HEAD I was on at the time of the creation.
- You can see the HEAD with git log -1:

```
$ git log -1 newBugFix --format=format:%H  
25fe20b2dbb20cac8aa43c5ad64494ef8ea64ffc
```

- If you want to add a description to the branch, you can do it with the --edit-description option for the git branch command:

```
$ git branch --edit-description newBugFix
```

# How to do it...

- The previous command will open an editor where you can type in a description:

## Refactoring the Hydro controller

The hydro controller code is currently horrible needs to be refactored.

- Close the editor and the message will be saved.

# How it works...

- Git stores the information in the local git config file; this also means that you cannot push this information to a remote repository.
- To retrieve the description for the branch, you can use the --get flag for the git config command:

```
$ git config --get branch.newBugFix.description  
Refactoring the Hydro controller  
The hydro controller code is currently horrible needs to be  
refactored.
```

# How it works...

- The branch information is stored as a file in .git/refs/heads/newBugFix:

```
$ cat .git/refs/heads/newBugFix  
25fe20b2dbb20cac8aa43c5ad64494ef8ea64ffc
```

- Note that it is the same commit hash we retrieved with the git log command

# Git references

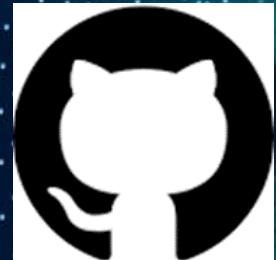
## **It's all about labels**

- In Git, a branch is nothing more than a label, a mobile label placed on a commit.
- In fact, every leaf on a Git branch has to be labeled with a meaningful name to allow us to reach it and then move around, go back, merge, rebase, or discard some commits when needed.

# Git references

- Let's start exploring this topic by checking the current status of our grocery repository; we do it using the well-known git log command, this time adding some new options:

```
[1] ~/grocery (master)
$ git log --oneline --graph --decorate
* e4a5e7b (HEAD -> master) Add an apple
* a57d783 Add a banana to the shopping list
```



# Git references

- We'll now do a new commit and see what happens:

```
[2] ~/grocery (master)
$ echo "orange" >> shoppingList.txt
```

```
[3] ~/grocery (master)
$ git commit -am "Add an orange"
[master 0e8b5cf] Add an orange
 1 file changed, 1 insertion(+)
```

# Git references

- Okay, go on now and take a look at the current repository situation:

```
[4] ~/grocery (master)
```

```
$ git log --oneline --graph --decorate
```

```
* 0e8b5cf (HEAD -> master) Add an orange
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```



# Git references

- But how will Git handle this feature? Let's go back to putting the nose again in the .git folder:

```
[5] ~/grocery (master)
$ ll .git/
total 21
drwxr-xr-x 1 san 1049089  0 Aug 25 11:20 .
drwxr-xr-x 1 san 1049089  0 Aug 25 11:19 ../
-rw-r--r-- 1 san 1049089 14 Aug 25 11:20 COMMIT_EDITMSG
-rw-r--r-- 1 san 1049089 208 Aug 17 13:51 config
-rw-r--r-- 1 san 1049089  73 Aug 17 11:11 description
-rw-r--r-- 1 san 1049089  23 Aug 17 11:11 HEAD
drwxr-xr-x 1 san 1049089  0 Aug 18 17:15 hooks/
-rw-r--r-- 1 san 1049089 217 Aug 25 11:20 index
drwxr-xr-x 1 san 1049089  0 Aug 18 17:15 info/
drwxr-xr-x 1 san 1049089  0 Aug 18 17:15 logs/
drwxr-xr-x 1 san 1049089  0 Aug 25 11:20 objects/
drwxr-xr-x 1 san 1049089  0 Aug 18 17:15 refs/
```

# Git references

- There's a refs folder: let's take a look inside:

```
[6] ~/grocery (master)
```

```
$ ll .git/refs/
```

```
total 4
```

```
drwxr-xr-x 1 san 1049089 0 Aug 18 17:15 ./
```

```
drwxr-xr-x 1 san 1049089 0 Aug 25 11:20 ../
```

```
drwxr-xr-x 1 san 1049089 0 Aug 25 11:20 heads/
```

```
drwxr-xr-x 1 san 1049089 0 Aug 17 11:11 tags/
```

# Git references

- Now go to heads:

```
[7] ~/grocery (master)
$ ll .git/refs/heads/
total 1
drwxr-xr-x 1 san 1049089 0 Aug 25 11:20 .
drwxr-xr-x 1 san 1049089 0 Aug 18 17:15 ../
-rw-r--r-- 1 san 1049089 41 Aug 25 11:20 master
```

- There's a master file inside! Let's see what's the content:

```
[8] ~/grocery (master)
$ cat .git/refs/heads/master
0e8b5cf1c1b44110dd36dea5ce0ae29ce22ad4b8
```

# Git references

## Creating a new branch

- Now that we have warmed up, the fun begins.
- Let's see what happens when you ask Git to create a new branch.
- Since we are going to serve a delicious fruit salad, it's time to set a branch apart for a berries-flavored variant recipe:

```
[9] ~/grocery (master)
$ git branch berries
```

# Git references

- So, git log again:

```
[10] ~/grocery (master)
$ git log --oneline --graph --decorate
* 0e8b5cf (HEAD -> master, berries) Add an orange
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```

# Git references

- Anyway, at the moment we continue to be located in the master branch; in fact, as you can see in the shell output prompt, it continues to appear (master) between the round parenthesis:

```
[10] ~/grocery (master)
```

- How can I switch branch? By using the git checkout command:

```
[11] ~/grocery (master)  
$ git checkout berries  
Switched to branch 'berries'
```



# Git references

- Do a git log to see:

[12] ~/grocery (berries)

```
$ git log --oneline --graph --decorate
```

```
* 0e8b5cf (HEAD -> berries, master) Add an orange  
* e4a5e7b Add an apple  
* a57d783 Add a banana to the shopping list
```



# Git references

## HEAD, or you are here

- During previous exercises we continued to see that HEAD thing while using git log, and now it's time to investigate a little bit.
- First of all, what is HEAD? As branches are, HEAD is a reference.
- It represents a pointer to the place on where we are right now, nothing more, nothing less.
- In practice instead, it is just another plain text file:

```
[13] ~/grocery (berries)
```

```
$ cat .git/HEAD
```

```
ref: refs/heads/berries
```

# Git references

- Add a blackberry to the shopping list:  
[14] ~/grocery (berries)  
\$ echo "blackberry" >> shoppingList.txt
- Then perform a commit:  
[15] ~/grocery (berries)  
\$ git commit -am "Add a blackberry"  
[berries ef6c382] Add a blackberry  
1 file changed, 1 insertion(+)



# Git references

- Take a look on what happened with the usual git log command:

```
[16] ~/grocery (berries)
```

```
$ git log --oneline --graph --decorate
```

```
* ef6c382 (HEAD -> berries) Add a blackberry
```

```
* 0e8b5cf (master) Add an orange
```

```
* e4a5e7b Add an apple
```

```
* a57d783 Add a banana to the shopping list
```

- Okay, so now our shoppingList.txt file appears to contain these text lines:

```
[17] ~/grocery (berries)
```

```
$ cat shoppingList.txt
```

```
banana
```

```
apple
```

```
orange
```

```
blackberry
```

# Git references

- Check out the master branch:

[18] ~/grocery (berries)

```
$ git checkout master
```

Switched to branch 'master'

- Look at the shoppingFile.txt content:

[19] ~/grocery (master)

```
$ cat shoppingList.txt
```

banana

apple

orange



# Git references

- We actually moved back to where we were before adding the blackberry; as it is being added in the berries branch, here in the master branch it does not exist: sounds good, doesn't it?
- Even the HEAD file has been updated accordingly:

```
[20] ~/grocery (master)
$ cat .git/HEAD
ref: refs/heads/master
```

# Git references

- Go back to the repository now; let's do the usual git log:

```
[21] ~/grocery (master)
```

```
$ git log --oneline --graph --decorate
```

```
* 0e8b5cf (HEAD -> master) Add an orange
```

```
* e4a5e7b Add an apple
```

```
* a57d783 Add a banana to the shopping list
```



# Git references

- Uh-oh: where is the berries branch? Don't worry: git log usually displays only the branch you are on, and the commit that belongs to it.
- To see all branches, you only need to add the --all option:  
[22] ~/grocery (master)

```
$ git log --oneline --graph --decorate --all
* ef6c382 (berries) Add a blackberry
* 0e8b5cf (HEAD -> master) Add an orange
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```

# Git references

- First, check out the berries branch:

```
[23] ~/grocery (master)
```

```
$ git checkout -
```

```
Switched to branch 'berries'
```

- Now a new command, git reset (please don't care about the --hard option for now):

```
[24] ~/grocery (berries)
```

```
$ git reset --hard master
```

```
HEAD is now at 0e8b5cf Add an orange
```

# Git references

- The git reset actually moves a branch from the current position to a new one; here we said Git to move the current berries branch to where master is, and the result is that now we have all the two branches pointing to the same commit:

```
[25] ~/grocery (berries)
```

```
$ git log --oneline --graph --decorate --all
```

```
* 0e8b5cf (HEAD -> berries, master) Add an orange
```

```
* e4a5e7b Add an apple
```

```
* a57d783 Add a banana to the shopping list
```

# Git references

- You can double-check this looking at refs files; this is the berries one:

```
[26] ~/grocery (berries)
```

```
$ cat .git/refs/heads/berries
```

```
0e8b5cf1c1b44110dd36dea5ce0ae29ce22ad4b8
```

- And this is the master one:

```
[27] ~/grocery (berries)
```

```
$ cat .git/refs/heads/master
```

```
0e8b5cf1c1b44110dd36dea5ce0ae29ce22ad4b8
```

# Git references

- Let's try another trick: we can use git reset to move the actual branch directly to a commit.
- And to make things more interesting, let's try to point the blackberry commit (if you scroll your shell window backwards, you can see its hash, which for me is ef6c382) so, git reset to the ef6c382 commit:

```
[28] ~/grocery (berries)
$ git reset --hard ef6c382
HEAD is now at ef6c382 Add a blackberry
```

# Git references

- And then do the usual git log:

[29] ~/grocery (berries)

```
$ git log --oneline --graph --decorate --all
* ef6c382 (HEAD -> berries) Add a blackberry
* 0e8b5cf (master) Add an orange
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```



# Git references

- Assume you want to add a watermelon to the shopping list, but later you realize you added it to the wrong berries branch; so, add "watermelon" to the shoppingList.txt file:

```
[30] ~/grocery (berries)
```

```
$ echo "watermelon" >> shoppingList.txt
```

- Then do the commit:

```
[31] ~/grocery (berries)
```

```
$ git commit -am "Add a watermelon"
```

```
[berries a8c6219] Add a watermelon
```

```
1 file changed, 1 insertion(+)
```

# Git references

- And do a git log to check the result:

```
[32] ~/grocery (berries)
```

```
$ git log --oneline --graph --decorate --all
```

```
* a8c6219 (HEAD -> berries) Add a watermelon
* ef6c382 Add a blackberry
* 0e8b5cf (master) Add an orange
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```

# Git references

- Now our aim here is: have a new melons branch, which the watermelon commit have to belong to, then set the house in order and move the berries branch back to the blackberry commit.
- To keep the watermelon commit, first create a melon branch that points to it with the well-known git branch command:

```
[33] ~/grocery (berries)
$ git branch melons
```

# Git references

- Let's check:

[34] ~/grocery (berries)

```
$ git log --oneline --graph --decorate --all
```

```
* a8c6219 (HEAD -> berries, melons) Add a watermelon
* ef6c382 Add a blackberry
* 0e8b5cf (master) Add an orange
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```

# Git references

- So, let's step back our berries branch using caret; do a `git reset --hard HEAD^`:

```
[35] ~/grocery (berries)
```

```
$ git reset --hard HEAD^
```

```
HEAD is now at ef6c382 Add a blackberry
```

- Let's see the result:

```
[36] ~/grocery (berries)
$ git log --oneline --graph --decorate --all
* a8c6219 (melons) Add a watermelon
* ef6c382 (HEAD -> berries) Add a blackberry
* 0e8b5cf (master) Add an orange
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```

# Git references

- Just to remark concepts, let's take a look at the shoppingList.txt file here in the berries branch:

```
[37] ~/grocery (berries)
```

```
$ cat shoppingList.txt
```

```
banana
```

```
apple
```

```
orange
```

```
blackberry
```

- Okay, here we have blackberry, other than the other previously added fruits.
- Switch to master and check again; check out the master branch

```
[38] ~/grocery (berries)
```

```
$ git checkout master
```

```
Switched to branch 'master'
```

# Git references

- Then cat the file:

```
[39] ~/grocery (master)
```

```
$ cat shoppingList.txt
```

```
banana
```

```
apple
```

```
orange
```

- Okay, no blackberry here, but only fruits added before the berries branch creation and now a last check on the melons branch; check out the branch:

```
[40] ~/grocery (master)
```

```
$ git checkout melons
```

```
Switched to branch 'melons'
```

# Git references

- And cat the shoppingList.txt file:

```
[41] ~/grocery (melons)
```

```
$ cat shoppingList.txt
```

```
banana
```

```
apple
```

```
orange
```

```
blackberry
```

```
watermelon
```

# Git references

- For the sake of the explanation, go back to the master branch and see what happens when we check out the previous commit, moving HEAD backward; perform a git checkout HEAD^:

```
[42] ~/grocery (master)
$ git checkout HEAD^
Note: checking out 'HEAD^'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this state without impacting any
branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b <new-branch-name>

HEAD is now at e4a5e7b... Add an apple
```

# Git references

- Here, Git says we are in a detached HEAD state. Being in this state basically means that HEAD does not reference a branch, but directly a commit, the e4a5e7b one in this case; do a git log and see:

```
[43] ~/grocery ((e4a5e7b...))

$ git log --oneline --graph --decorate --all
* a8c6219 (melons) Add a watermelon
* ef6c382 (berries) Add a blackberry
* 0e8b5cf (master) Add an orange
* e4a5e7b (HEAD) Add an apple
* a57d783 Add a banana to the shopping list
```

# Git references

- First of all, in the shell prompt you can see that between rounds, that now are doubled, there is not a branch name, but the first seven characters of the commit, ((e4a5e7b...)).
- Then, HEAD is now stuck to that commit, while branches, especially the master one, remains at their own place.
- As a result, the HEAD file now contains the hash of that commit, not a ref to a branch as before:

```
[44] ~/grocery ((e4a5e7b...))
```

```
$ cat .git/HEAD
```

```
e4a5e7b3c64bee8b60e23760626e2278aa322f05
```

# Git references

- Okay, let's have some fun; modify the shoppingList.txt file, adding a bug:

```
[45] ~/grocery ((e4a5e7b...))
```

```
$ echo "bug" > shoppingList.txt
```

- Then commit this voluntary mistake:

```
[46] ~/grocery ((e4a5e7b...))
```

```
$ git commit -am "Bug eats all the fruits!"
```

```
[detached HEAD 07b1858] Bug eats all the fruits!
```

```
1 file changed, 1 insertion(+), 2 deletions(-)
```

# Git references

- Let's cat the file:

```
[47] ~/grocery ((07b1858...))
```

```
$ cat shoppingList.txt
```

```
bug
```

- Ouch, we actually erased all your shopping list files!
- What happened in the repository then?

# Git references

- Okay, so if we now check out master again, what happens? Give it a try:

```
[49] ~/grocery ((07b1858...))
$ git checkout master

Warning: you are leaving 1 commit behind, not connected to
any of your branches:

    07b1858 Bug eats all the fruits!

If you want to keep it by creating a new branch, this may be a good time to do so with:
  git branch <new-branch-name> 07b1858

Switched to branch 'master'
```

# Git references

- Let's check the situation:

```
[50] ~/grocery (master)
```

```
$ git log --oneline --graph --decorate --all
* a8c6219 (melons) Add a watermelon
* ef6c382 (berries) Add a blackberry
* 0e8b5cf (HEAD -> master) Add an orange
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```



# Git references

- Let's try it, creating a bug branch:

[51] ~/grocery (master)

\$ git branch bug 07b1858

- Let's see what happened:

```
[52] ~/grocery (master)
$ git log --oneline --graph --decorate --all
* 07b1858 (bug) Bug eats all the fruits!
| * a8c6219 (melons) Add a watermelon
| * ef6c382 (berries) Add a blackberry
| * 0e8b5cf (HEAD -> master) Add an orange
|/
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```

# Git references

- We can take a look at it with a convenient Git command, git reflog show:

```
[53] ~/grocery (master)
$ git reflog show
0e8b5cf HEAD@{0}: checkout: moving from 07b18581801f9c2c08c25cad3b43aeee7420ffdd to master
07b1858 HEAD@{1}: commit: Bug eats all the fruits!
e4a5e7b HEAD@{2}: checkout: moving from master to HEAD^
0e8b5cf HEAD@{3}: reset: moving to 0e8b5cf
e4a5e7b HEAD@{4}: reset: moving to HEAD^
0e8b5cf HEAD@{5}: checkout: moving from melons to master
a8c6219 HEAD@{6}: checkout: moving from master to melons
0e8b5cf HEAD@{7}: checkout: moving from berries to master
ef6c382 HEAD@{8}: reset: moving to HEAD^
a8c6219 HEAD@{9}: commit: Add a watermelon
ef6c382 HEAD@{10}: reset: moving to ef6c382
ef6c382 HEAD@{11}: reset: moving to ef6c382
0e8b5cf HEAD@{12}: reset: moving to master
ef6c382 HEAD@{13}: checkout: moving from master to berries
0e8b5cf HEAD@{14}: checkout: moving from berries to master
ef6c382 HEAD@{15}: commit: Add a blackberry
0e8b5cf HEAD@{16}: checkout: moving from master to berries
0e8b5cf HEAD@{17}: commit: Add an orange
e4a5e7b HEAD@{18}: commit: Add an apple
a57d783 HEAD@{19}: commit (amend): Add a banana to the shopping list
c7a0883 HEAD@{20}: commit (initial): Add a banana to the shopping list
```

# Git references

- Actually, here there are all the movements the HEAD reference made in my repository since the beginning, in reverse order, as you may have already noticed.
- In fact, the last one (HEAD@{0}) says:

checkout: moving from  
07b18581801f9c2c08c25cad3b43aeee7420ffdd to master

# Git references

```
[54] ~/grocery (master)
$ git reflog berries
```

```
ef6c382 berries@{0}: reset: moving to HEAD^
a8c6219 berries@{1}: commit: Add a watermelon
ef6c382 berries@{2}: reset: moving to ef6c382
0e8b5cf berries@{3}: reset: moving to master
ef6c382 berries@{4}: commit: Add a blackberry
0e8b5cf berries@{5}: branch: Created from master
```

# Git references

- Creating a tag is simple: you only need the git tag command, followed by a tag name; we can create one in the tip commit of bug branch to give it a try; check out the bug branch:

```
[1] ~/grocery (master)
```

```
$ git checkout bug
```

```
Switched to branch 'bug'
```

- Then use the git tag command followed by the funny bugTag name:

```
[2] ~/grocery (bug)
```

```
$ git tag bugTag
```

# Git references

- Let's see what git log says:

```
[3] ~/grocery (bug)
$ git log --oneline --graph --decorate --all
* 07b1858 (HEAD -> bug, tag: bugTag) Bug eats all the fruits!
| * a8c6219 (melons) Add a watermelon
| * ef6c382 (berries) Add a blackberry
| * 0e8b5cf (master) Add an orange
|/
*
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```

# Git references

- If you do a commit in this branch, you will see the bugTag will remain at its place; add a new line to the same old shopping list file:

```
[4] ~/grocery (bug)
```

```
$ echo "another bug" >> shoppingList.txt
```

- Perform a commit:

```
[5] ~/grocery (bug)
```

```
$ git commit -am "Another bug!"
```

```
[bug 5d605c6] Another bug!
```

```
1 file changed, 1 insertion(+)
```

# Git references

- Then look at the current situation:

[6] ~/grocery (bug)

```
$ git log --oneline --graph --decorate --all
* 5d605c6 (HEAD -> bug) Another bug!
* 07b1858 (tag: bugTag) Bug eats all the fruits!
| * a8c6219 (melons) Add a watermelon
| * ef6c382 (berries) Add a blackberry
| * 0e8b5cf (master) Add an orange
|
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```



# Git references

- Even tags are references, and they are stored, as branches, as simple text files in the tags subfolder within the .git folder; take a look under the .git/refs/tags folder, you will see a bugTag file; look at the content:

```
[7] ~/grocery (bug)
```

```
$ cat .git/refs/tags/bugTag
```

```
07b18581801f9c2c08c25cad3b43aeee7420ffdd
```

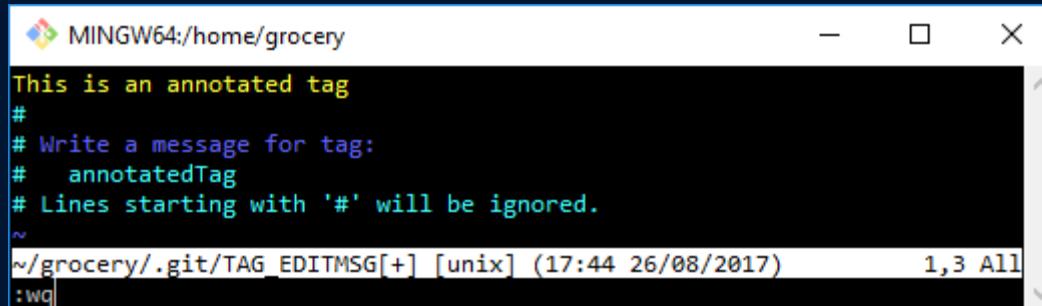
# Git references

- To create one, simply append -a to the command; let's create another one to give this a try:

[8] ~/grocery (bug)

\$ git tag -a annotatedTag 07b1858

- At this point Git opens the default editor, to allow you to write the tag message, as in the following screenshot:



The screenshot shows a terminal window titled "MINGW64:/home/grocery". The command entered is \$ git tag -a annotatedTag 07b1858. The terminal then displays the message "This is an annotated tag" followed by a series of '#' comments. The user has typed the message "# Write a message for tag: # annotatedTag # Lines starting with '#' will be ignored." and pressed Enter. The bottom of the terminal shows the command line again with the message and the status "1,3 All". The cursor is at the end of the message, ready for the user to type their tag message.

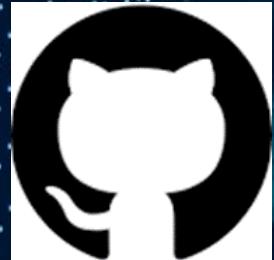
# Git references

- Save and exit, and then see the log:

```
[9] ~/grocery (bug)
```

```
$ git log --oneline --graph --decorate --all
```

```
* 5d605c6 (HEAD -> bug) Another bug!
* 07b1858 (tag: bugTag, tag: annotatedTag) Bug eats all the fruits!
| * a8c6219 (melons) Add a watermelon
| * ef6c382 (berries) Add a blackberry
| * 0e8b5cf (master) Add an orange
|/
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```



# Git references

- A new ref has been created:

```
[10] ~/grocery (bug)
```

```
$ cat .git/refs/tags/annotatedTag
```

```
17c289ddf23798de6eee8fe6c2e908cf0c3a6747
```

- But even a new object: try to cat-file the hash you see in the reference:

```
[11] ~/grocery (bug)
```

```
$ git cat-file -p 17c289
```

```
object 07b18581801f9c2c08c25cad3b43aeee7420ffdd
```

```
type commit
```

```
tag annotatedTag
```

```
tagger Ferdinando Santacroce <ferdinando.santacroce@gmail.com> 150376226 4 +0200
```

```
This is an annotated tag
```

# Merging in Git

# Merging branches

In Git, merging two (or more!) branches is the act of making their personal history meet each other. When they meet, two things can happen:

- Files in their tip commit are different, so some conflict will rise
- Files do not conflict
- Commits of the target branch are directly behind commits of the branch we are merging, so a fast-forward will happen

# Merging branches

- Let's give it a try.
- We can try to merge the melons branch into the master one; to do so, you have to check out the target branch, master in this case, and then fire a git merge <branch name> command; as I'm already on the master branch, I go straight with the merge command:

```
[1] ~/grocery (master)
$ git merge melons
Auto-merging shoppingList.txt
CONFLICT (content): Merge conflict in shoppingList.txt
Automatic merge failed; fix conflicts and then commit the result.
```

# Merging branches

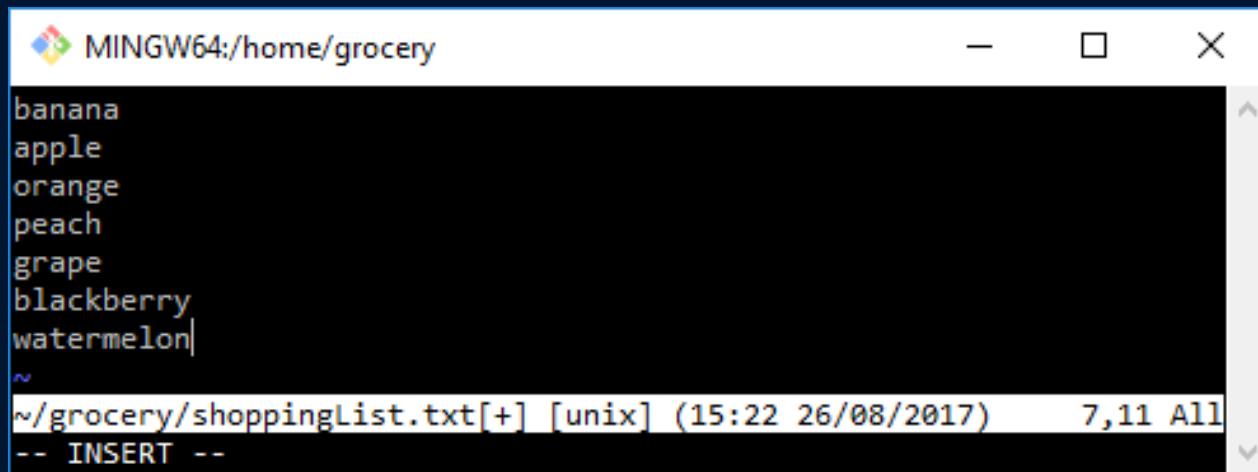
- See the conflict with git diff:

```
[2] ~/grocery (master|MERGING)
$ git diff
diff --cc shoppingList.txt
index 862debc,7786024..0000000
--- a/shoppingList.txt
+++ b/shoppingList.txt
@@@ -1,5 -1,5 +1,10 @@@
    banana
    apple
    orange
+<<<<<< HEAD
+peach
-grape

++grape
+=====
+ blackberry
+ watermelon
++>>>>> melons
```

# Merging branches

- I will edit the file enqueueing blackberry and watermelon after peach and grape, as per the following screenshot:



A screenshot of a terminal window titled "MINGW64:/home/grocery". The window contains the following text:

```
banana
apple
orange
peach
grape
blackberry
watermelon|
```

The cursor is positioned at the end of the word "watermelon". Below the terminal window, the status bar displays the path "/grocery/shoppingList.txt[+]" and the date and time "(15:22 26/08/2017)". To the right of the status bar, there are three numbers: "7,11 All". At the bottom left of the terminal window, the text "-- INSERT --" is visible.

# Merging branches

- After saving the file, add it to the staging area and then commit:

```
[3] ~/grocery (master|MERGING)
$ git add shoppingList.txt
```

```
[4] ~/grocery (master|MERGING)
$ git commit -m "Merged melons branch into master"
[master e18a921] Merged melons branch into master
```

# Merging branches

- Now take a look at the log:

```
[5] ~/grocery (master)
$ git log --oneline --graph --decorate --all
*   e18a921 (HEAD -> master) Merged melons branch into master
|\ \
| * a8c6219 (melons) Add a watermelon
| * ef6c382 (berries) Add a blackberry
* | 6409527 Add a grape
* | 603b9d1 Add a peach
|/
* 0e8b5cf Add an orange
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```

# Merging branches

- Suggestion: look at the merge commit with git cat-file -p:

```
[6] ~/grocery (master)
$ git cat-file -p HEAD
tree 2916dd995ee356351c9b49a5071051575c070e5f
parent 6409527a1f06d0bbe680d461666ef8b137ac7135
parent a8c62190fb1c54d1034db78a87562733a6e3629c
author Ferdinando Santacroce <ferdinando.santacroce@gmail.com> 1503754221 +0200
committer Ferdinando Santacroce <ferdinando.santacroce@gmail.com> 1503754221 +0200
```

Merged melons branch into master

# Merging branches

## Fast forwarding

- A merge not always generates a new commit; to test this case, try to merge the melons branch into a berries one:

```
[7] ~/grocery (master)
$ git checkout berries
Switched to branch 'berries'

[8] ~/grocery (berries)
$ git merge melons
Updating ef6c382..a8c6219
Fast-forward
  shoppingList.txt | 1 +
  1 file changed, 1 insertion(+)

[9] ~/grocery (berries)
$ git log --oneline --graph --decorate --all
*   e18a921 (master) Merged melons branch into master
|\ \
| * a8c6219 (HEAD -> berries, melons) Add a watermelon
| * ef6c382 Add a blackberry
* | 6409527 Add a grape
* | 603b9d1 Add a peach
|/
* 0e8b5cf Add an orange
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```

# Merging branches

- Move back the berries branch where it was using git reset:

```
[10] ~/grocery (berries)
$ git reset --hard HEAD^
HEAD is now at ef6c382 Add a blackberry

[11] ~/grocery (berries)
$ git log --oneline --graph --decorate --all
*   e18a921 (master) Merged melons branch into master
|\ \
| * a8c6219 (melons) Add a watermelon
| * ef6c382 (HEAD -> berries) Add a blackberry
* | 6409527 Add a grape
* | 603b9d1 Add a peach
|/
* 0e8b5cf Add an orange
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```

# Merging branches

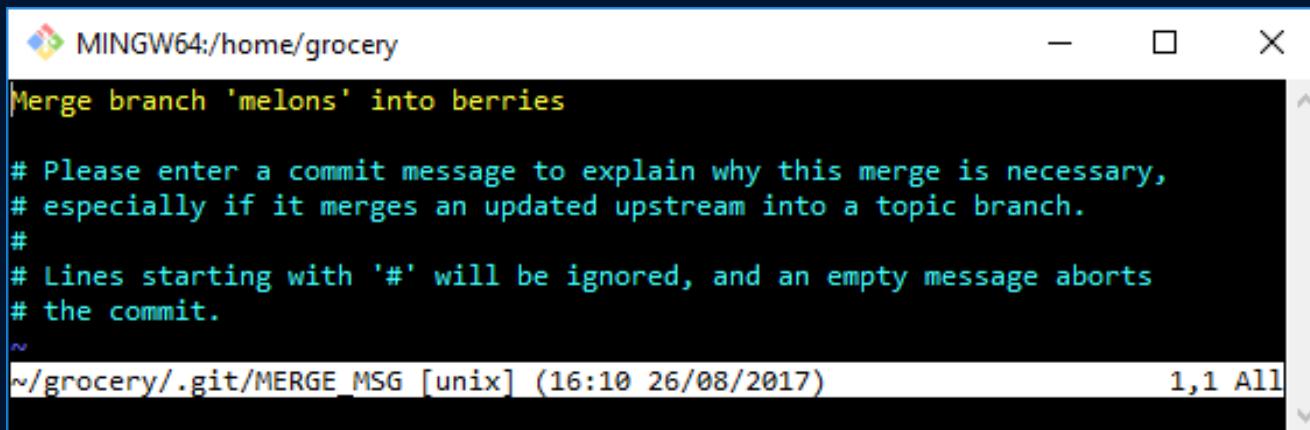
- We have just undone a merge, did you realize it?
- Okay, now do the merge again with the --no-ff option:

```
[12] ~/grocery (berries)  
$ git merge --no-ff melons
```



# Merging branches

- Git will now open your default editor to allow you to specify a commit message, as shown in the following screenshot:



The terminal window shows the following text:

```
Merge branch 'melons' into berries

# Please enter a commit message to explain why this merge is necessary,
# especially if it merges an updated upstream into a topic branch.
#
# Lines starting with '#' will be ignored, and an empty message aborts
# the commit.
~
~/grocery/.git/MERGE_MSG [unix] (16:10 26/08/2017) 1,1 All
```

# Merging branches

- Accept the default message, save and exit:

```
[13] ~/grocery (berries)
```

```
Merge made by the 'recursive' strategy.--all  
shoppingList.txt | 1 +  
1 file changed, 1 insertion(+)
```

# Merging branches

- Now a git log:

```
[14] ~/grocery (berries)
$ git log --oneline --graph --decorate --all
* cb912b2 (HEAD -> berries) Merge branch 'melons' into berries
|\ \
| | * e18a921 (master) Merged melons branch into master
| | |\ \
| | | / \
| | | |
| | | *
| | | * a8c6219 (melons) Add a watermelon
| | |
* | ef6c382 Add a blackberry
| * 6409527 Add a grape
| * 603b9d1 Add a peach
| /
* 0e8b5cf Add an orange
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```

# Merging branches

- We are done with these experiments; anyway, I want to undo this merge, because I want to keep the repository as simple as possible to allow you to better understand the exercise we do together; go with a git reset --hard HEAD^:

```
[15] ~/grocery (berries)
$ git reset --hard HEAD^
HEAD is now at ef6c382 Add a blackberry

[16] ~/grocery (berries)
$ git log --oneline --graph --decorate --all
*   e18a921 (master) Merged melons branch into master
|\ \
| * a8c6219 (melons) Add a watermelon
| * ef6c382 (HEAD -> berries) Add a blackberry
* | 6409527 Add a grape
* | 603b9d1 Add a peach
|/
* 0e8b5cf Add an orange
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```

# Merging branches

Okay, now undo even the past merge we did on the master branch:

```
[17] ~/grocery (master)
$ git reset --hard HEAD^
HEAD is now at 6409527 Add a grape

[18] ~/grocery (master)
$ git log --oneline --graph --decorate --all
* 6409527 (HEAD -> master) Add a grape
* 603b9d1 Add a peach
| * a8c6219 (melons) Add a watermelon
| * ef6c382 (berries) Add a blackberry
|/
* 0e8b5cf Add an orange
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```

# 6. Collaborating on Your Code



# Git Fundamentals - Working Remotely

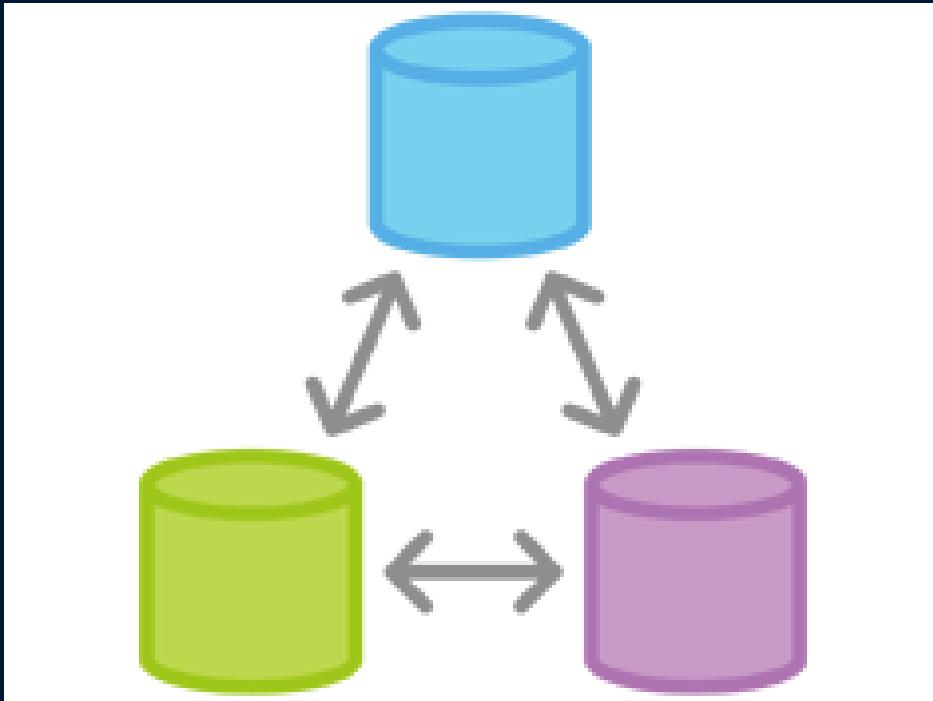
In this lesson, we finally start to work in a distributed manner, using remote servers as a contact point for different developers. These are the main topics we will focus on:

- Dealing with remotes
- Cloning a remote repository
- Working with online hosting services, such as Azure DevOps and GitHub

# Working with remotes

- Git is a tool for versioning files, as you know, but it has been built with collaboration in mind.
- In 2005, Linus Torvalds had the need for a light and efficient tool to share the Linux kernel code, allowing him and hundreds of other people to work on it without going crazy.

# Working with remotes



# Working with remotes

## Clone a local repository

- Create a new folder on your disk to clone our grocery repository:

[1] ~

```
$ mkdir grocery-cloned
```

- Then clone the grocery repository using the git clone command:

```
[2] ~ $ cd grocery-cloned [3] ~/grocery-cloned $ git clone ~/grocery . Cloning into '.'...
```

# Working with remotes

- Now, go directly to the point with a git log command:

```
[4] ~/grocery-cloned (master)
$ git log --oneline --graph --decorate --all
* 6409527 (HEAD -> master, origin/master, origin/HEAD) Add a grape
* 603b9d1 Add a peach
| * a8c6219 (origin/melons) Add a watermelon
| * ef6c382 (origin/berries) Add a blackberry
|/
* 0e8b5cf Add an orange
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```

# Working with remotes

- But don't worry: a local branch in which to work locally can be created by simply checking it out:

```
[5] ~/grocery-cloned (master)
```

```
$ git checkout berries
```

```
Branch berries set up to track remote branch berries from origin.  
Switched to a new branch 'berries'
```

# Working with remotes

- Now, look at the log again:

```
[6] ~/grocery-cloned (berries)
$ git log --oneline --graph --decorate --all
* 6409527 (origin/master, origin/HEAD, master) Add a grape
* 603b9d1 Add a peach
| * a8c6219 (origin/melons) Add a watermelon
| * ef6c382 (HEAD -> berries, origin/berries) Add a blackberry
|/
* 0e8b5cf Add an orange
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```

# Working with remotes

- Let's try:

```
[7] ~/grocery-cloned (berries)
$ echo "blueberry" >> shoppingList.txt
```

```
[8] ~/grocery-cloned (berries)
$ git commit -am "Add a blueberry"
[berries ab9f231] Add a blueberry
Committer: Santacroce Ferdinando <san@intre.it>
```

# Working with remotes

- You can suppress this message by setting them explicitly:

```
git config --global user.name "Your Name"
```

```
git config --global user.email you@example.com
```

- After doing this, you may fix the identity used for this commit with the following code:

```
git commit --amend --reset-author
```

1 file changed, 1 insertion(+)

# Working with remotes

OK, let's see what happened:

```
[9] ~/grocery-cloned (berries)
$ git log --oneline --graph --decorate --all
* ab9f231 (HEAD -> berries) Add a blueberry
| * 6409527 (origin/master, origin/HEAD, master) Add a grape
| * 603b9d1 Add a peach
| | * a8c6219 (origin/melons) Add a watermelon
| |
|/
|/
* | ef6c382 (origin/berries) Add a blackberry
|/
* 0e8b5cf Add an orange
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```

# Working with remotes

- Now, we will try to push the modifications in the berries branch to the origin; the command is git push, followed by the name of the remote and the target branch:

```
[10] ~/grocery-cloned (berries)
$ git push origin berries
Counting objects: 3, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 323 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To C:/Users/san/Google Drive/Packt/PortableGit/home/grocery
    ef6c382..ab9f231  berries -> berries
```

# Working with remotes

- Now, we obviously want to see if, in the remote repository, there is a new commit in the berries branch; so, open the grocery folder in a new console and do git log:

```
[11] ~
$ cd grocery

[12] ~/grocery (master)
$ git log --oneline --graph --decorate --all
* ab9f231 (berries) Add a blueberry
| * 6409527 (HEAD -> master) Add a grape
| * 603b9d1 Add a peach
| | * a8c6219 (melons) Add a watermelon
| |
|/
|/
* | ef6c382 Add a blackberry
|/
* 0e8b5cf Add an orange
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```

# Working with remotes

## Getting remote commits with git pull

- Now, it's time to experiment the inverse: retrieving updates from the remote repository and applying them to our local copy.
- So, make a new commit in the grocery repository, and then, we will download it into the grocery-cloned one:

```
[13] ~/grocery (master)  
$ printf "\r\n" >> shoppingList.txt
```

# Working with remotes

- I firstly need to create a new line, because due to the previous grape rebase, we ended having the shoppinList.txt file with no new line at the end, as echo " " >> <file> usually does:

```
[14] ~/grocery (master)  
$ echo "apricot" >> shoppingList.txt
```

```
[15] ~/grocery (master)  
$ git commit -am "Add an apricot"  
[master 741ed56] Add an apricot  
1 file changed, 2 insertions(+), 1 deletion(-)
```

# Working with remotes

- Let's try git pull for now, then we will try to use git fetch and git merge separately.
- Go back to the grocery-cloned repository, switch to the master branch, and do a git pull:

```
[16] ~/grocery-cloned (berries)
$ git checkout master
Your branch is up-to-date with 'origin/master'.
Switched to branch 'master'
```

# Working with remotes

- For now, go with git pull: the command wants you to specify the name of the remote you want to pull from, which is origin in this case, and then the branch you want to merge into your local one, which is master, of course:

```
[17] ~/grocery-cloned (master)
$ git pull origin master
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 C:/Users/san/Google Drive/Packt/PortableGit/home/grocery
 * branch      master      -> FETCH_HEAD
   6409527..741ed56  master      -> origin/master
Updating 6409527..741ed56
Fast-forward
  shoppingList.txt | 3 +++
  1 file changed, 2 insertions(+), 1 deletion(-)
```

# Working with remotes

- OK, now I want you to try doing these steps in a separate manner; create the umpteenth new commit in the grocery repository, the master branch:

```
[18] ~/grocery (master)
$ echo "plum" >> shoppingList.txt

[19] ~/grocery (master)
$ git commit -am "Add a plum"
[master 50851d2] Add a plum
1 file changed, 1 insertion(+)
```

# Working with remotes

- Now perform a git fetch on grocery-cloned repository:

```
[20] ~/grocery-cloned (master)
$ git fetch
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 C:/Users/san/Google Drive/Packt/PortableGit/home/grocery
    741ed56..50851d2  master      -> origin/master
```

# Working with remotes

- Do a git status now:



```
[21] ~/grocery-cloned (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
```

# Working with remotes

- Now, let's sync with a git merge; to merge a remote branch, we have to specify, other than the branch name, even the remote one, as we did in the git pull command previously:

```
[22] ~/grocery-cloned (master)
$ git merge origin master
Updating 741ed56..50851d2
Fast-forward
  shoppingList.txt | 1 +
  1 file changed, 1 insertion(+)
```

# Working with remotes

## How Git keeps track of remotes

- Git stores remote branch labels in a similar way to how it stores the local branches ones; it uses a subfolder in refs for the scope, with the symbolic name we used for the remote, in this case origin, the default one:

```
[23] ~/grocery-cloned (master)
$ ll .git/refs/remotes/origin/
total 3
drwxr-xr-x 1 san 1049089 0 Aug 27 11:25 .
drwxr-xr-x 1 san 1049089 0 Aug 26 18:19 ../
-rw-r--r-- 1 san 1049089 41 Aug 26 18:56 berries
-rw-r--r-- 1 san 1049089 32 Aug 26 18:19 HEAD
-rw-r--r-- 1 san 1049089 41 Aug 27 11:25 master
```

COMPLETE

Activity: Creating a Pull  
Request

# Working with a public server on GitHub

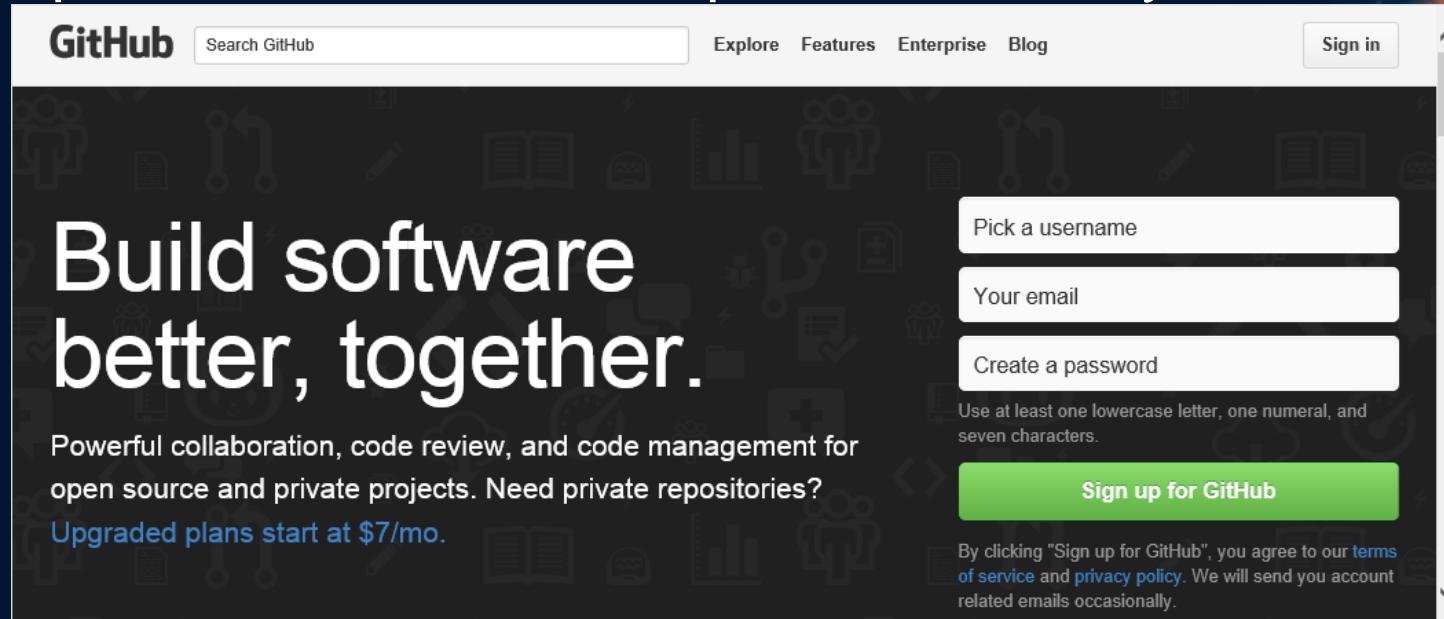
To start working with a public hosted remote, we have to get one.

Today, it is not difficult to achieve; the world has plenty of free online services offering room for Git repositories.

One of the most commonly used is GitHub.

# Working with a public server on GitHub

Sign up, filling the textboxes, as per the following image, and provide a username, a password, and your email:



# Working with a public server on GitHub

 Search GitHub

Explore Gist Blog Help

 fsantacroce +     

Contributions  Repositories  Public activity 

**Contributions**

Dec Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov

M  
W  
F

Summary of Pull Requests, issues opened, and commits. [Learn more.](#)

Less  More

This is your **contribution graph**. When you make a commit to a repository, you'll get a  for that day. Make more contributions and you'll get a darker green square. Over time, your chart might start looking [something like this](#).

We have a quick guide that will show you how to create your first repository. You'll also make a commit and **earn your first green square!**

 [Read the Hello World guide](#)

0 Followers 0 Starred 0 Following

# Working with a public server on GitHub

The screenshot shows a GitHub user profile for the account `fsantacroce`. The top navigation bar includes links for Explore, Gist, Blog, Help, and a user icon. To the right of the user icon are options for adding a repository, creating a gist, settings, and a download link. Below the navigation is a sidebar with links for Repositories and Public activity, and a prominent [Edit profile](#) button. A search bar is followed by filters for All, Public, Private, Sources, Forks, and Mirrors, with a green [New](#) button. The main content area displays the message: **fsantacroce doesn't have any public repositories yet.**

# Working with a public server on GitHub

Owner  / Repository name

Great repository names are short and memorable. Need inspiration? How about [yolo-ironman](#).

Description (optional)

This repository contains recipes I like to share with my friends

 Public  
Anyone can see this repository. You choose who can commit.

 Private  
You choose who can see and commit to this repository.

Initialize this repository with a README  
This will allow you to git clone the repository immediately. Skip this step if you have already run git init locally.

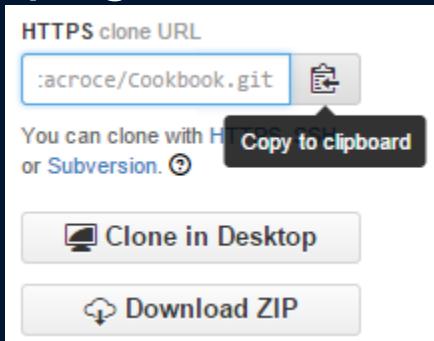
Add .gitignore:  | Add a license:

**Create repository**

# Working with a public server on GitHub

## Cloning the repository

- Using this command is quite simple; in this case, all we need to know is the URL of the repository to clone.
- The URL is provided by GitHub on the right down part of the repository home page:



# Working with a public server on GitHub

- Obviously, the URL of your repository will be different; as you can see, GitHub URLs are composed as follows:

<https://github.com/<Username>/<RepositoryName>.git>:

```
[1] ~
$ git clone https://github.com/fsantacroce/Cookbook.git
Cloning into 'Cookbook'...
remote: Counting objects: 15, done.
remote: Total 15 (delta 0), reused 0 (delta 0), pack-reused 15
Unpacking objects: 100% (15/15), done.

[2] ~
$ cd Cookbook/

[3] ~/Cookbook (master)
$ ll
total 13
drwxr-xr-x 1 san 1049089 0 Aug 27 14:16 .
drwxr-xr-x 1 san 1049089 0 Aug 27 14:16 ..
drwxr-xr-x 1 san 1049089 0 Aug 27 14:16 .git/
-rw-r--r-- 1 san 1049089 150 Aug 27 14:16 README.md
```



# Working with a public server on GitHub

## **Uploading modifications to remotes**

So, let's try to edit the README.md file and upload modifications to GitHub:

- Edit the README.md file using your preferred editor, adding, for example, a new sentence.
- Add it to the index and then commit.
- Put your commit on the remote repository using the git push command.

# Working with a public server on GitHub

- But firstly, set the user and email this time, so Git will not output the message we have seen in the previous lessons:

```
[4] ~/Cookbook (master)
$ git config user.name "Ferdinando Santacroce"

[5] ~/Cookbook (master)
$ git config user.email "ferdinando.santacroce@gmail.com"

[6] ~/Cookbook (master)
$ vim README.md

Add a sentence then save and close the editor.

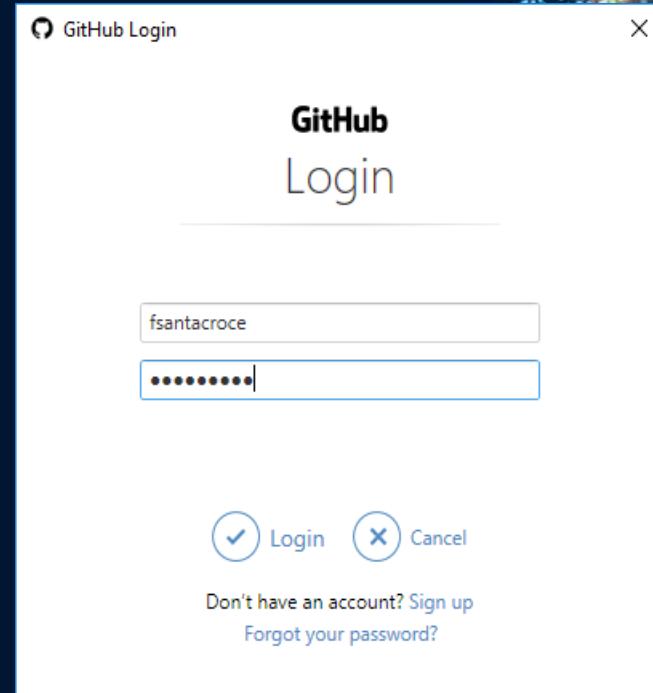
[7] ~/Cookbook (master)
$ git add README.md

[8] ~/Cookbook (master)
$ git commit -m "Add a sentence to readme"
[master 41bdbe6] Add a sentence to readme
 1 file changed, 2 insertions(+)
```

# Working with a public server on GitHub

- Now, try to type git push and press ENTER, without specifying anything else:

```
[9] ~/Cookbook (master)  
$ git push
```



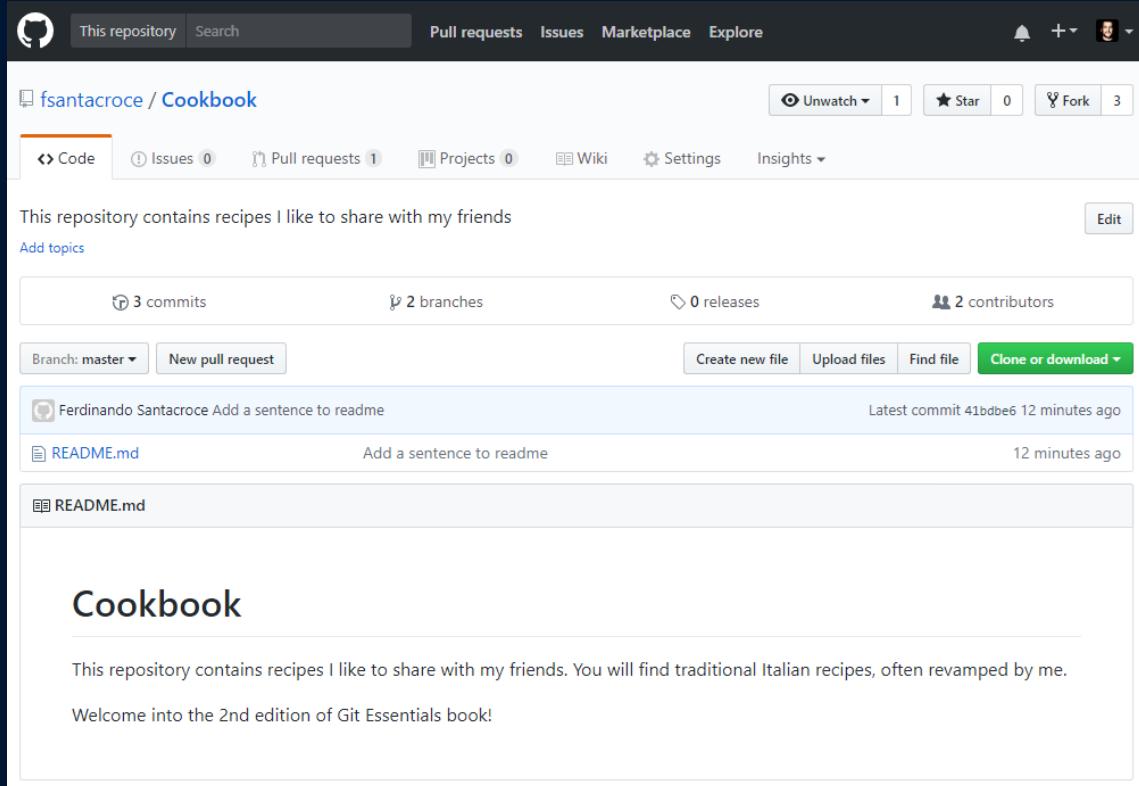
- Here, in my Windows 10 workstation, this window appears:

# Working with a public server on GitHub

- Input your credentials, and then press the Login button; after that, Git continues with:

```
[10] ~/Cookbook (master)
$ git push
Counting objects: 3, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 328 bytes | 0 bytes/s, done.
Total 3 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To https://github.com/fsantacroce/Cookbook.git
  e1e7236..41bdbe6  master -> master
```

# Working with a public server on GitHub



This repository contains recipes I like to share with my friends

Add topics

3 commits 2 branches 0 releases 2 contributors

Branch: master New pull request Create new file Upload files Find file Clone or download

Ferdinando Santacroce Add a sentence to readme Latest commit 41bdb6 12 minutes ago

README.md Add a sentence to readme 12 minutes ago

README.md

## Cookbook

This repository contains recipes I like to share with my friends. You will find traditional Italian recipes, often revamped by me.

Welcome into the 2nd edition of Git Essentials book!

# Working with a public server on GitHub

## Pushing a new branch to the remote

- Create a new branch, for instance Risotti.
- Add to it a new file, for example, Risotto-all-a-Milanese.md, and commit it.
- Push the branch to the remote using `git push -u origin Risotti`.

# Working with a public server on GitHub

```
[11] ~/Cookbook (master)
$ git branch Risotti

[12] ~/Cookbook (master)
$ git checkout Risotti

[13] ~/Cookbook (Risotti)
$ notepad Risotto-alla-Milanese.md

[14] ~/Cookbook (Risotti)
$ git add Risotto-alla-Milanese.md

[15] ~/Cookbook (Risotti)
$ git commit -m "Add risotto alla milanese recipe ingredients"
[Risotti b62bc1f] Add risotto alla milanese recipe ingredients
 1 file changed, 15 insertions(+)
 create mode 100644 Risotto-alla-Milanese.md

[16] ~/Cookbook (Risotti)
$ git push -u origin Risotti
Total 0 (delta 0), reused 0 (delta 0)
Branch Risotti set up to track remote branch Risotti from origin.
To https://github.com/fsantacroce/Cookbook.git
 * [new branch]      Risotti -> Risotti
```

# Working with a public server on GitHub

- If you want to see remotes actually configured in your repository, you can type a simple git remote command, followed by -v (--verbose) to get some more details:

```
[17] ~/Cookbook (master)
```

```
$ git remote -v
```

```
origin https://github.com/fsantacroce/Cookbook.git (fetch)  
origin https://github.com/fsantacroce/Cookbook.git (push)
```

# Working with a public server on GitHub

- To better understand the way our repository is now configured, try to type `git remote show origin`:

```
[18] ~/Cookbook (master)
$ git remote show origin
* remote origin
  Fetch URL: https://github.com/fsantacroce/Cookbook.git
  Push  URL: https://github.com/fsantacroce/Cookbook.git
  HEAD branch: master
  Remote branches:
    Pasta   tracked
    Risotti tracked
    master  tracked
  Local branches configured for 'git pull':
    Risotti merges with remote Risotti
    master  merges with remote master
  Local refs configured for 'git push':
    Risotti pushes to Risotti (up to date)
    master   pushes to master  (fast-forwardable)
```



# Working with a public server on GitHub

Create a new local repository to publish, following these simple steps:

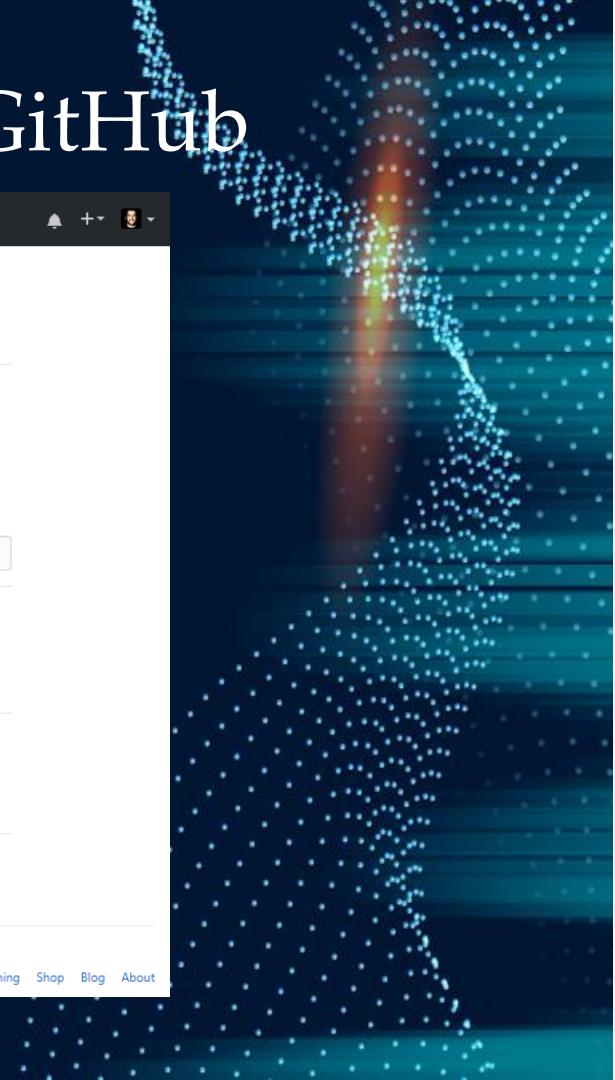
- Go to our local repositories folder.
- Create a new HelloWorld folder.
- In it place a new repository, as we did in first lesson.

# Working with a public server on GitHub

```
[19] ~  
$ mkdir HelloWorld  
  
[20] ~  
$ cd HelloWorld/  
  
[21] ~/HelloWorld  
$ git init  
Initialized empty Git repository in C:/Users/san/Google  
Drive/Packt/PortableGit/home>HelloWorld/.git/  
  
[22] ~/HelloWorld (master)  
$ echo "Hello World!" >> README.md  
  
[23] ~/HelloWorld (master)  
$ git add README.md  
  
[24] ~/HelloWorld (master)  
$ git config user.name "Ferdinando Santacroce"  
  
[25] ~/HelloWorld (master)  
$ git config user.email "ferdinando.santacroce@gmail.com"  
  
[26] ~/HelloWorld (master)  
$ git commit -m "First commit"  
[master (root-commit) 5b41441] First commit  
1 file changed, 1 insertion(+)  
create mode 100644 README.md  
  
[27] ~/HelloWorld (master)
```

- Add a new README.md file and commit it.

# Working with a public server on GitHub



Screenshot of the GitHub 'Create a new repository' page.

The page title is 'Create a new repository'. Below it, a sub-instruction reads: 'A repository contains all the files for your project, including the revision history.'

The 'Owner' field shows 'fsantacroce' and the 'Repository name' field shows 'HelloWorld' with a green checkmark icon.

The 'Description (optional)' field contains the text 'A simple repository for tests'.

The 'Visibility' section shows 'Public' selected (radio button is checked), with the description: 'Anyone can see this repository. You choose who can commit.' Below it, 'Private' is also listed with the description: 'You choose who can see and commit to this repository.'

The 'Initialize this repository with a README' checkbox is unchecked, with the note: 'This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.'

Below the checkbox are two buttons: 'Add .gitignore: None ▾' and 'Add a license: None ▾'.

A large green 'Create repository' button is at the bottom.

Page footer: © 2017 GitHub, Inc. Terms Privacy Security Status Help Contact GitHub API Training Shop Blog About

# Working with a public server on GitHub

- Adding a remote to a local repository
- To publish our HelloWorld repository, we simply have to add its first remote; adding a remote is quite simple: `git remote add origin <remote-repository-url>`
- So, this is the full command we have to type in the Bash shell:

```
[27] ~/HelloWorld (master)
$ git remote add origin https://github.com/fsantacroce>HelloWorld.git
```

# Working with a public server on GitHub

## Pushing a local branch to a remote repository

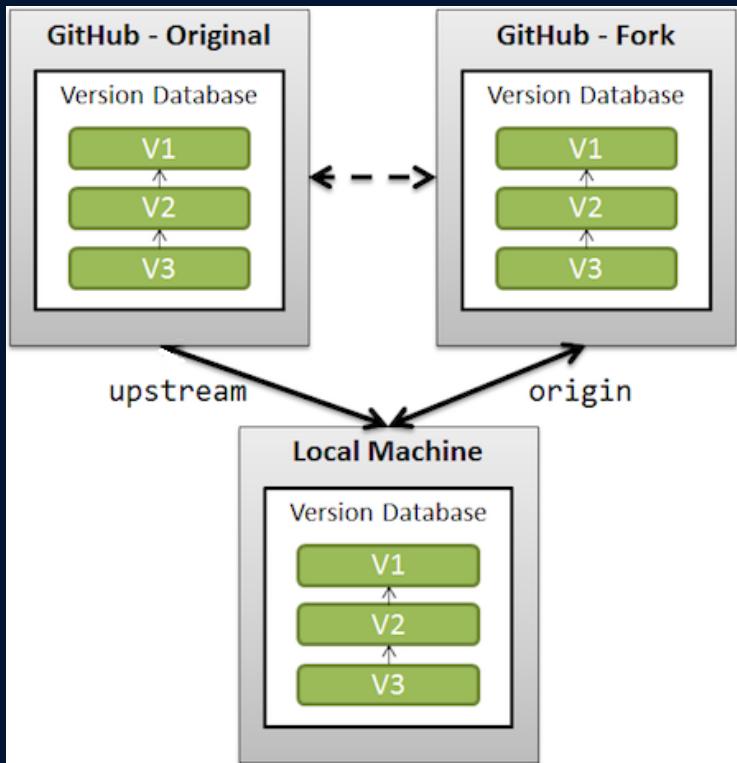
- After that, push your local changes to the remote using  
git push -u origin master:

```
[28] ~/HelloWorld (master)
$ git push -u origin master
Counting objects: 3, done.
Writing objects: 100% (3/3), 231 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
Branch master set up to track remote branch master from origin.
To https://github.com/fsantacroce/HelloWorld.git
 * [new branch]      master -> master
```

# Working with a public server on GitHub

- When you fork on GitHub, what you get is essentially a server-side clone of the repository on your GitHub account,
- If you clone your forked repository locally, in the remote list, you will find an origin that points to your account repository.
- While the original repository will assume the upstream alias (you have to add it manually anyway).

# Working with a public server on GitHub

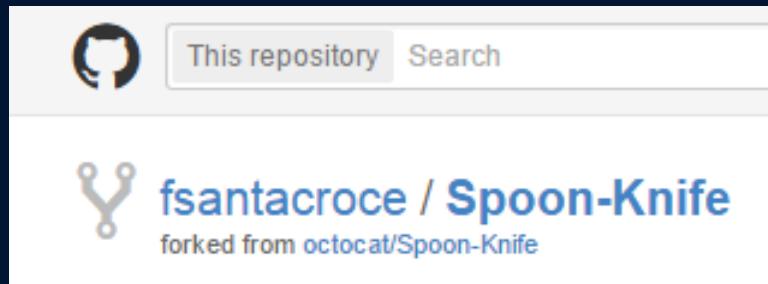


# Working with a public server on GitHub

- Fork the repository using the Fork button at the right of the page:



- After a funny photocopy animation, you will get a brand-new Spoon-Knife repository in your GitHub account:



# Working with a public server on GitHub

- Now, you can clone that repository locally, as we did before:

```
[1] ~  
$ git clone https://github.com/fsantacroce/Spoon-Knife.git  
Cloning into 'Spoon-Knife'...  
remote: Counting objects: 19, done.  
remote: Total 19 (delta 0), reused 0 (delta 0), pack-reused 19  
Unpacking objects: 100% (19/19), done.  
  
[2] ~  
$ cd Spoon-Knife  
  
[3] ~/Spoon-Knife (master)  
$ git remote -v  
origin https://github.com/fsantacroce/Spoon-Knife.git (fetch)  
origin https://github.com/fsantacroce/Spoon-Knife.git (push)
```

# Working with a public server on GitHub

- As you can see, the upstream remote is not present, you have to add it manually; to add it, use the git remote add command:

```
[4] ~/Spoon-Knife (master)
$ git remote add upstream https://github.com/octocat/Spoon-Knife.git

[5] ~/Spoon-Knife (master)
$ git remote -v
origin https://github.com/fsantacroce/Spoon-Knife.git (fetch)
origin https://github.com/fsantacroce/Spoon-Knife.git (push)
upstream https://github.com/octocat/Spoon-Knife.git (fetch)
upstream https://github.com/octocat/Spoon-Knife.git (push)
```

# Working with a public server on GitHub

## Creating a pull request

- To create a pull request, you have to go on your GitHub account and make it directly from your forked account; but first, you have to know that pull requests can be made only from separated branches.
- At this point of the course, you are probably used to creating a new branch for a new feature or refactor purpose, so this is nothing new, isn't it?

# Working with a public server on GitHub

- To make an attempt, let's create a local TeaSpoon branch in our repository, commit a new file, and push it to our GitHub account:

```
[6] ~/Spoon-Knife (master)
$ git branch TeaSpoon

[7] ~/Spoon-Knife (master)
$ git checkout TeaSpoon
Switched to branch 'TeaSpoon'

[8] ~/Spoon-Knife (TeaSpoon)
$ vi TeaSpoon.md

[9] ~/Spoon-Knife (TeaSpoon)
$ git add TeaSpoon.md

[10] ~/Spoon-Knife (TeaSpoon)
$ git commit -m "Add a TeaSpoon to the cutlery"
[TeaSpoon 62a99c9] Add a TeaSpoon to the cutlery
1 file changed, 2 insertions(+)
 create mode 100644 TeaSpoon.md

[11] ~/Spoon-Knife (TeaSpoon)
$ git push origin TeaSpoon
Counting objects: 3, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 417 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/fsantacroce/Spoon-Knife.git
d0dd1f6..62a99c9  TeaSpoon -> TeaSpoon
```

# Working with a public server on GitHub

This repo is for demonstration purposes only.

Add topics

3 commits 4 branches 0 releases 1 contributor

Branch: master New pull request Create new file Upload files Find file Clone or download

This branch is even with octocat:master.

octocat Pointing to the guide for forking Latest commit d0dd1f6 on Feb 13, 2014

README.md Pointing to the guide for forking 4 years ago

index.html Created index page for future collaborative edits 4 years ago

styles.css Create styles.css and updated README 4 years ago

README.md

**Well hello there!**

This repository is meant to provide an example for *forking* a repository on GitHub.

Creating a *fork* is producing a personal copy of someone else's project. Forks act as a sort of bridge between the original repository and your personal copy. You can submit *Pull Requests* to help make other people's projects better by offering your changes up to the original project. Forking is at the core of social coding at GitHub.

After forking this repository, you can make some changes to the project, and submit a *Pull Request* as practice.

For some more information on how to fork a repository, [check out our guide, "Forking Projects"](#). Thanks! ❤

# Working with a public server on GitHub

This repository | Search Pull requests Issues Marketplace Explore

octocat / Spoon-Knife Watch 310 Star 9,998 Fork 93,201

Code Issues 773 Pull requests 5,000+ Projects 0 Wiki Insights

## Comparing changes

Choose two branches to see what's changed or to start a new pull request. If you need to, you can also compare across forks.

base fork: octocat/Spoon-Knife base: master ... head fork: fsantacroce/Spoon-Knife compare: master

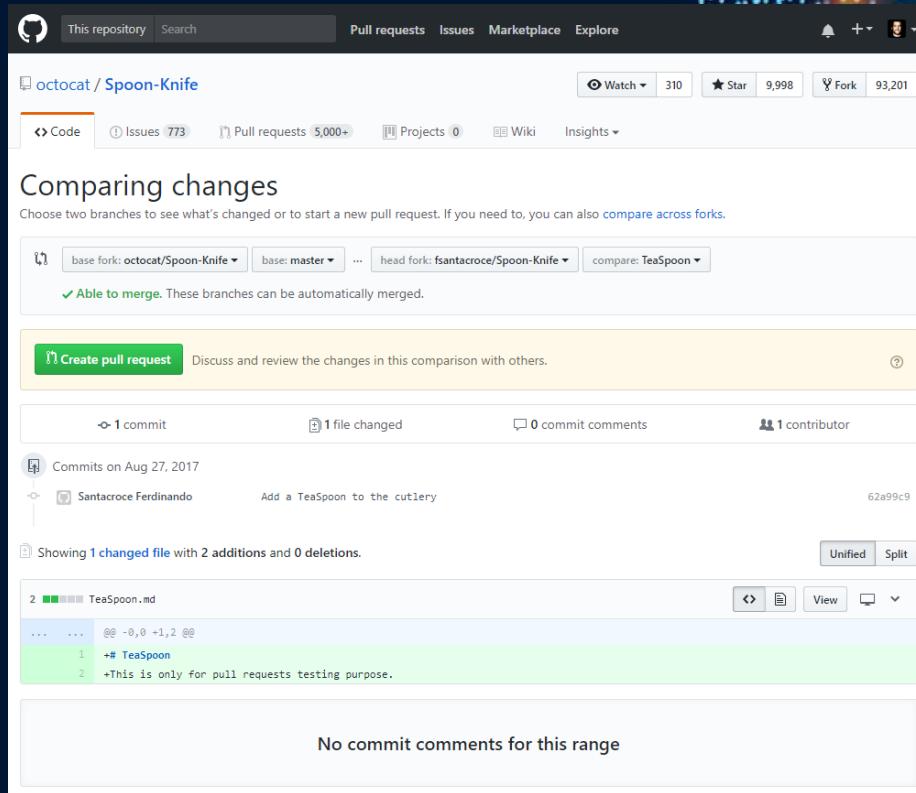
Choose a head branch

- test #12724 No description available
- changed #9216 Here are some changes...
- update the title, add index back up file #8716 update the title, add index back up file
- Masterrrr #8480 No description available
- added new file #5838 fasfd

There isn't anything to compare.  
octocat:master and fsantacroce:master are identical.

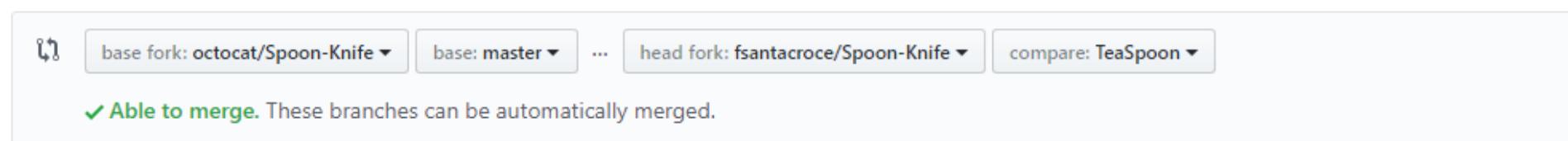
# Working with a public server on GitHub

- Go to the branches combo (1), select TeaSpoon branch (2), and then GitHub will show you something similar to the following screenshot:



# Working with a public server on GitHub

- In the top left corner of the preceding screenshot, you will find what branches GitHub is about to compare for you; take a look at details in the following image:



# Working with a public server on GitHub

- This means that you are about to compare your local TeaSpoon branch with the original master branch of the octocat user.
  - At the end of the page, you can see all the different details (files added, removed, changed, and so on):

2茶匙.md

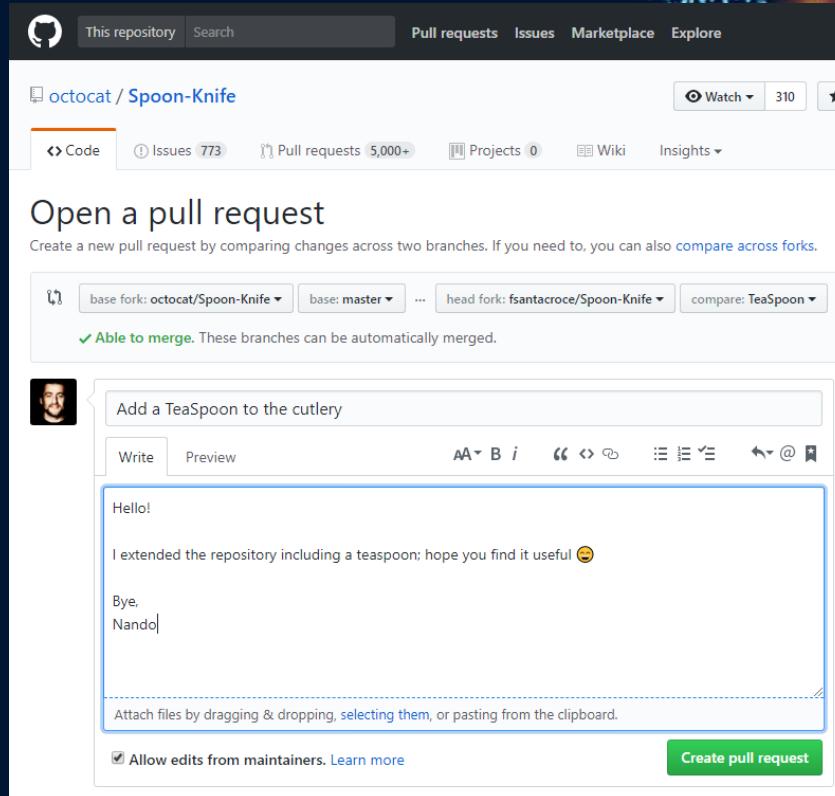
... ... @@ -0,0 +1,2 @@

1 +# TeaSpoon

2 +This is only for pull requests testing purpose.

# Working with a public server on GitHub

- Now, you can click on the green Create pull request button; the window in the following screenshot will appear:



# Summary

- In this lesson, we finally got in touch with the Git ability to manage multiple remote copies of repositories.
- This gives you a wide range of possibilities to better organize your collaboration workflow inside your team.
- In the next lesson, you will learn some advanced techniques using well-known and niche commands.

COMPLETE

Activity: Code Review

# 7. Merging Pull Requests

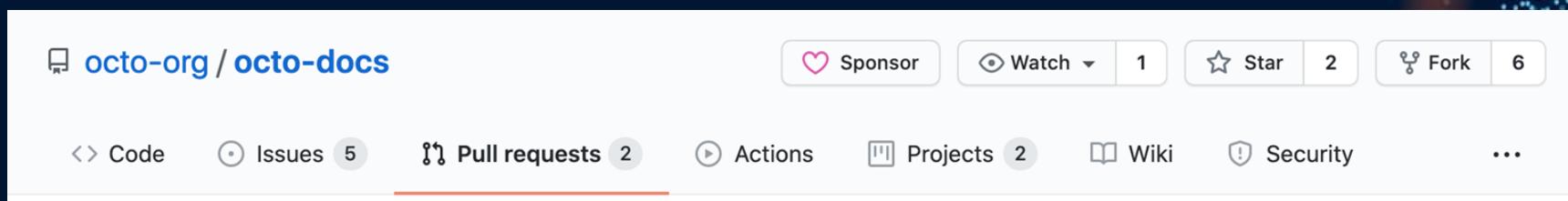


# Merge Explained

- Merge a pull request into the upstream branch when work is completed.
- Anyone with push access to the repository can complete the merge.

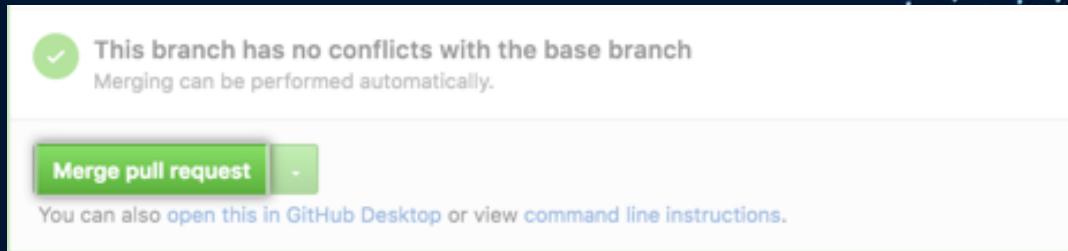
# Merging Your Pull Request

- Under your repository name, click  Pull requests.



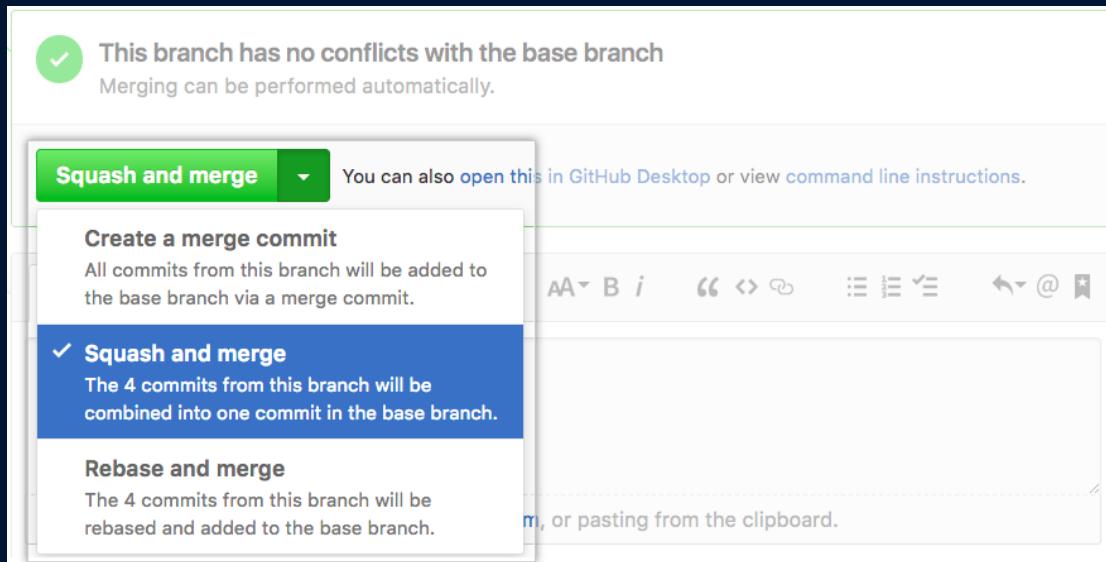
# Merging Your Pull Request

- Depending on the merge options enabled for your repository, you can:
- Merge all of the commits into the base branch by clicking Merge pull request.
- If the Merge pull request option is not shown, then click the merge drop down menu and select Create a merge commit.



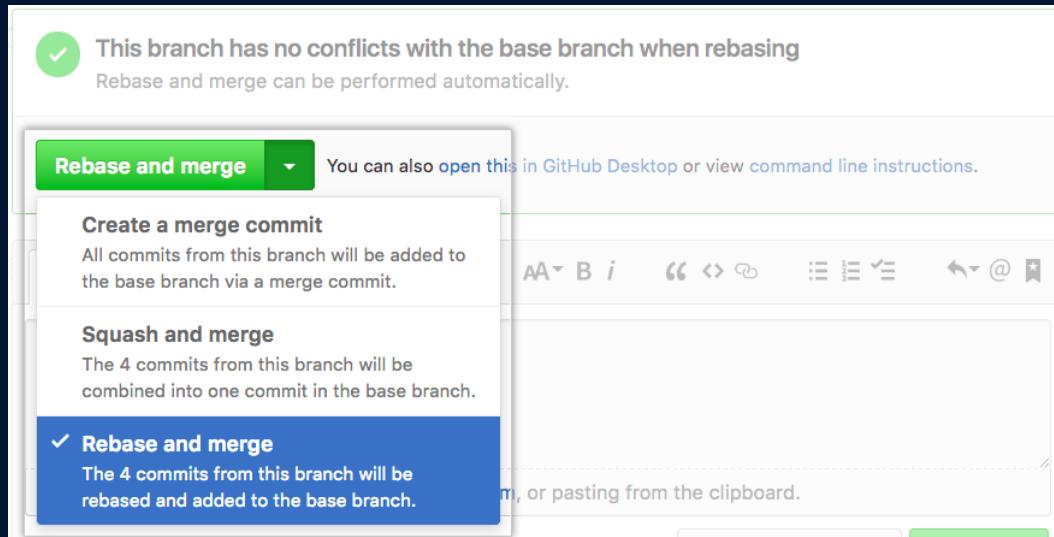
# Merging Your Pull Request

- Squash the commits into one commit by clicking the merge drop down menu, selecting Squash and merge and then clicking the Squash and merge button.



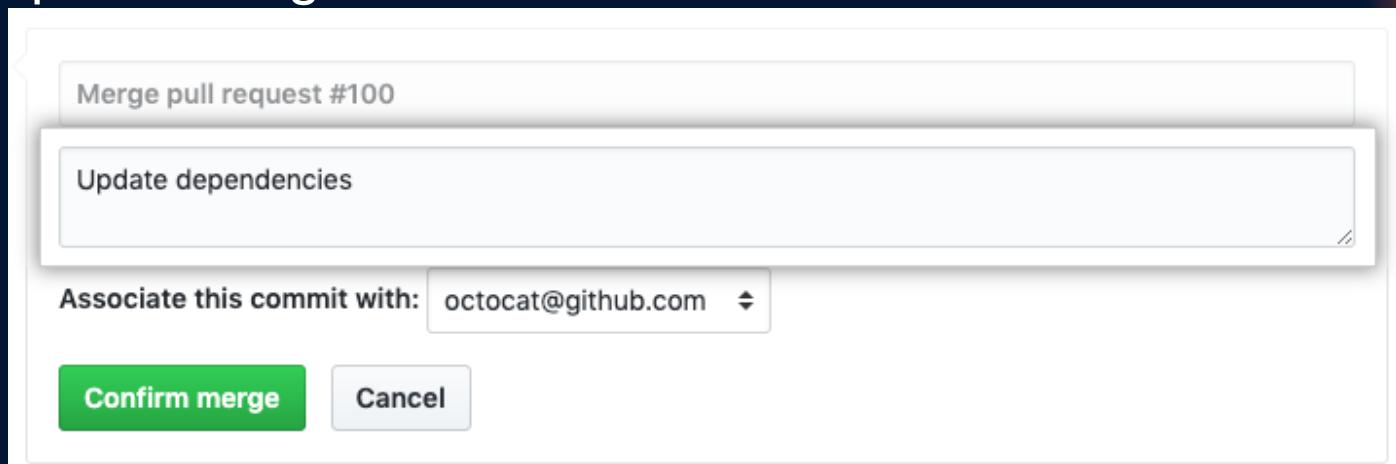
# Merging Your Pull Request

- Rebase the commits individually onto the base branch by clicking the merge drop down menu, selecting Rebase and merge and then clicking the Rebase and merge button.



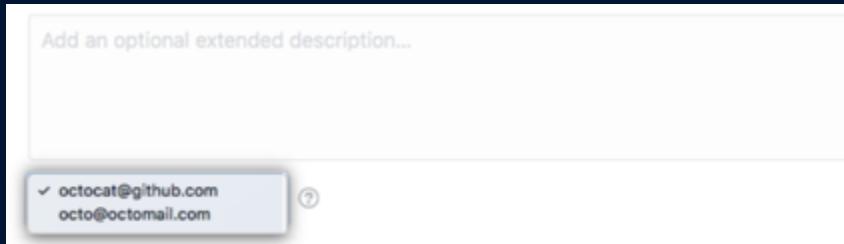
# Merging Your Pull Request

- If prompted, type a commit message, or accept the default message.
- For information about the default commit messages for squash merges



# Merging Your Pull Request

- If you have more than one email address associated with your account on GitHub.com, click the email address drop-down menu and select the email address to use as the Git author email address.
- Only verified email addresses appear in this drop-down menu.
- If you enabled email address privacy, then <username>@users.noreply.github.com is the default commit author email address.



# Pull Request and Issues

# Create a pull request

octocat commented on Mar 25 • edited

Member ...

## Why

Closes #43

Adds a Hide images / Show images toggle button

@octo-org/octo-team What do you think? Should the default be images displayed or images hidden?

## What is changing

- Users now have a button to show/hide images on the page
- When images are hidden, a placeholder icon is displayed
- The user's image preference will follow them to future pages

## Example

Check out [this page in staging](#) for a nice example (screenshot below):

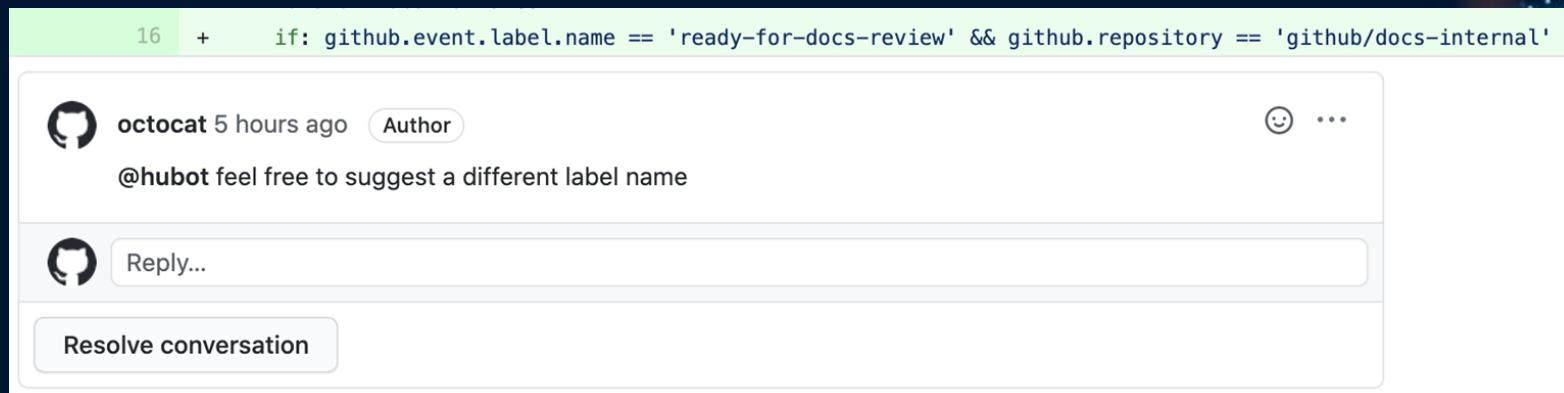
- In the upper-right corner of any page, click your profile photo, then click **Settings**.
- Under Profile Picture, click **Edit**.
- Click **Upload a photo....**
- Crop your picture. When you're done, click **Set new profile picture**.

**Images are off, click to show**

^ &

# Create a pull request

- In addition to filling out the body of the pull request, you can add comments to specific lines of the pull request to explicitly point something out to the reviewers.



# Address review comments

- Reviewers should leave questions, comments, and suggestions.
- Reviewers can comment on the whole pull request or add comments to specific lines.
- You and reviewers can insert images or code suggestions to clarify comments.

# Merge your pull request

- Once your pull request is approved, merge your pull request.
- This will automatically merge your branch so that your changes appear on the default branch.
- GitHub retains the history of comments and commits in the pull request to help future contributors understand your changes.
- GitHub will tell you if your pull request has conflicts that must be resolved before merging.

COMPLETE

Activity: Creating a New File

# Resolving Merge Conflicts



# Merge Conflicts

- Merge conflicts occur when competing changes are made to the same line of a file, or when one person edits a file and another person deletes the same file.
- To resolve a merge conflict caused by competing line changes, you must choose which changes to incorporate from the different branches in a new commit.

# Local Merge Conflicts

- Open Git Bash.
- Navigate into the local Git repository that has the merge conflict.

```
cd REPOSITORY-NAME
```

# Local Merge Conflicts

```
$ git status
> # On branch branch-b
> # You have unmerged paths.
> #   (fix conflicts and run "git commit")
> #
> # Unmerged paths:
> #   (use "git add ..." to mark resolution)
> #
> # both modified:    styleguide.md
> #
> no changes added to commit (use "git add" and/or "git commit -a")
```

# Local Merge Conflicts

- In this example, one person wrote "open an issue" in the base or HEAD branch and another person wrote "ask your question in IRC" in the compare branch or branch-a.
- If you have questions, please

```
<<<<<< HEAD
```

```
open an issue
```

```
=====
```

```
ask your question in IRC.
```

```
>>>>> branch-a
```

# Local Merge Conflicts

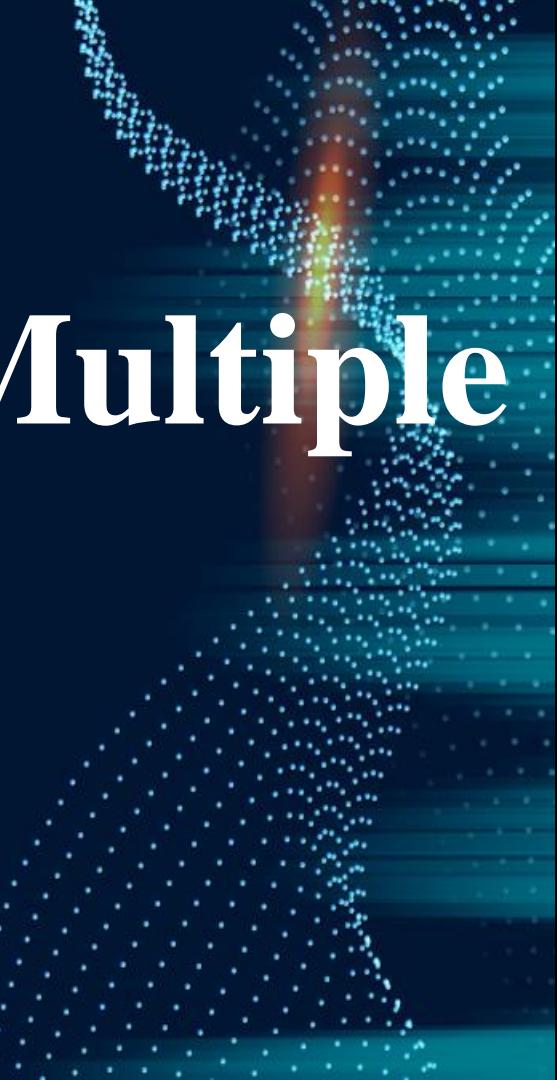
- Add or stage your changes.

```
$ git add .
```

- Commit your changes with a comment.

```
$ git commit -m "Resolved merge conflict by incorporating  
both suggestions."
```

# 12. Working with Multiple Conflicts

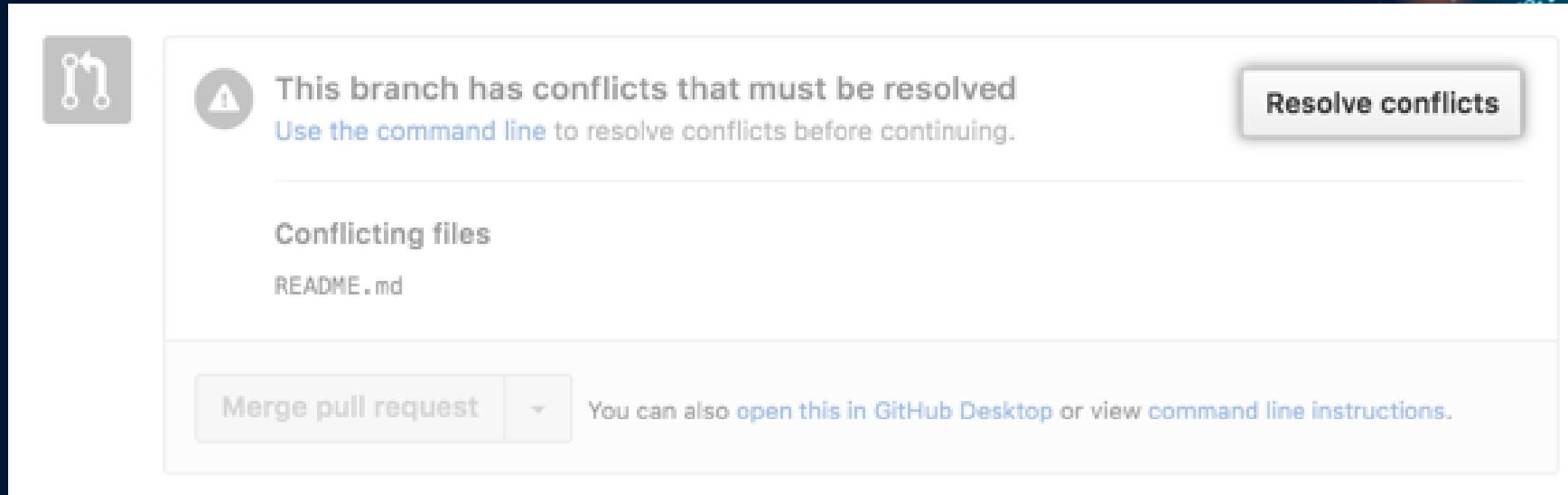


# Remote Merge Conflicts

- You can resolve simple merge conflicts that involve competing line changes on GitHub, using the conflict editor.
- You can only resolve merge conflicts on GitHub that are caused by competing line changes, such as when people make different changes to the same line of the same file on different branches in your Git repository.

# Exploring

- Near the bottom of your pull request, click Resolve conflicts.



# Exploring

Resolving conflicts between `add-emoji` and `master` and committing changes ➔ `add-emoji`

1 conflicting file	README.md	1 conflict	Prev ▲	Next ▼	⚙️	Mark as resolved
 README.md README.md	<pre>1 Octo-Repo 2 ===== 3 4 Contact @octo-org/core for questions about this repository. 5 6 ### Installing 7 8 Instructions for Installing! 9 10 #### Contributing 11 12 See the <a href="#">[CONTRIBUTING file]</a>(../CONTRIBUTING) for guidance on contributing to this project. 13 14 #### Authors 15 16 &lt;&lt;&lt;&lt; add-emoji 17 We're all authors :star: :zap: :sparkles: 18 ===== 19 We're all authors :star: :zap: :heart: 20 &gt;&gt;&gt;&gt; master 21 22 This commit will be verified! 23</pre>	1 conflict	Prev ▲	Next ▼	⚙️	Mark as resolved

# Exploring

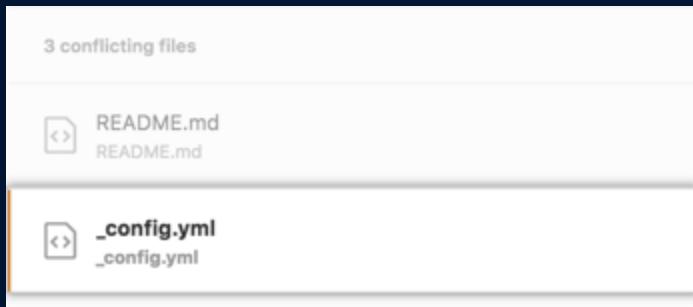
- Once you've resolved all the conflicts in the file, click Mark as resolved.

Resolving conflicts between `add-emoji` and `master` and committing changes ➔ `add-emoji`

1 conflicting file	README.md	1 conflict	Prev ⌂	Next ⌄	⚙️	Mark as resolved
 README.md README.md	1 Octo-Repo 2 ===== 3 4 Contact @octo-org/core for questions about this repository.					

# Exploring

- If you have more than one file with a conflict, select the next file you want to edit on the left side of the page under "conflicting files" and repeat steps four through seven until you've resolved all of your pull request's merge conflicts.



# Exploring

- Once you've resolved all your merge conflicts, click Commit merge.
- This merges the entire base branch into your head branch.

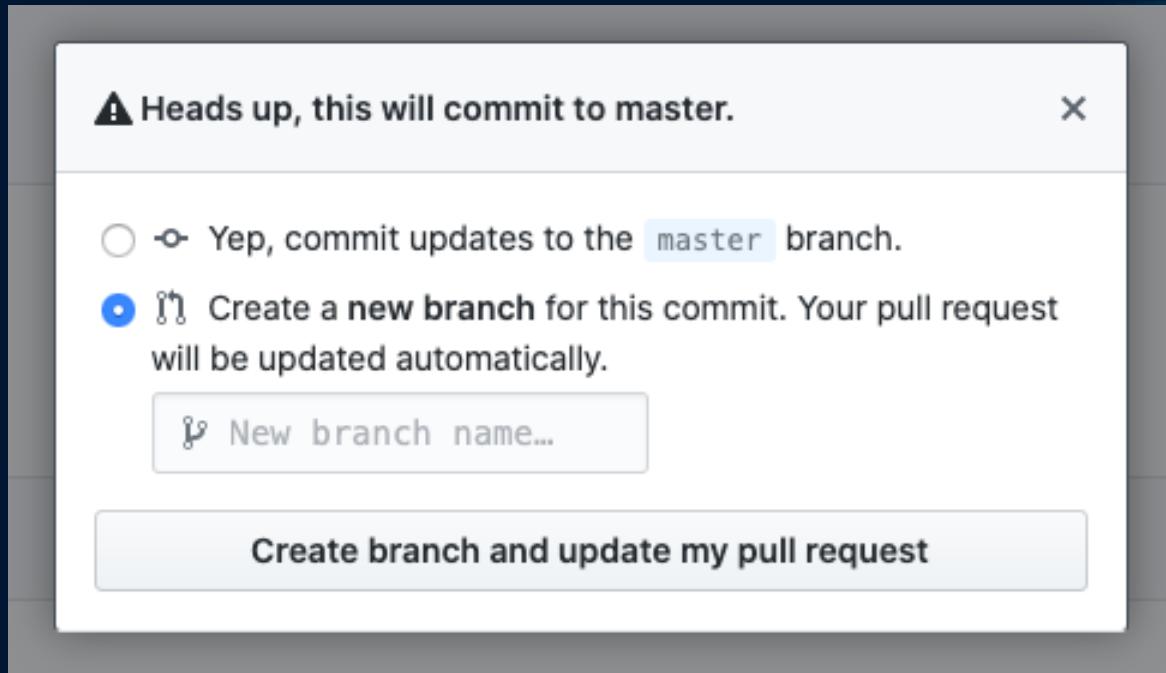
Add .sparkies. #00

Resolving conflicts between `add-emoji` and `master` and committing changes ➔ `add-emoji`

**Commit merge**

1 conflicting file	README.md	✓ Resolved
<code>README.md</code>	<pre>1 Octo-Repo 2 ===== 3 4 Contact @Octo-org/core for questions about this repository.</pre>	

# Exploring



# Staging area, working tree, and HEAD commit

- Let's focus on this right now; move to the master branch, if not already there, then type the git status command; it allows us to see the actual status of the staging area:

```
[1] ~/grocery (master)
$ git status
On branch master
nothing to commit, working tree clean
```

# Staging area, working tree, and HEAD commit

- Git says there's nothing to commit, our working tree is clean.
- But what's a working tree? Is it the same as the working directory we talked about? Well, yes and no, and it's confusing, I know.
- Git had (and still have) some troubles with names; in fact, as we said a couple of lines before, even for the staging area we have two names (the other one is index).

# Staging area, working tree, and HEAD commit

- Add a peach to the shoppingList.txt file:

```
[2] ~/grocery (master)
```

```
$ echo "peach" >> shoppingList.txt
```

- Then make use of this new learnt command again, git status:

```
[3] ~/grocery (master)
```

```
$ git status
```

```
On branch master
```

```
Changes not staged for commit:
```

```
  (use "git add <file>..." to update what will be committed)
```

```
  (use "git checkout -- <file>..." to discard changes in working directory)
```

```
modified:   shoppingList.txt
```

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

# Staging area, working tree, and HEAD commit

- Let's try to add it; we will see the second option later.
- So, try a git add command, with nothing more:

```
[4] ~/grocery (master)
```

```
$ git add
```

Nothing specified, nothing added.

Maybe you wanted to say 'git add .'?



# Staging area, working tree, and HEAD commit

- Git version 1.x:

	New files	Modified files	Deleted files	
git add -A	yes	yes	yes	Stage all (new, modified, deleted) files
git add .	yes	yes	no	Stage new and modified files only
git add -u	no	yes	yes	Stage modified and deleted files only

# Staging area, working tree, and HEAD commit

- Another basic usage is to specify the file we want to add; let's give it a try:

```
[5] ~/grocery (master)  
$ git add shoppingList.txt
```

# Staging area, working tree, and HEAD commit

- Okay, time to look back at our repository; go with a git status now:

```
[6] ~/grocery (master)
```

```
$ git status
```

```
On branch master
```

```
Changes to be committed:
```

```
(use "git reset HEAD <file>..." to unstage)
```

```
modified: shoppingList.txt
```

# Staging area, working tree, and HEAD commit

- For now, leave things how they are, and do a commit:

```
[7] ~/grocery (master)
```

```
$ git commit -m "Add a peach"
```

```
[master 603b9d1] Add a peach
```

```
 1 file changed, 1 insertion(+)
```

- Check the status:

```
[8] ~/grocery (master)
```

```
$ git status
```

```
On branch master
```

```
nothing to commit, working tree clean
```

# Staging area, working tree, and HEAD commit

- So, follow me and make things more interesting; add an onion to the shopping list and then add it to the staging area, and then add a garlic and see what happens:

```
[9] ~/grocery (master)
$ echo "onion" >> shoppingList.txt

[10] ~/grocery (master)
$ git add shoppingList.txt

[11] ~/grocery (master)
$ echo "garlic" >> shoppingList.txt

[12] ~/grocery (master)
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   shoppingList.txt

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

    modified:   shoppingList.txt
```

# Staging area, working tree, and HEAD commit

- If you want to see the difference between the working tree version and the staging area one, try to input only the git diff command without any option or argument:

```
[13] ~/grocery (master)
$ git diff
diff --git a/shoppingList.txt b/shoppingList.txt
index f961a4c..20238b5 100644
--- a/shoppingList.txt
+++ b/shoppingList.txt
@@ -3,3 +3,4 @@ apple
orange
peach
onion
+garlic
```

# Staging area, working tree, and HEAD commit

- We have to use the git diff --cached HEAD command:

```
[14] ~/grocery (master)
$ git diff --cached HEAD
diff --git a/shoppingList.txt b/shoppingList.txt
index 175eeef..f961a4c 100644
--- a/shoppingList.txt
+++ b/shoppingList.txt
@@ -2,3 +2,4 @@ banana
apple
orange
peach
+onion
```

# Staging area, working tree, and HEAD commit

- The last experiment that we can do is compare the HEAD version with the working tree one; let's do it with a git diff HEAD:

```
[15] ~/grocery (master)
$ git diff HEAD
diff --git a/shoppingList.txt b/shoppingList.txt
index 175eeef..20238b5 100644
--- a/shoppingList.txt
+++ b/shoppingList.txt
@@ -2,3 +2,5 @@ banana
apple
orange
peach
+onion
+garlic
```

# Staging area, working tree, and HEAD commit

- The following figure draws three areas of Git:



# Staging area, working tree, and HEAD commit

- Check the repository current status:

```
[16] ~/grocery (master)
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   shoppingList.txt

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

    modified:   shoppingList.txt
```

# Staging area, working tree, and HEAD commit

- This is the actual situation, remember? We have an onion in the staging area and a garlic more in the working tree.
- Now go with a git reset HEAD:

```
[17] ~/grocery (master)
$ git reset HEAD shoppingList.txt
Unstaged changes after reset:
M  shoppingList.txt
```

# Staging area, working tree, and HEAD commit

- Well, time to verify what happened:

```
[18] ~/grocery (master)$ git status
On branch master
Changes not staged for commit:

(use "git add <file>..." to update what will be committed) (use "git checkout -- <file>..." to
discard changes in working directory)
modified:   shoppingList.txt no changes added to commit (use
"git add" and/or "git commit -a")
```

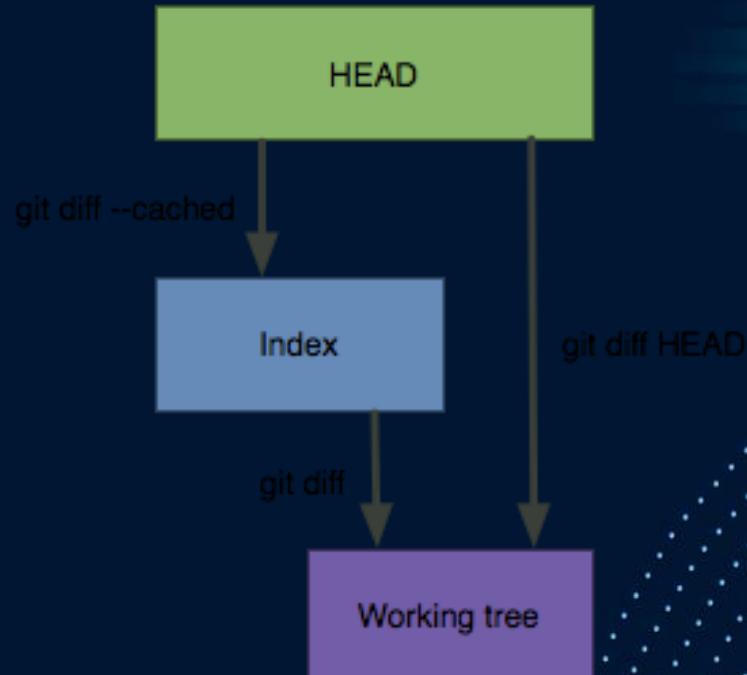
# Staging area, working tree, and HEAD commit

```
[19] ~/grocery (master)
$ git diff
diff --git a/shoppingList.txt b/shoppingList.txt
index 175eeef..20238b5 100644
--- a/shoppingList.txt
+++ b/shoppingList.txt
@@ -2,3 +2,5 @@ banana
apple
orange
peach
+onion
+garlic
```

- Let's verify this using the git diff command:

# Staging area, working tree, and HEAD commit

- The following figure shows a quick summary of git diff different behaviors:



# Staging area, working tree, and HEAD commit

- The command for that is `git checkout -- <file>`, as Git gently reminds in the `git status` output message.
- Give it a try:

```
[20] ~/grocery (master)
$ git checkout -- shoppingList.txt
```

- Check the status:

```
[21] ~/grocery (master)
$ git status
On branch master
nothing to commit, working tree clean
```

# Staging area, working tree, and HEAD commit

- Check the content of the file:

```
[22] ~/grocery (master)
```

```
$ cat shoppingList.txt
```

```
banana
```

```
apple
```

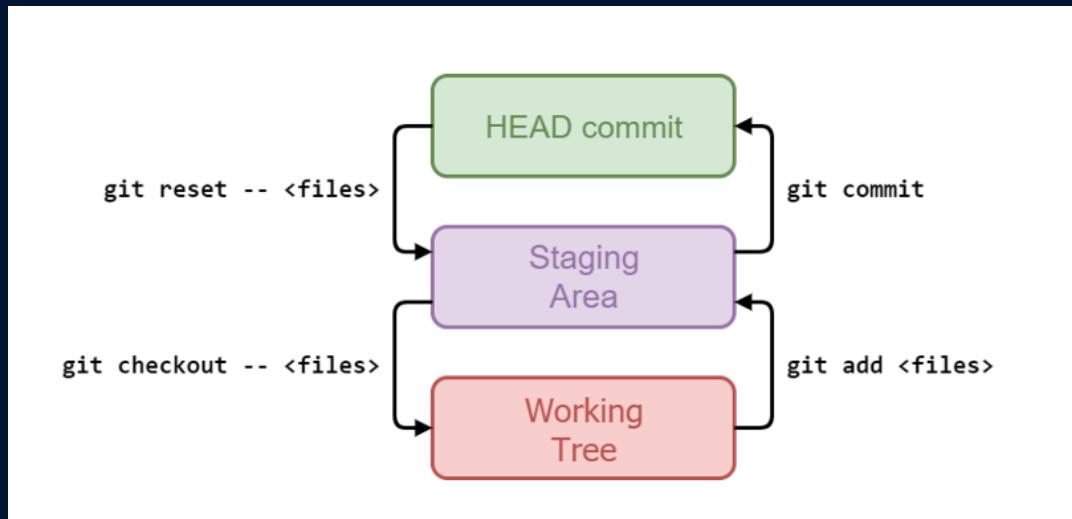
```
orange
```

```
peach
```



# Staging area, working tree, and HEAD commit

- The following figure summarizes the commands to move changes between those three areas:



# Staging area, working tree, and HEAD commit

## File status lifecycle

- In a Git repository, files pass through some different states.
- When you first create a file in the working tree, Git notices it and tells you there's a new untracked file.
- Let's try to create a new file.txt in our grocery repository and see the output of git status:

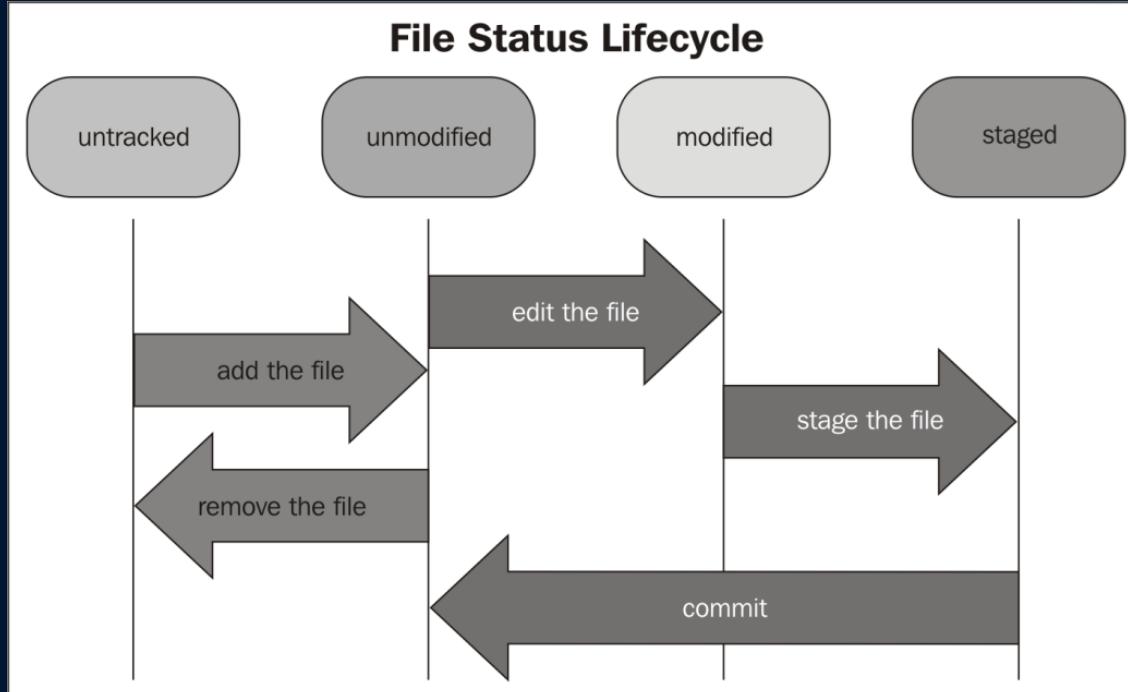
```
[23] ~/grocery (master)
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    file.txt

nothing added to commit but untracked files present (use "git add" to track)
```

# Staging area, working tree, and HEAD commit

- The following figure summarizes these states:



# Staging area, working tree, and HEAD commit

## All you need to know about checkout and reset

- First of all, we need to do some housekeeping.
- Go back to the grocery repository and clean up the working tree; double-check that you are in the master branch, and then do a git reset --hard master:

```
[24] ~/grocery (master)
```

```
$ git reset --hard master
```

```
HEAD is now at 603b9d1 Add a peach
```

# Staging area, working tree, and HEAD commit

- Then, delete the bug branch we created some time ago; the command to delete a branch is again the git branch command, this time followed by a -d option and then the branch name:

```
[25] ~/grocery (master)
```

```
$ git branch -d bug
```

```
error: The branch 'bug' is not fully merged.
```

```
If you are sure you want to delete it, run 'git branch -D bug'.
```

# Staging area, working tree, and HEAD commit

- No problem, we don't need that commit; so, use the capital -D option to force the deletion:

```
[26] ~/grocery (master)
```

```
$ git branch -D bug
```

Deleted branch bug (was 07b1858).

- Okay, now we are done, and the repository is in good shape, as the git log command shows:

```
[27] ~/grocery (master)
$ git log --oneline --graph --decorate --all
* 603b9d1 (HEAD -> master) Add a peach
| * a8c6219 (melons) Add a watermelon
| * ef6c382 (berries) Add a blackberry
|/
* 0e8b5cf Add an orange
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```

# Staging area, working tree, and HEAD commit

- Git checkout overwrites all the tree areas
- Now switch to the melons branch using the git checkout command:

```
[28] ~/grocery (master)
$ git checkout melons
Switched to branch 'melons'
```

- Check the log:

```
[29] ~/grocery (melons)
$ git log --oneline --graph --decorate --all
* 603b9d1 (master) Add a peach
| * a8c6219 (HEAD -> melons) Add a watermelon
| * ef6c382 (berries) Add a blackberry
|/
* 0e8b5cf Add an orange
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```

# Staging area, working tree, and HEAD commit

- We can try it; add a potato to the shopping list file:

```
[30] ~/grocery (melons)
```

```
$ echo "potato" >> shoppingList.txt
```

- Then checkout master:

```
[31] ~/grocery (melons)
```

```
$ git checkout master
```

```
error: Your local changes to the following files would be overwritten by checkout:
```

```
shoppingList.txt
```

```
Please commit your changes or stash them before you switch branches.
```

```
Aborting
```

# Staging area, working tree, and HEAD commit

- To avoid messing up our repo again, go into a detached HEAD state, so at the end it will be easier to throw all the things away.
- To do this, checkout directly the penultimate commit on the master branch:

```
[32] ~/grocery (master)
$ git checkout HEAD~1
Note: checking out 'HEAD~1'.
```

You are in 'detached HEAD' state. You can look around, make experimental changes and commit them, and you can discard any commits you make in this state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may do so (now or later) by using -b with the checkout command again. Example:

```
git checkout -b <new-branch-name>
```

```
HEAD is now at 0e8b5cf... Add an orange
```

# Staging area, working tree, and HEAD commit

- Now just replicate the onion and garlic situation we used before: append an onion to the file and add it to the staging area, and then add a garlic:

```
[34] ~/grocery ((0e8b5cf...))
$ echo "onion" >> shoppingList.txt

[35] ~/grocery ((0e8b5cf...))
$ git add shoppingList.txt

[36] ~/grocery ((0e8b5cf...))
$ echo "garlic" >> shoppingList.txt

[37] ~/grocery ((0e8b5cf...))
$ git status
HEAD detached at 0e8b5cf
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   shoppingList.txt

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

    modified:   shoppingList.txt
```

# Staging area, working tree, and HEAD commit

- Now use the git diff command to be sure we are in the situation we desire; check the differences with the staging area:

```
[38] ~/grocery ((0e8b5cf...))
$ git diff --cached HEAD
diff --git a/shoppingList.txt b/shoppingList.txt
index edc9072..063aa2f 100644
--- a/shoppingList.txt
+++ b/shoppingList.txt
@@ -1,3 +1,4 @@
banana
apple
orange
+onion
```

# Staging area, working tree, and HEAD commit

- Check the differences between the working tree and HEAD commit:

```
[39] ~/grocery ((0e8b5cf...))
$ git diff HEAD
diff --git a/shoppingList.txt b/shoppingList.txt
index edc9072..93dcf0e 100644
--- a/shoppingList.txt
+++ b/shoppingList.txt
@@ -1,3 +1,5 @@
banana
apple
orange
+onion
+garlic
```

# Staging area, working tree, and HEAD commit

- Now try to do a soft reset to the master branch, with the git reset --soft master command:

```
[40] ~/grocery ((0e8b5cf...))  
$ git reset --soft master
```

- Diff to the staging area:

```
[41] ~/grocery ((603b9d1...))  
$ git diff --cached HEAD  
diff --git a/shoppingList.txt b/shoppingList.txt  
index 175eeeef..063aa2f 100644  
--- a/shoppingList.txt  
+++ b/shoppingList.txt  
@@ -1,4 +1,4 @@  
banana  
apple  
orange  
-peach  
+onion
```

# Staging area, working tree, and HEAD commit

- The same is if you compare the HEAD commit with a working tree:

```
[42] ~/grocery ((603b9d1...))
$ git diff HEAD
diff --git a/shoppingList.txt b/shoppingList.txt
index 175eeeef..93dcf0e 100644
--- a/shoppingList.txt
+++ b/shoppingList.txt
@@ -1,4 +1,5 @@
banana
apple
orange
-peach
+onion
+garlic
```

# Staging area, working tree, and HEAD commit

- Another option is the mixed reset; you can do it using the --mixed option (or simply using no options, as this is the default):

```
[43] ~/grocery ((603b9d1...))  
$ git reset --mixed master  
Unstaged changes after reset:  
M shoppingList.txt
```

- Okay, there's something different here: Git tells us about unstaged changes.
- In fact, the --mixed option makes Git overwrite even the staging area, not only the HEAD commit.

# Staging area, working tree, and HEAD commit

- If you check differences between the HEAD commit and staging area with git diff, you will see that there are no differences:

```
[44] ~/grocery ((603b9d1...))  
$ git diff --cached HEAD
```

# Staging area, working tree, and HEAD commit

- Instead, differences arise between the HEAD commit and working tree:

```
[45] ~/grocery ((603b9d1...))
$ git diff HEAD
diff --git a/shoppingList.txt b/shoppingList.txt
index 175eeef..93dcf0e 100644
--- a/shoppingList.txt
+++ b/shoppingList.txt
@@ -1,4 +1,5 @@
banana
apple
orange
-peach
+onion
+garlic
```

# Staging area, working tree, and HEAD commit

- At this point, you can presume what is the purpose of the --hard option: it overwrites all the three areas:

```
[46] ~/grocery ((603b9d1...))
```

```
$ git reset --hard master
```

```
HEAD is now at 603b9d1 Add a peach
```

```
[47] ~/grocery ((603b9d1...))
```

```
$ git diff --cached HEAD
```

```
[48] ~/grocery ((603b9d1...))
```

```
$ git diff HEAD
```

# Staging area, working tree, and HEAD commit

- Now we know a little more about both the git checkout and git reset command; but before leaving, go back in a non-detached HEAD state, checking out the master branch:

```
[49] ~/grocery ((603b9d1...))  
$ git checkout master  
Switched to branch 'master'
```

# Advanced Troubleshooting

# Creating a New Local Repository



# Initializing a new local repository

## Start a new git repository

- Your first instinct, when you start to do something new, should be git init.
- You're starting to write a new paper, you're writing a bit of code to do a computer simulation, you're messing around with some new data ... anything: think git init.

# Initializing a new local repository

Say you've just got some data from a collaborator and are about to start exploring it.

- Create a directory to contain the project.
- Go into the new directory.
- Type `git init`.
- Write some code.
- Type `git add` to add the files (see the typical use page).
- Type `git commit`.



# A new repo from an existing project

Say you've got an existing project that you want to start tracking with git.

- Go into the directory containing the project.
- Type `git init`.
- Type `git add` to add all of the relevant files.
- You'll probably want to create a `.gitignore` file right away, to indicate all of the files you don't want to track. Use `git add .gitignore`, too.
- Type `git commit`.

# Connect it to Azure DevOps / GitHub / Remote

You've now got a local git repository. You can use git locally, like that, if you want. But if you want the thing to have a home on github, do the following.

- Go to github.
- Log in to your account.
- Click the new repository button in the top-right. You'll have an option there to initialize the repository with a README file, but I don't.
- Click the "Create repository" button.

# Initializing a new local repository

- Now, follow the second set of instructions, “Push an existing repository...”

```
$ git remote add origin git@github.com:username/new_repo  
$ git push -u origin master
```

- Actually, the first line of the instructions will say

```
$ git remote add origin  
https://github.com/username/new\_repo
```

# Fixing Commit Mistakes



# Revising your last commit

- You have a commit history like this one (older commits on top):

f7f3f6d create file\_a class and methods

310154e improve SomeOtherClass code

a5f4a0d update changelog file

- But you forgot to add just a single line on the changelog file you mentioned on the commit a5f4a0d :(

# Revising your last commit

- You could simply commit it and have a history like this:

```
f7f3f6d create file_a class and methods  
310154e improve SomeOtherClass code  
a5f4a0d update changelog file  
a4gx124 add missing line to changelog
```

# Revising your last commit

## The Basic of the Amend Command

- Just add the modified file(s):

```
$ (some_branch) git add changelog.md
```

- And amend it:

```
$ (some_branch) git commit --amend
```



# Revising your last commit

```
update changelog file

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# Author: John Doe <john@doe.com>
# Date: Thu Apr 21 22:40:30 2016 -0300
#
# On branch some_branch
# Your branch is up-to-date with 'origin/some_branch'.
#
# Changes to be committed:
# modified: changelog.md
```

# Amending a Commit Without Changing Its Message

- If you don't want to change your commit message, you can run the amend command with the no-edit flag, like this:

```
$ (some_branch) git commit --amend --no-edit
```



# Pushing an Amended Commit

- If you haven't pushed the last commit yet to your remote, a single push is enough.
- Otherwise, you'll have to push using the -f option since you've rewritten your commit history:

```
$ (some_branch) git push -f origin some_branch
```

# Rewriting History with Git Reset



# Understanding reset

- The git reset command is a complex and versatile tool for undoing changes. It has three primary forms of invocation.
- These forms correspond to command line arguments --soft, --mixed, --hard.
- The three arguments each correspond to Git's three internal state management mechanism's, The Commit Tree (HEAD), The Staging Index, and The Working Directory.

# Git Reset & Three Trees of Git

- To get started we will create a new repository with the commands below:

```
$ mkdir git_reset_test
$ cd git_reset_test/
$ git init .
Initialized empty Git repository in /git_reset_test/.git/
$ touch reset_lifecycle_file
$ git add reset_lifecycle_file
$ git commit -m"initial commit"
[main (root-commit) d386d86] initial commit
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 reset_lifecycle_file
```

# The working directory

- In our demo repository, we modify and add some content to the `reset_lifecycle_file`.
- Invoking `git status` shows that Git is aware of the changes to the file.

```
$ echo 'hello git reset' > reset_lifecycle_file
```

```
$ git status
```

On branch main

Changes not staged for commit:

(use "git add ..." to update what will be committed)

(use "git checkout -- ..." to discard changes in working directory)

modified: `reset_lifecycle_file`

# Reset Modes

- In addition to updating the commit ref pointers, git reset will modify the state of the three trees.
- The ref pointer modification always happens and is an update to the third tree, the Commit tree.
- The command line arguments --soft, --mixed, and --hard direct how to modify the Staging Index, and Working Directory trees.

# Reset Soft

```
$ git add reset_lifecycle_file
```

```
$ git ls-files -s
```

```
100644 67cc52710639e5da6b515416fd779d0741e3762e 0
```

```
reset_lifecycle_file
```

```
$ git status
```

```
On branch main
```

```
Changes to be committed:
```

```
(use "git reset HEAD ..." to unstage)
```

```
modified: reset_lifecycle_file
```

```
Untracked files:
```

```
(use "git add ..." to include in what will be committed)
```

```
new_file
```

- When the --soft argument is passed, the ref pointers are updated and the reset stops there.
- The Staging Index and the Working Directory are left untouched. This behavior can be hard to clearly demonstrate.

# Reset Soft

- With the repository in this state we now execute a soft reset.

```
$ git reset --soft  
$ git status  
On branch main  
Changes to be committed:  
(use "git reset HEAD ..." to unstage)
```

```
modified: reset_lifecycle_file  
$ git ls-files -s  
100644 67cc52710639e5da6b515416fd779d0741e3762e 0  
reset_lifecycle_file
```

- The Staging Index is reset to the state of the specified commit.
- Any changes that have been undone from the Staging Index are moved to the Working Directory.

```
$ echo 'new file content' > new_file  
$ git add new_file  
$ echo 'append content' >> reset_lifecycle_file  
$ git add reset_lifecycle_file  
$ git status  
On branch main  
Changes to be committed:  
  (use "git reset HEAD ..." to unstage)  
new file: new_file  
modified: reset_lifecycle_file  
$ git ls-files -s  
100644 8e66654a5477b1bf4765946147c49509a431f963 0 new_file  
100644 7ab362db063f9e9426901092c00a3394b4bec53d 0 reset_lifecycle_file
```

## Reset Mixed

- The Staging Index has been reset and the pending changes have been moved into the Working Directory.
- Compare this to the --hard reset case where the Staging Index was reset and the Working Directory was reset as well, losing these updates.

```
$ git reset --mixed
```

```
$ git status
```

On branch main

Changes not staged for commit:

(use "git add ..." to update what will be committed)

(use "git checkout -- ..." to discard changes in working directory)

modified: reset\_lifecycle\_file

Untracked files:

(use "git add ..." to include in what will be committed)

new\_file

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

```
$ git ls-files -s
```

```
100644 d7d77c1b04b5edd5acfc85de0b592449e5303770 0 reset_lifecycle_file
```

# Reset Hard

- To demonstrate this, let's continue with the three tree example repo we established earlier.
- First let's make some modifications to the repo, Execute the following commands in the example repo:

```
$ echo 'new file content' > new_file
```

```
$ git add new_file
```

```
$ echo 'changed content' >> reset_lifecycle_file
```

# Reset Hard

```
$ git status
```

On branch main

Changes to be committed:

(use "git reset HEAD ..." to unstage)

new file: new\_file

Changes not staged for commit:

(use "git add ..." to update what will be committed)

(use "git checkout -- ..." to discard changes in working directory)

modified: reset\_lifecycle\_file

# Reset Hard

- Before moving forward let us also examine the state of the Staging Index:

```
$ git ls-files -s  
100644 8e66654a5477b1bf4765946147c49509a431f963 0  
new_file  
100644 d7d77c1b04b5edd5acf85de0b592449e5303770 0  
reset_lifecycle_file
```

# Reset Hard

- Let us now execute a git reset --hard and examine the new state of the repository.

```
$ git reset --hard
```

```
HEAD is now at dc67808 update content of reset_lifecycle_file
```

```
$ git status
```

```
On branch main
```

```
nothing to commit, working tree clean
```

```
$ git ls-files -s
```

```
100644 d7d77c1b04b5edd5acfc85de0b592449e5303770 0  
reset.lifecycle_file
```

# Reset Hard

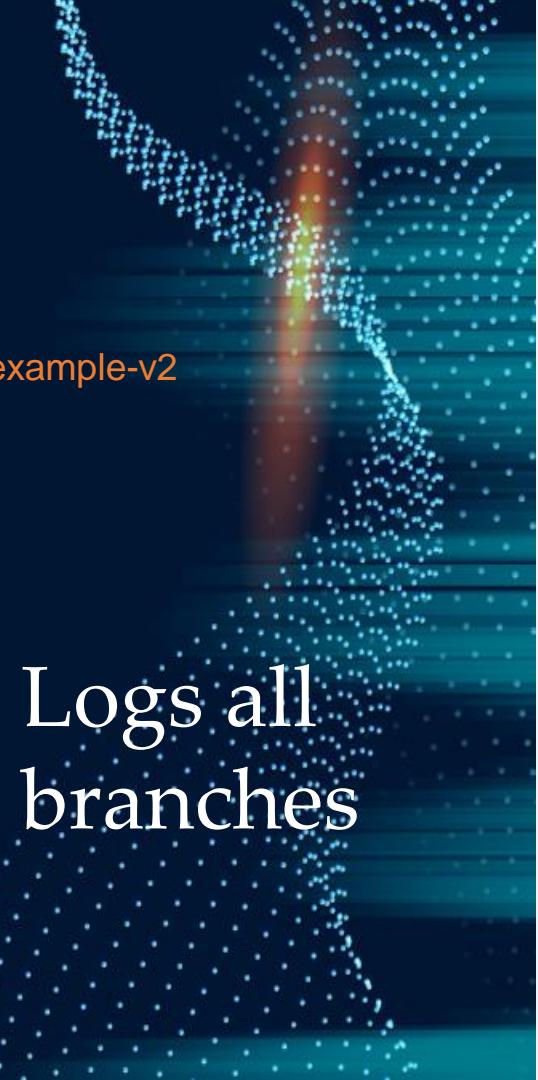
- If you haven't pushed the last commit yet to your remote, a single push is enough.
- Otherwise, you'll have to push using the -f option since you've rewritten your commit history:

```
$ (some_branch) git push -f origin some_branch
```

# 20. Getting it Back

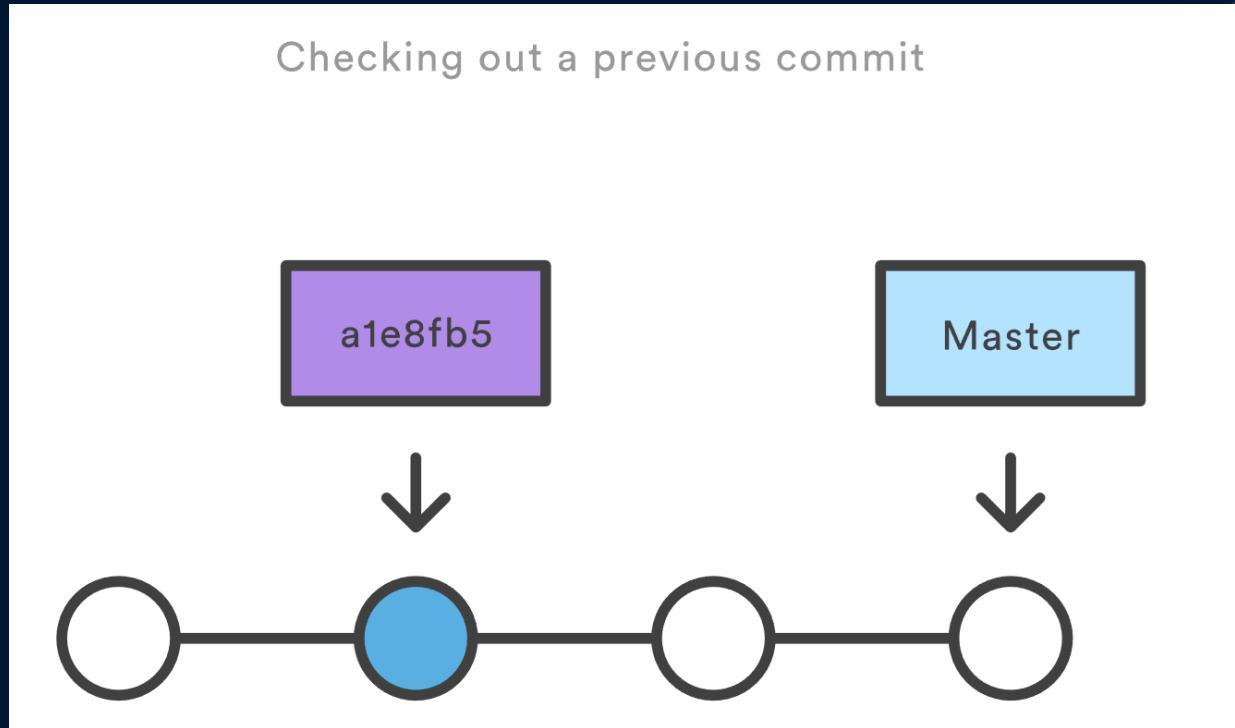


```
git log --oneline
e2f9a78fe Replaced FlyControls with OrbitControls
d35ce0178 Editor: Shortcuts panel Safari support.
9dbe8d0cf Editor: Sidebar.Controls to Sidebar.Settings.Shortcuts. Clean up.
05c5288fc Merge pull request #12612 from TyLindberg/editor-controls-panel
0d8b6e74b Merge pull request #12805 from harto/patch-1
23b20c22e Merge pull request #12801 from gam0022/improve-raymarching-example-v2
fe78029f1 Fix typo in documentation
7ce43c448 Merge pull request #12794 from WestLangley/dev-x
17452bb93 Merge pull request #12778 from OndrejSpanel/unitTestFixes
b5c1b5c70 Merge pull request #12799 from dhritzkiv/patch-21
1b48ff4d2 Updated builds.
88adbcdf6 WebVRManager: Clean up.
2720fbb08 Merge pull request #12803 from dmarcos/parentPoseObject
9ed629301 Check parent of poseObject instead of camera
219f3eb13 Update GLTFLoader.js
15f13bb3c Update GLTFLoader.js
6d9c22a3b Update uniforms only when onWindowResize
881b25b58 Update ProjectionMatrix on change aspect
```



Logs all  
branches

# You just want that one commit



# Viewing an old revision

- This example assumes that you've started developing a crazy experiment, but you're not sure if you want to keep it or not.
- To help you decide, you want to take a look at the state of the project before you started your experiment.
- First, you'll need to find the ID of the revision you want to see.

`git log --oneline`

# Viewing an old revision

- Let's say your project history looks something like the following:  
b7119f2 Continue doing crazy things  
872fa7e Try something crazy  
a1e8fb5 Make some important changes to hello.txt  
435b61d Create hello.txt  
9773e52 Initial import
- You can use git checkout to view the “Make some import changes to hello.txt” commit as follows:  
`git checkout a1e8fb5`

# Viewing an old revision

- To continue developing, you need to get back to the “current” state of your project:  
`git checkout main`
- This assumes that you're developing on the default main branch. Once you're back in the main branch, you can use either `git revert` or `git reset` to undo any undesired changes.

# How to undo a commit with git checkout

- Using the git checkout command we can checkout the previous commit, a1e8fb5, putting the repository in a state before the crazy commit happened.
- Checking out a specific commit will put the repo in a "detached HEAD" state.
- This means you are no longer working on any branch.
- In a detached state, any new commits you make will be orphaned when you change branches back to an established branch.

# How to undo a public commit with git revert

- If we execute `git revert HEAD`, Git will create a new commit with the inverse of the last commit. This adds a new commit to the current branch history and now makes it look like:

```
git log --oneline  
e2f9a78 Revert "Try something crazy"  
872fa7e Try something crazy  
a1e8fb5 Make some important changes to hello.txt  
435b61d Create hello.txt  
9773e52 Initial import
```

# How to undo a commit with git reset

- For this undo strategy we will continue with our working example. git reset is an extensive command with multiple uses and functions.
- If we invoke git reset --hard a1e8fb5 the commit history is reset to that specified commit.
- Examining the commit history with git log will now look like:

```
git log --oneline
```

```
a1e8fb5 Make some important changes to hello.txt
```

```
435b61d Create hello.txt
```

```
9773e52 Initial import
```

# Undoing the last commit

- These strategies are all applicable to the most recent commit as well.
- In some cases though, you might not need to remove or reset the last commit, Maybe it was just made prematurely.
- In this case you can amend the most recent commit.
- Once you have made more changes in the working directory and staged them for commit by using git add, you can execute git commit --amend.

# Oops, I didn't mean to reset

## Undoing uncommitted changes

- Before changes are committed to the repository history, they live in the staging index and the working directory. You may need to undo changes within these two areas.
- The staging index and working directory are internal Git state management mechanisms.
- For more detailed information on how these two mechanisms operate, visit the [git reset](#) page which explores them in depth.

# Oops, I didn't mean to reset

## Undoing public changes

- When working on a team with remote repositories, extra consideration needs to be made when undoing changes.
- Git reset should generally be considered a 'local' undo method.
- A reset should be used when undoing changes to a private branch.
- This safely isolates the removal of commits from other branches that may be in use by other developers.

# Cherry picking

- Let's play with it a little bit.
- Assume you want to pick the blackberry from the berries branch, and then apply it into the master branch; this is the way:

```
[1] ~/grocery (master)
$ git cherry-pick ef6c382
error: could not apply ef6c382... Add a blackberry
hint: after resolving the conflicts, mark the corrected paths
hint: with 'git add <paths>' or 'git rm <paths>'
hint: and commit the result with 'git commit'
```

# Cherry picking

- Okay, the cherry pick raised a conflict, of course:

```
[2] ~/grocery (master|CHERRY-PICKING)
$ git diff
diff --cc shoppingList.txt
index 862debc,b05b25f..0000000
--- a/shoppingList.txt
+++ b/shoppingList.txt
@@@ -1,5 -1,4 +1,9 @@@
banana
apple
orange
++<<<<< HEAD
+peach
-grape
++grape
=====
+ blackberry
++>>>>> ef6c382... Add a blackberry
```

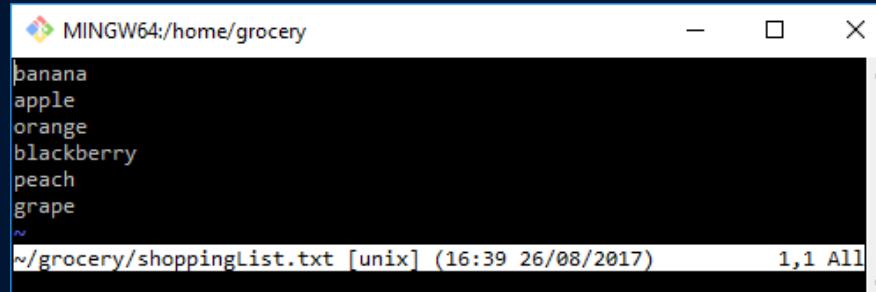
# Cherry picking

- The fourth line of both the shoppingList.txt file versions has been modified with different fruits.
- Resolve the conflict and then add a commit:

[3] ~/grocery (master|CHERRY-PICKING)

\$ vi shoppingList.txt

- The following is a screenshot of my Vim console, and the files are arranged as I like:



A screenshot of a terminal window titled "MINGW64:/home/grocery". The window displays a list of fruits in a single column:

```
banana
apple
orange
blackberry
peach
grape
~
```

The status bar at the bottom shows the file path as "~/grocery/shoppingList.txt [unix] (16:39 26/08/2017)" and the status "1,1 All".

# Cherry picking

```
[4] ~/grocery (master|CHERRY-PICKING)
$ git add shoppingList.txt

[5] ~/grocery (master|CHERRY-PICKING)
$ git status
On branch master
You are currently cherry-picking commit ef6c382.
  (all conflicts fixed: run "git cherry-pick --continue")
  (use "git cherry-pick --abort" to cancel the cherry-pick operation)

Changes to be committed:

modified:   shoppingList.txt
```

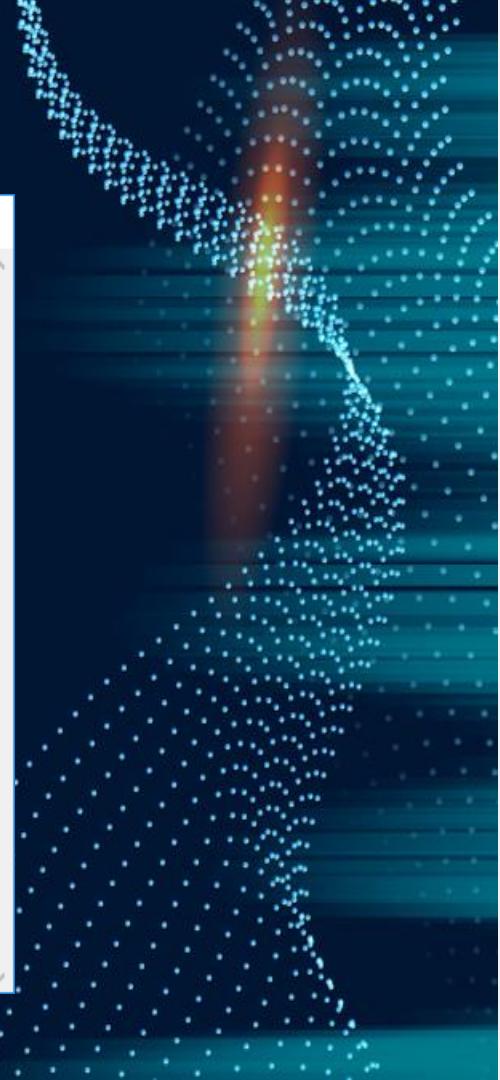
# Cherry picking

- Now go on and commit:

```
[6] ~/grocery (master)
$ git commit -m "Add a cherry-picked blackberry"
On branch master
nothing to commit, working tree clean

[7] ~/grocery (master)
$ git log --oneline --graph --decorate --all
* 99dd471 (HEAD -> master) Add a cherry-picked blackberry
* 6409527 Add a grape
* 603b9d1 Add a peach
| * a8c6219 (melons) Add a watermelon
| * ef6c382 (berries) Add a blackberry
|/
* 0e8b5cf Add an orange
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```

# Cherry picking



```
MINGW64:/home/grocery
Add a blackberry

(cherry picked from commit ef6c3821fd9a92b90bc9fc444befffc5326f4228)

# Conflicts:
#       shoppingList.txt
#
# It looks like you may be committing a cherry-pick.
# If this is not correct, please remove the file
#       .git/CHERRY_PICK_HEAD
# and try again.

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# Date:      Fri Aug 25 13:24:38 2017 +0200
#
# On branch master
# You are currently cherry-picking commit ef6c382.
#
# Changes to be committed:
#       modified:   shoppingList.txt
#
#
~/grocery/.git/COMMIT_EDITMSG [unix] (16:56 26/08/2017) 8,1 All
```

# Git Reflog

Git reflog (short for "reference log") is a Git command that keeps a record of all the actions and updates made to your repository's references (like HEAD, branches, or tags) within your local Git environment. It is particularly useful for recovering lost commits or diagnosing issues in your Git workflow.

# Git Reflog

Command: git reflog

Example output:

```
c3d947f (HEAD -> main) HEAD@{0}: commit: Fixed a  
typo in README  
5a1f98b HEAD@{1}: rebase: updated  
the feature branch  
d4f9a62 HEAD@{2}: checkout:  
moving from feature to main  
e34d7f3 HEAD@{3}:  
commit: Added a new feature
```

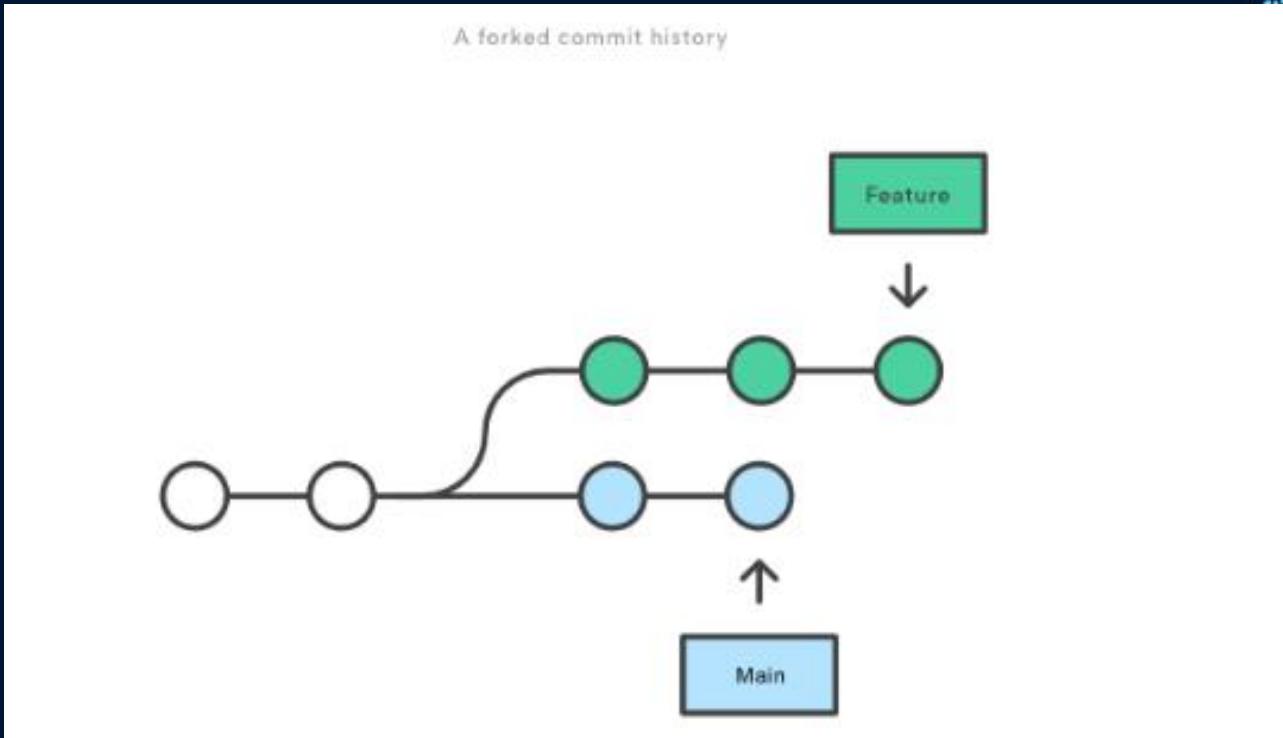
# 21. Merge Strategies: Rebase



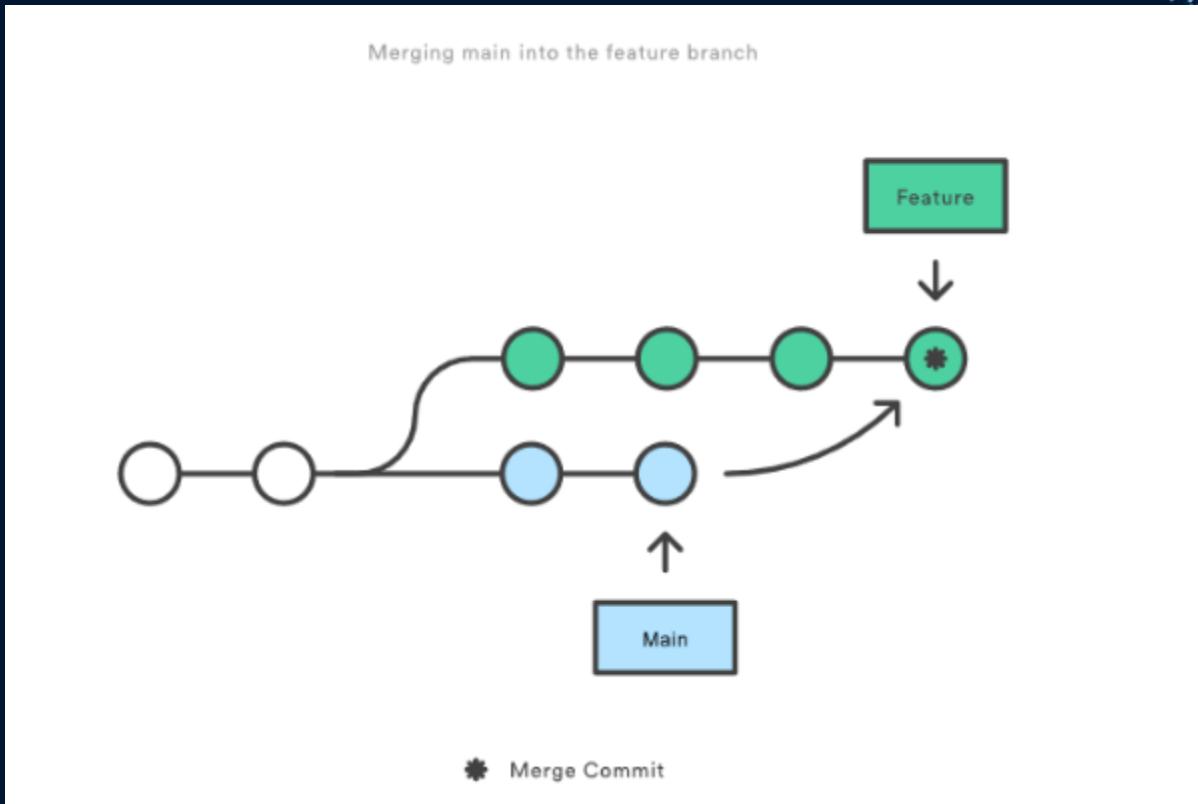
# About Git rebase

- Rebase is one of two Git utilities designed to integrate changes from one branch onto another.
- Rebasing is the process of combining or moving a sequence of commits on top of a new base commit.
- Git rebase is the linear process of merging.

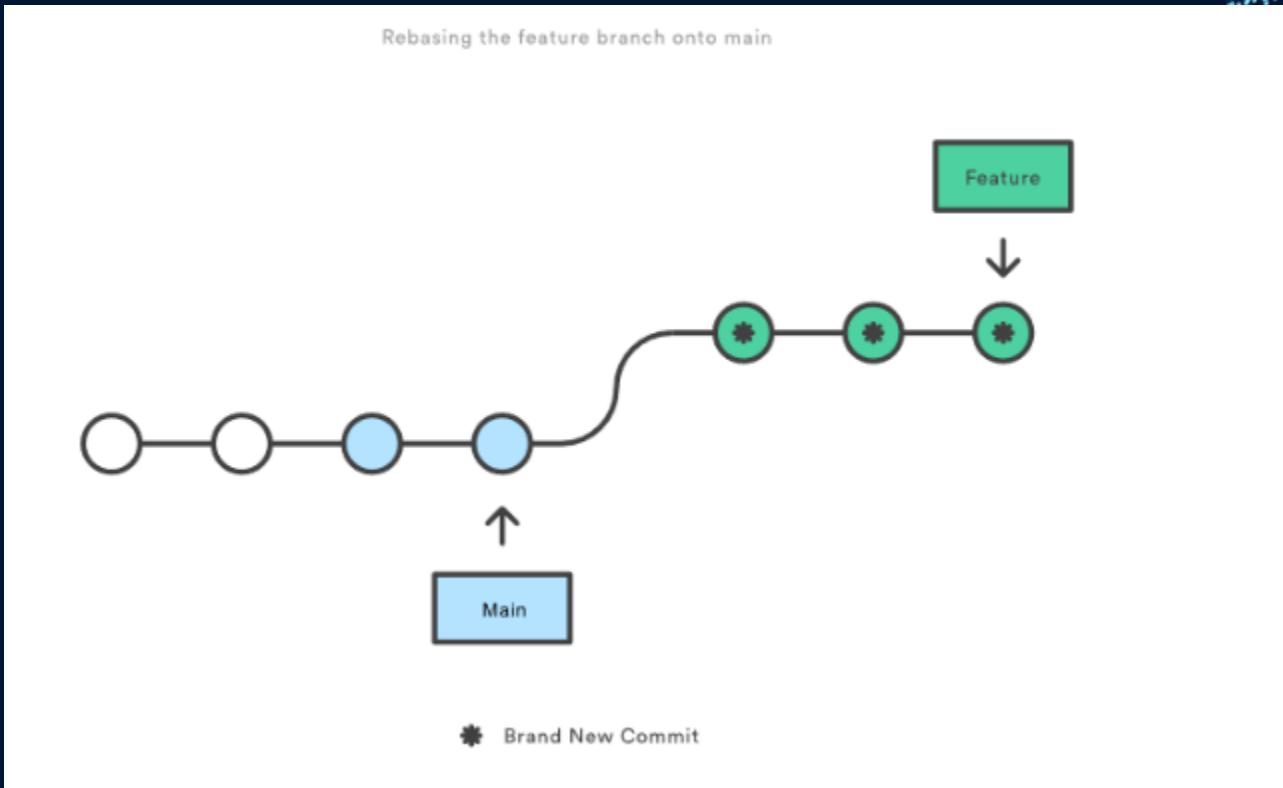
# Understanding Git Merge Strategies



# Understanding Git Merge Strategies



# Understanding Git Merge Strategies



# Rebasing

Basically, with git rebase you rewrite history; with this statement, I mean you can use rebase command to achieve the following:

- Combine two or more commits into a new one
- Discard a previous commit you did
- Change the starting point of a branch, split it, and much more

# Rebasing

## Reassembling commits

- Suppose we erroneously added half a grape in the shoppingList.txt file, then the other half, but at the end we want to have only one commit for the entire grape; follow me with these steps.
- Add a gr to the shopping list file:  
[1] ~/grocery (master)  
\$ echo -n "gr" >> shoppingList.txt

# Rebasing

- The -n option is for not adding a new line.
- Cat the file to be sure:

```
[2] ~/grocery (master)
```

```
$ cat shoppingList.txt
```

```
banana
```

```
apple
```

```
orange
```

```
peach
```

```
gr
```

# Rebasing

- Okay, we have a commit with half a grape.
- Go on and add the other half, ape:

```
[4] ~/grocery (master)
```

```
$ echo -n "ape" >> shoppingList.txt
```

- Check the file:

```
[5] ~/grocery (master)
```

```
$ cat shoppingList.txt
```

banana

apple

orange

peach

grape



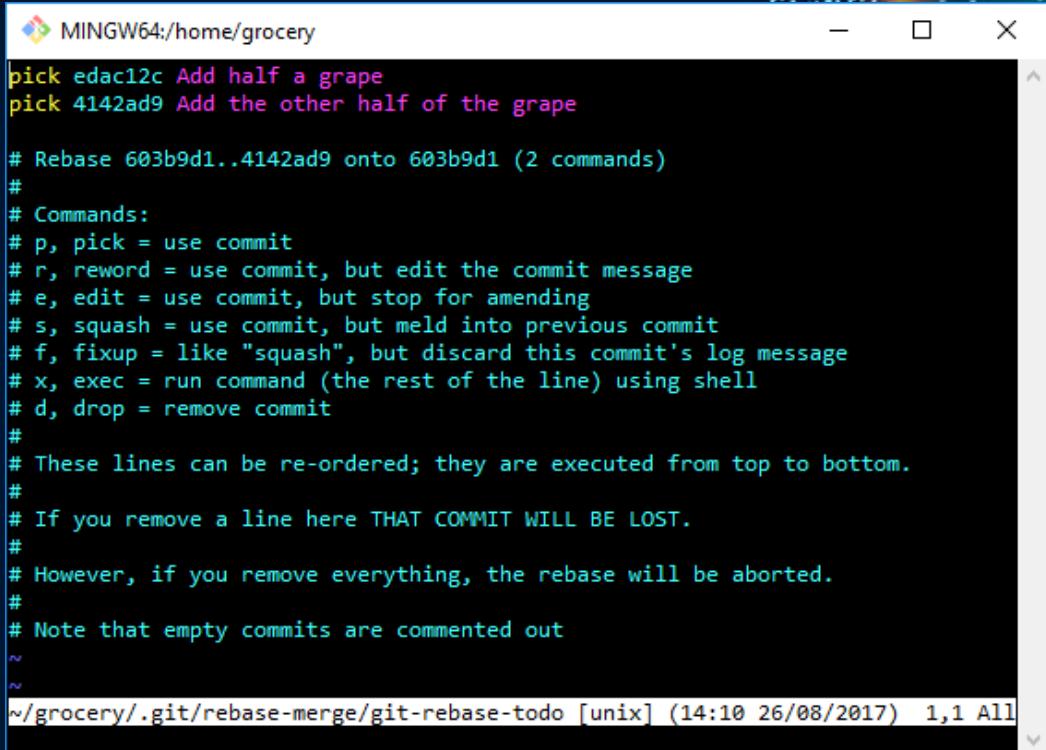
# Rebasing

- Check the log:

```
[7] ~/grocery (master)
$ git log --oneline --graph --decorate --all
* 4142ad9 (HEAD -> master) Add the other half of the grape
* edac12c Add half a grape
* 603b9d1 Add a peach
| * a8c6219 (melons) Add a watermelon
| * ef6c382 (berries) Add a blackberry
|/
* 0e8b5cf Add an orange
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```

# Rebasing

This is a screenshot  
of the console:



```
pick edac12c Add half a grape
pick 4142ad9 Add the other half of the grape

# Rebase 603b9d1..4142ad9 onto 603b9d1 (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
~
~

~/grocery/.git/rebase-merge/git-rebase-todo [unix] (14:10 26/08/2017) 1,1 All
```

# Rebasing

- To resolve our issue, I will reword the first commit and then fixup the second; the following is a screenshot of my console:



```
MINGW64:/home/grocery
reword edac12c Add half a grape
f 4142ad9 Add the other half of the grape

# Rebase 603b9d1..4142ad9 onto 603b9d1 (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
~
~
<grocery/.git/rebase-merge/git-rebase-todo[+] [unix] (14:10 26/08/2017)2,6 All
:wq|
```

# Rebasing

```
MINGW64:/home/grocery
Add half a grape

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# Date:      Sat Aug 26 14:00:58 2017 +0200
#
# interactive rebase in progress; onto 603b9d1
# Last command done (1 command done):
#   reword edac12c Add half a grape
# Next command to do (1 remaining command):
#   f 4142ad9 Add the other half of the grape
# You are currently editing a commit while rebasing branch 'master' on '603b9d1'.
#
# Changes to be committed:
#       modified:   shoppingList.txt
#
~
~/grocery/.git/COMMIT_EDITMSG [unix] (14:44 26/08/2017)          1,1 All
```

# Rebasing

- Now edit the message, and then save and exit, like in the following screenshot:

The screenshot shows a terminal window titled "MINGW64:/home/grocery". The command "git commit -c ORIG\_HEAD" has been run, which starts an interactive rebase. The terminal displays the commit message editor with the following content:

```
Add a grape

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# Date:      Sat Aug 26 14:00:58 2017 +0200
#
# interactive rebase in progress; onto 603b9d1
# Last command done (1 command done):
#   reword edac12c Add half a grape
# Next command to do (1 remaining command):
#   f 4142ad9 Add the other half of the grape
# You are currently editing a commit while rebasing branch 'master' on '603b9d1'.
#
# Changes to be committed:
#       modified:   shoppingList.txt
#
~
```

The bottom status bar shows the file path as "~/grocery/.git/COMMIT\_EDITMSG[+]" and the date and time as "(14:44 26/08/2017)". The status bar also indicates "11,5 All" and shows the command ":wq" being typed.

# Rebasing

- This is the final message from Git:

```
[8] ~/grocery (master)
$ git rebase -i HEAD~2
unix2dos: converting file C:/Users/san/Google Drive/Packt/PortableGit/home/grocery/[detached HEAD
53c73dd] Add a grape
Date: Sat Aug 26 14:00:58 2017 +0200
1 file changed, 1 insertion(+)
Successfully rebased and updated refs/heads/master.
```

- Take a look at the log:

```
[9] ~/grocery (master)
$ git log --oneline --graph --decorate --all
* 6409527 (HEAD -> master) Add a grape
* 603b9d1 Add a peach
| * a8c6219 (melons) Add a watermelon
| * ef6c382 (berries) Add a blackberry
|
* 0e8b5cf Add an orange
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```

# Rebasing

- Let's start by creating a new branch that points to commit 0e8b5cf, the orange one:

[1] ~/grocery (master)

```
$ git branch nuts 0e8b5cf
```

- This time I used the git branch command followed by two arguments, the name of the branch and the commit where to stick the label.
- As a result, a new nuts branch has been created:

```
[2] ~/grocery (master)
$ git log --oneline --graph --decorate --all
* 6409527 (HEAD -> master) Add a grape
* 603b9d1 Add a peach
| * a8c6219 (melons) Add a watermelon
| * ef6c382 (berries) Add a blackberry
|
* 0e8b5cf (nuts) Add an orange
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```

# Rebasing

- Move HEAD to the new branch with the git checkout command:

```
[3] ~/grocery (master)
```

```
$ git checkout nuts
```

```
Switched to branch 'nuts'
```

- Okay, now it's time to add a walnut; add it to the shoppingList.txt file:

```
[4] ~/grocery (nuts)
```

```
$ echo "walnut" >> shoppingList.txt
```

- Then do the commit:

```
[5] ~/grocery (nuts)
```

```
$ git commit -am "Add a walnut"
```

```
[master 3d3ae9c] Add a walnut
```

```
1 file changed, 1 insertion(+), 1 deletion(-)
```

# Rebasing

Check the log:

```
[6] ~/grocery (nuts)
$ git log --oneline --graph --decorate --all
* 9a52383 (HEAD -> nuts) Add a walnut
| * 6409527 (master) Add a grape
| * 603b9d1 Add a peach
|/
| * a8c6219 (melons) Add a watermelon
| * ef6c382 (berries) Add a blackberry
|/
* 0e8b5cf Add an orange
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```

# Rebasing

- Let's do it, rebasing the nuts branch on top of master; double-check that you actually are in the nuts branch, as a rebase command basically rebases the current branch (nuts) to the target one, master; so:

```
[7] ~/grocery (nuts)
$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: Add a walnut
Using index info to reconstruct a base tree...
M  shoppingList.txt
Falling back to patching base and 3-way merge...
Auto-merging shoppingList.txt
CONFLICT (content): Merge conflict in shoppingList.txt
Patch failed at 0001 Add a walnut
The copy of the patch that failed is found in: .git/rebase-apply/patch

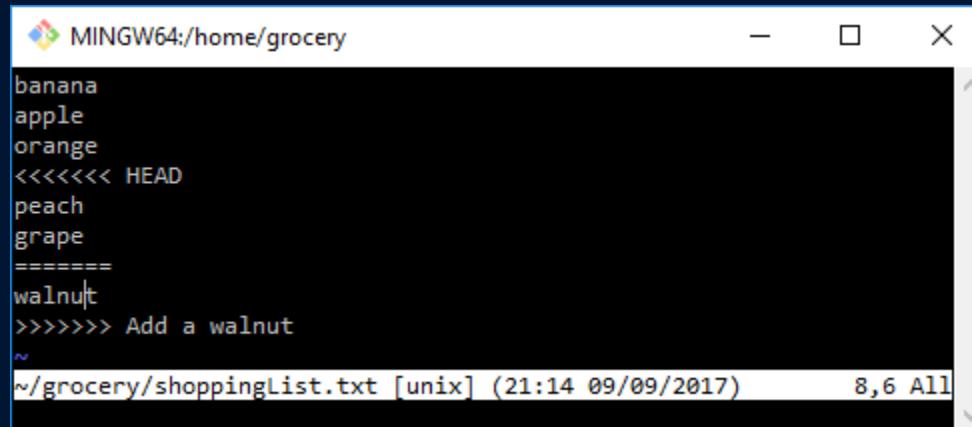
When you have resolved this problem, run "git rebase --continue".
If you prefer to skip this patch, run "git rebase --skip" instead.
To check out the original branch and stop rebasing, run "git rebase --abort".

error: Failed to merge in the changes.
```

# Rebasing

- Now, back to our repository; if you open the file with Vim, you can see the generated conflict:

```
[8] ~/grocery (nuts|REBASE 1/1)
$ vi shoppingList.txt
```



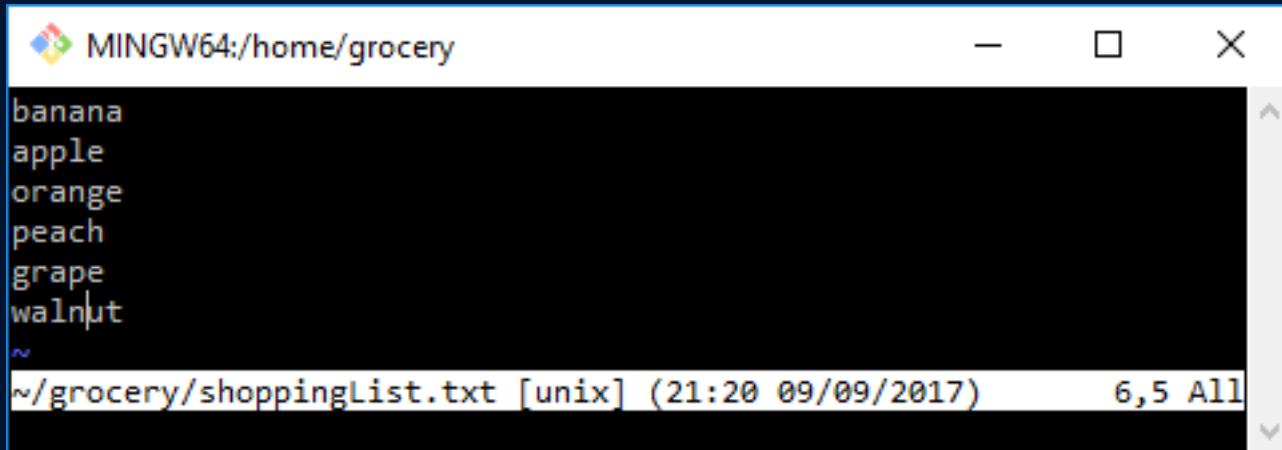
A screenshot of a terminal window titled "MINGW64:/home/grocery". The window displays a file named "shoppingList.txt" with the following content:

```
banana
apple
orange
<<<<< HEAD
peach
grape
=====
=====
walnut
>>>> Add a walnut
~
```

The word "nut" is highlighted in red, indicating a conflict. The status bar at the bottom shows the path "/grocery/shoppingList.txt [unix]" and the date and time "(21:14 09/09/2017)".

# Rebasing

- I will fix it adding the walnut at the end of the file:



```
MINGW64:/home/grocery
banana
apple
orange
peach
grape
walnut
~
~/grocery/shoppingList.txt [unix] (21:20 09/09/2017)       6,5 All
```

# Rebasing

- Now, the next step is to git add the shoppingList.txt file to the staging area, and then go on with the git rebase --continue command, as the previous message suggested:

```
[9] ~/grocery (nuts|REBASE 1/1)
$ git add shoppingList.txt
```

```
[10] ~/grocery (nuts|REBASE 1/1)
$ git rebase --continue
Applying: Add a walnut
```

```
[11] ~/grocery (nuts)
$
```

# Rebasing

Now take a look at the repo using git log as usual:

```
[12] ~/grocery (nuts)
$ git log --oneline --graph --decorate --all
* 383d95d (HEAD -> nuts) Add a walnut
* 6409527 (master) Add a grape
* 603b9d1 Add a peach
| * a8c6219 (melons) Add a watermelon
| * ef6c382 (berries) Add a blackberry
|/
* 0e8b5cf Add an orange
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```

# Rebasing

- Okay, now to keep the simplest and most compact repository.
- We cancel the walnut commit and put everything back in place as it was before this little experiment, even removing the nuts branch:

```
[13] ~/grocery (nuts)
$ git reset --hard HEAD^
HEAD is now at 6409527 Add a grape

[14] ~/grocery (nuts)
$ git checkout master
Switched to branch 'master'

[15] ~/grocery (master)
$ git branch -d nuts
Deleted branch nuts (was 6409527).
```

# Git squash

In Git, squashing refers to combining multiple commits into a single commit. This is often done to clean up the commit history, making it more concise and easier to understand, especially when merging feature branches or making pull requests.

# Git squash

## Advantages of Git Squash

### Streamlined Commits:

- Keeps commit history concise and meaningful.

### Easier Code Reviews:

- Reviewers can focus on a single, cohesive commit.

### Better Repository Maintenance:

- Reduces clutter in long-term repositories.

# Git squash

## When to Use Git Squash

### Before Merging:

- Clean up a feature branch before merging it into the main branch.

### During Pull Requests:

- Provide a clear and concise set of changes for maintainers to review.

### After Work-in-Progress (WIP):

- Consolidate experimental or intermediate commits into a single polished commit.

# 8. Viewing Local Project History Using Git Log

# Using Git Log

- In modern Git, you can trace the evolution of the line range within the file using git log -L, which is currently limited to walk starting from a single revision (zero or one positive revision arguments) and a single file.
- The range is given either by denoting the start and end of the range with -L <start>,<end>:<file> (where either <start> or <end> can be the line number or /regexp/), or a function to track with -L :<funcname regexp>:<file>.

# Selecting and formatting the git log output

- Now that you know how to select revisions to examine and to limit which revisions are shown (selecting those that are interesting).
- There is a huge number and variety of options of the git log command available for this.

# Predefined and user defined output formats

```
$ git log --pretty="%h - %an, %ar : %s"  
50f84e3 - Junio C Hamano, 7 days ago : Update draft release notes  
0953113 - Junio C Hamano, 10 days ago : Second batch for 2.1  
afa53fe - Nick Alcock, 2 weeks ago : t5538: move http push tests out
```

# Using Git Log

Placeholder	Description of output
%H	Commit hash (full SHA-1 identifier of revision)
%h	Abbreviated commit hash
%an	Author name
%ae	Author e-mail
%ar	Author date, relative
%cn	Committer name
%ce	Committer email
%cr	Committer date, relative
%s	Subject (first line of a commit message, describing revision)
%%	A raw %

# Using Git Log

```
$ git log --graph --decorate --oneline origin/maint
*   bce14aa (origin/maint) Sync with 1.9.4
|\ \
| * 34d5217 (tag: v1.9.4) Git 1.9.4
| * 12188a8 Merge branch 'rh/prompt' into maint
| |\ \
| * \  64d8c31 Merge branch 'mw/symlinks' into maint
| |\ \
* | | | d717282 t5537: re-drop http tests
* | | | e156455 (tag: v2.0.0) Git 2.0
```



# Useful Techniques



# Useful techniques

- Append a new fruit to the shopping list, then try to switch branch.
- Git won't allow you to do so, because with the checkout you would lose your local (not yet committed) changes to the shoppingList.txt file.
- So, type the git stash command; your changes will be set apart and removed from your current branch, letting you switch to another one (berries, in this case).

# Useful techniques

## Git commit amend - modify the last commit

- This trick is for people that don't double-check what they're doing.
- If you have pressed the enter key too early, there's a way to modify the last commit message or add that file you forgot, using the git commit command with the --amend option:

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

# Useful techniques

- The last tip I want to suggest is to use the Git GUI [3] ~/Spoon-Knife (master)  
\$ git gui blame README.md

```
Git Gui (Spoon-Knife): File Viewer
Repository Edit Help
Commit: File: README.md
bb4c bb4c 1 ### Well hello there!
TO TO 2
| | 3 This repository is meant to provide an example for *forking* a repository on GitHub.
| | 4
| | 5 Creating a *fork* is producing a personal copy of someone else's project. Forks are
| | 6
| | 7 After forking this repository, you can make some changes to the project, and submit
| | 8
d0dd d0dd 9 For some more information on how to fork a repository, [check out our guide, "Fo
< >
commit bb4cc8d3b2e14b3af5df699876dd4ff3acd00b7f
Author: The Octocat <octocat@nowhere.com> Tue Feb 4 23:38:36 2014
Committer: The Octocat <octocat@nowhere.com> Thu Feb 13 00:18:55 2014
Create styles.css and updated README
< >
Annotation complete.
```

# Tricks

- If you want to set up a bare repository, you only have to use the --bare option:

```
$ git init --bare NewRepository.git
```

- As you may have noticed, I called it NewRepository.git, using a .git extension; this is not mandatory, but is a common way to identify bare repositories. If you pay attention, you will note that even in GitHub every repository ends with a .git extension.

# Tricks

## Converting a regular repository to a bare one

- It can happen that you start working on a project in a local repository, and then you feel the need to move it to a centralized server to make it available for other people or from other locations.
- You can easily convert a regular repository to a bare one using the git clone command with the same --bare option:

```
$ git clone --bare my_project my_project.git
```

# Tricks

## Archiving the repository

- To archive the repository without including versioning information, you can use the git archive command; there are many output formats but the classic one is the .zip one:

```
$ git archive master --format=zip --output=../repoBackup.zip
```

# Tricks

- Please note that using this command is not the same as backing up folders in a filesystem.
- As you will have noticed, the git archive command can produce archives in a smarter way, including only files in a branch or even in a single commit.
- For example, by doing this you are archiving only the last commit:

```
$ git archive HEAD --format=zip --output=../headBackup.zip
```

# Tricks

## Bundling the repository

- Another interesting command is the git bundle command.
- With git bundle, you can export a snapshot from your repository and then restore it wherever you want.
- Suppose you want to clone your repository on another computer, and the network is down or absent; with this command, you can create a repo.bundle file of the master branch:

```
$ git bundle create ../repo.bundle master
```

# Tricks

- With this other command, we can restore the bundle in the other computer using the git clone command:

```
$ cd /OtherComputer/Folder  
$ git clone repo.bundle repo -b master
```

# Summary

- In this lesson, we enhanced our knowledge about Git and its wide set of commands.
- We discovered how configuration levels work, and how to set our preferences using Git by, for example, adding useful command aliases to the shell.
- Then we looked at how Git deals with stashes, providing the way to shelve then and reapply changes.

# What is a Fork?

- A fork is a copy of a repository that you manage.
- Forks let you make changes to a project without affecting the original repository.
- You can fetch updates from or submit changes to the original repository with pull requests.

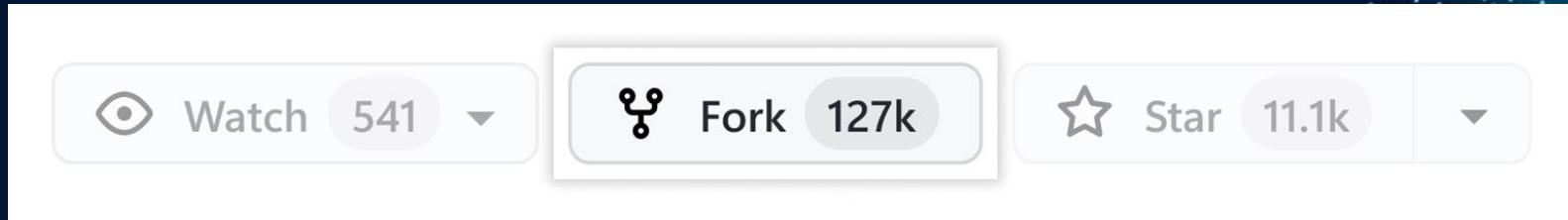
# Propose changes to someone else's project

For example, you can use forks to propose changes related to fixing a bug. Rather than logging an issue for a bug you have found, you can:

- Fork the repository.
- Make the fix.
- Submit a pull request to the project owner.

# Creating a Fork

- On GitHub.com, navigate to the octocat/Spoon-Knife repository.
- In the top-right corner of the page, click Fork.



# Creating a Fork

- Select an owner for the forked repository.

## Create a new fork

A *fork* is a copy of a repository. Forking a repository allows you to freely experiment with changes without affecting the original project. [View existing forks](#).

---

**Owner \***

 octocat▼

**Repository name \***

Spoon-Knife ✓

By default, forks are named the same as their parent repository. You can customize the name to distinguish it further.

# Creating a Fork

- By default, forks are named the same as their parent repositories.
- You can change the name of the fork to distinguish it further.

Create a new fork

A *fork* is a copy of a repository. Forking a repository allows you to freely experiment with changes without affecting the original project. [View existing forks](#).

---

Owner \*

 octocat

Repository name \*

Spoon-Knife ✓

By default, forks are named the same as their parent repository. You can customize the name to distinguish it further.

# Creating a Fork

## Create a new fork

A *fork* is a copy of a repository. Forking a repository allows you to freely experiment with changes without affecting the original project. [View existing forks.](#)

Owner \*



octocat▼

Repository name \*

A text input field containing the text "Spoon-Knife".A small green checkmark icon indicating the repository name is valid.

By default, forks are named the same as their parent repository. You can customize the name to distinguish it further.

Description (optional)

This repo is for demonstration purposes only.

# Creating a Fork

Owner \*

Select an owner ▾

Repository name \*

Spoon-Knife

By default, forks are named the same as their parent repository. You can customize the name to distinguish it further.

Description (optional)

This repo is for demonstration purposes only.

Copy the main branch only

Contribute back to octocat/Spoon-Knife by adding your own branch. [Learn more.](#)

# Creating a Fork

- Click Create fork.

Copy the main branch only

Contribute back to octocat/Spoon-Knife by adding your own branch. [Learn more.](#)

ⓘ You are creating a fork in your personal account.

Create fork

# Understanding Workflows

# Adopting a workflow - a wise act

- Now that we learnt how to perform good commits, it's time to fly higher and think of workflows.
- The thing that separates a great repository from a junkyard is the way you manage releases.
- The way you react when there is a bug to fix in particular version of your software.
- And the way you act when you have to make users able to beta-test incoming features.

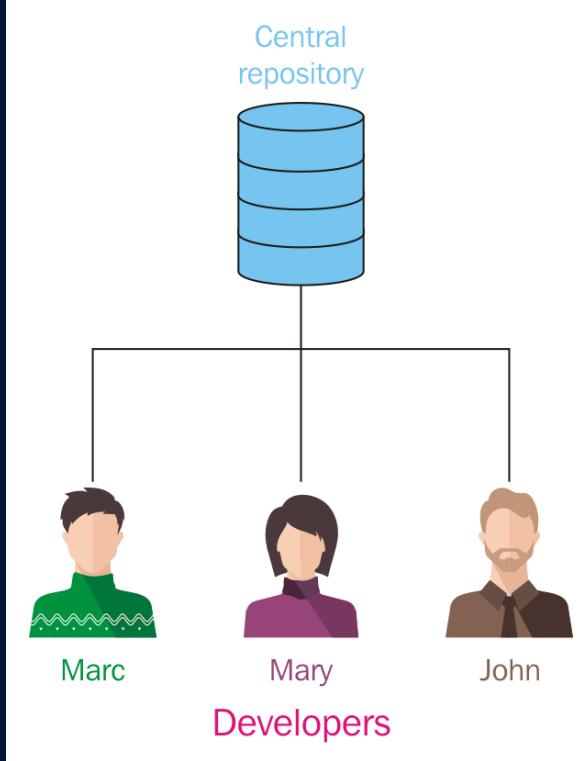
# Adopting a workflow - a wise act

## Centralized workflows

- As we used to do in other VCS like Subversion or so, even in Git it is not uncommon to adopt a centralized way of working.
- If you work in a team, it is often necessary to share repositories with others, so a common point of contact becomes indispensable.

# Adopting a workflow - a wise act

- The scenario is represented in the following picture:

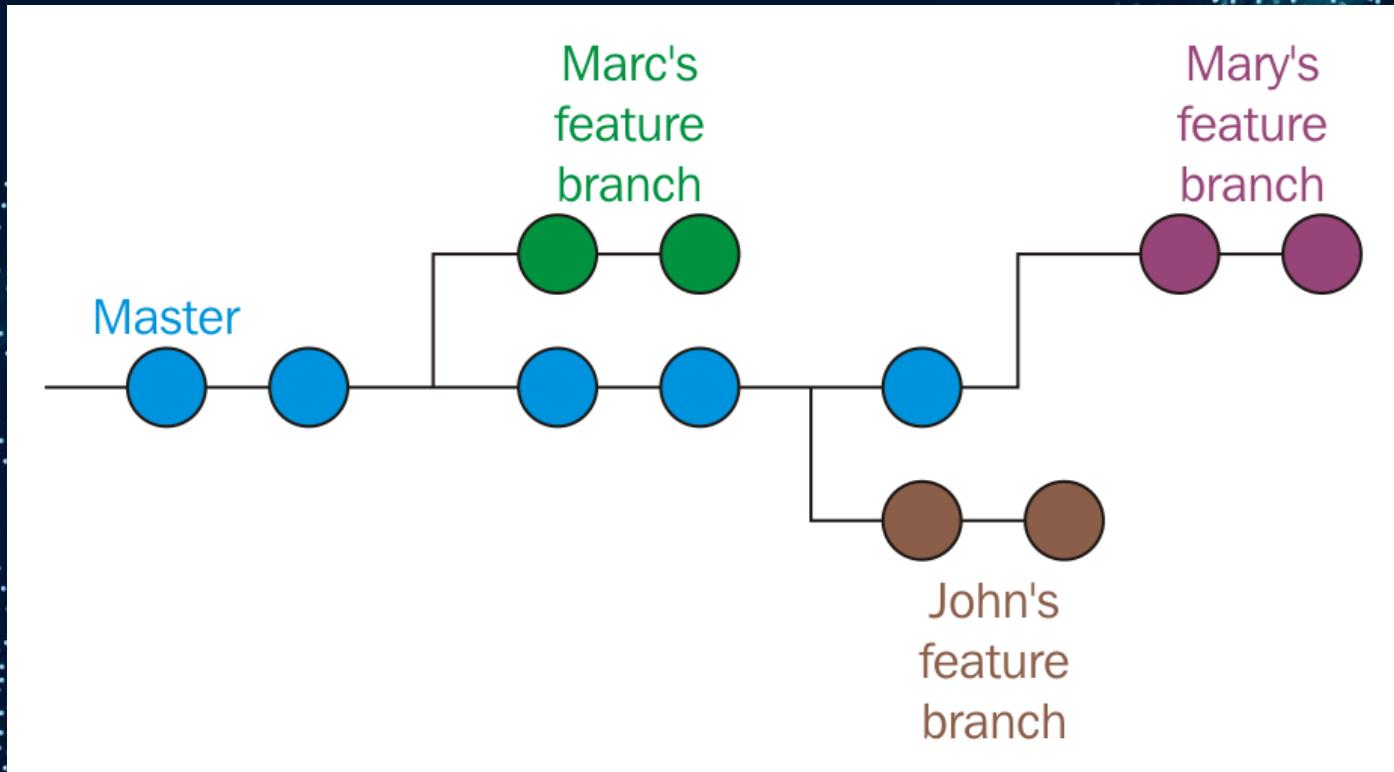


# Adopting a workflow - a wise act

## **Feature branch workflow**

- At this point, you will probably choose at least a feature branch approach, where every single developer works on his branch.

# Adopting a workflow - a wise act

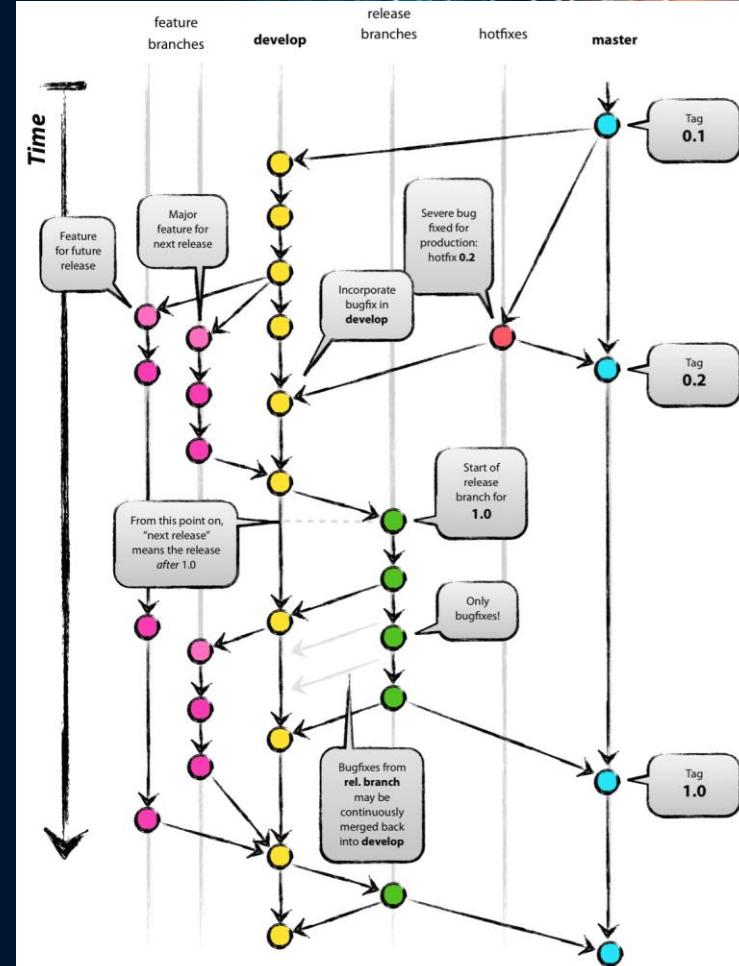


# Adopting a workflow - a wise act

## Gitflow

- His workflow has gained success over the years, to the point that many other developers (including me!), teams, and companies are starting to use it.
- Atlassian, a well-known company that offers Git related services like BitBucket, integrates the Gitflow directly in their GUI tool, the nice SourceTree.

- Even the Git flow work flow is a centralized one, and it is well described by the following image:



# Adopting a workflow - a wise act

## GitHub flow

- The previously described Git Flow has tons of followers.
- But it's always a matter of taste; someone else found it too complex and rigid for their situation.
- In fact there are other ways to manage software repositories that have gained consensus during the last few years.

# Adopting a workflow - a wise act

## **Anything in the master branch is deployable**

- Similar to GitFlow, even here in GitHub flow, deploy is done from the master branch.
- This is the only main branch in this flow; in Gitflow there are no hotfix, develop, or other particular branches.
- Bug fixes, new implementations and so on are constantly merged onto the master branch.

# Adopting a workflow - a wise act

## **Creating descriptive branches off of master**

- In GitFlow you always branch from the master, so it's easy to get a forest of branches to look at when you have to pull one.
- To better identify them, in GitHub flow you have to use descriptive names to get meaningful topic branches.

# Adopting a workflow - a wise act

## **Pushing to named branches constantly**

- Another great difference when comparing GitHub flow to Gitflow is that in GitHub flow you push feature branches to the remote regularly.
- Even if you are the only developer involved and interested.
- This is done for continuous integration and testing, or maybe also for backup purposes; regarding the backup part.
- Even if I already exposed my opinion in merit, I can't say this is a bad thing.

# Adopting a workflow - a wise act

## **Opening a pull request at any time**

Previously, we talked about GitHub and made a quick try with Pull Requests.

- We have seen that basically they are for contributing: you fork someone else's repository.
- Create a new branch, make some modifications and then ask for a pull request from the original author.

# Adopting a workflow - a wise act

## Merging only after pull request review

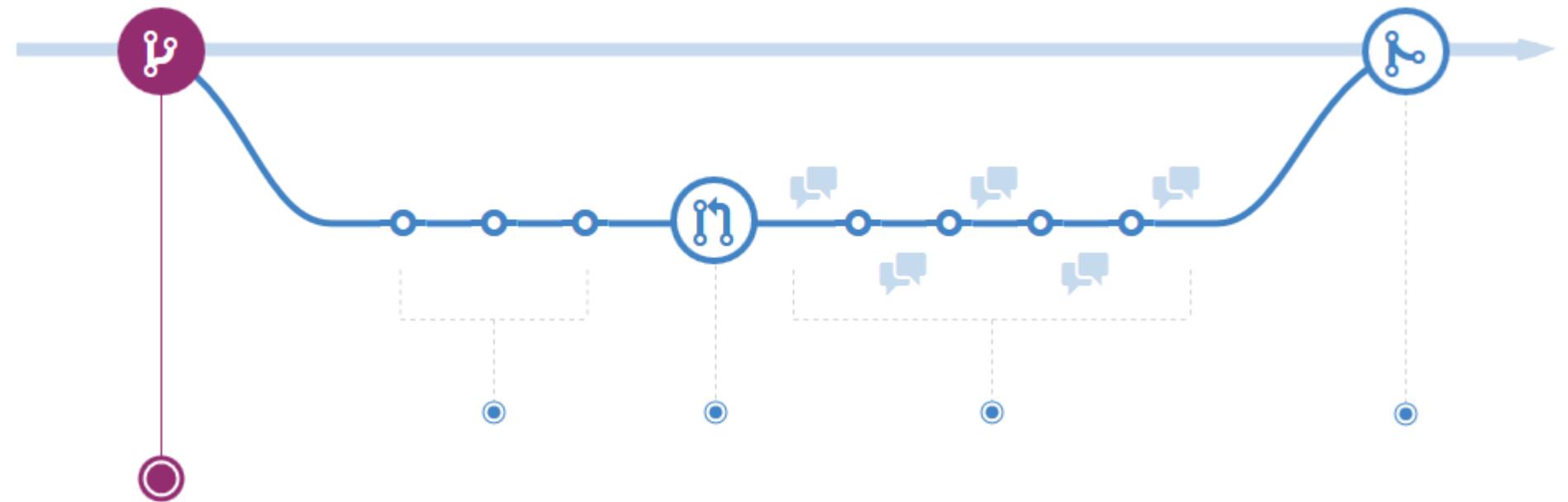
- You can understand now that the pull requested branch stage we have seen above becomes a sort of review stage, where other users can take a look at the code.
- Simply leave a positive comment, just a +1 to let other users know that they are confident about the job, and that they approve its merge into master.

# Adopting a workflow - a wise act

## **Deploying immediately after review**

- At this stage, you merge your branch into master and the work is done.
- The deploy is not instantly fired, but at GitHub they have a very straight and robust deployment procedure, so they can do that easily.
- They deploy big branches with 50 commits but even branches with a single commit.
- A single line of code change, because deployment is very quick and cheap for them.

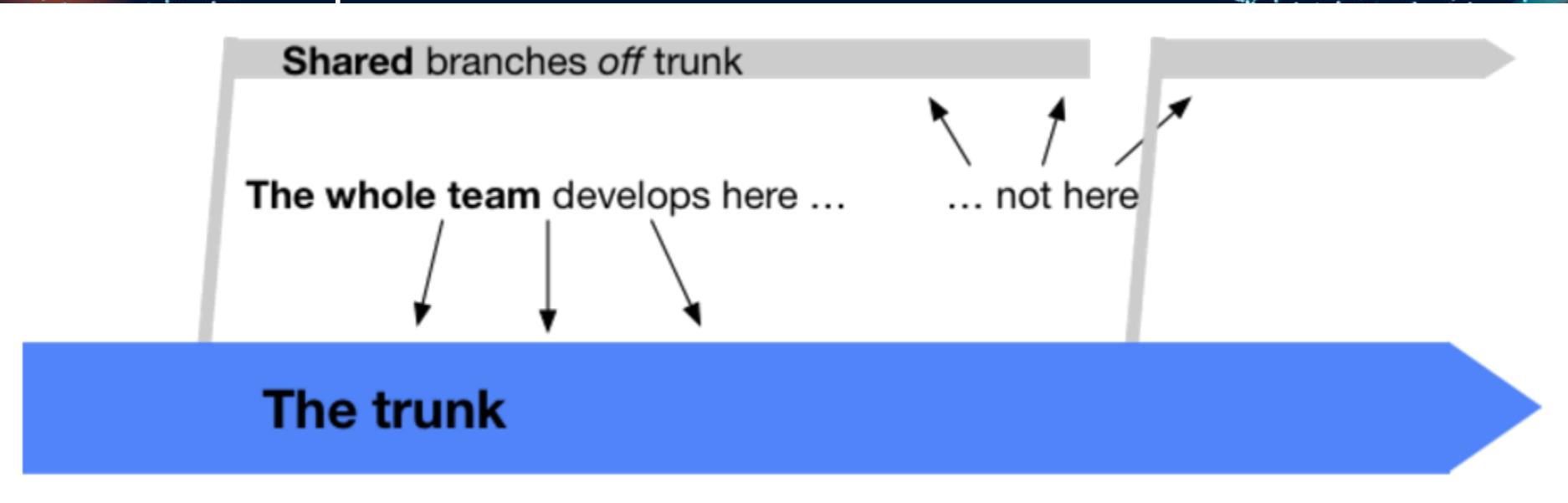
# Adopting a workflow - a wise act



# Adopting a workflow - a wise act

## Trunk-based development

- In the picture below there's the essence of this flow:

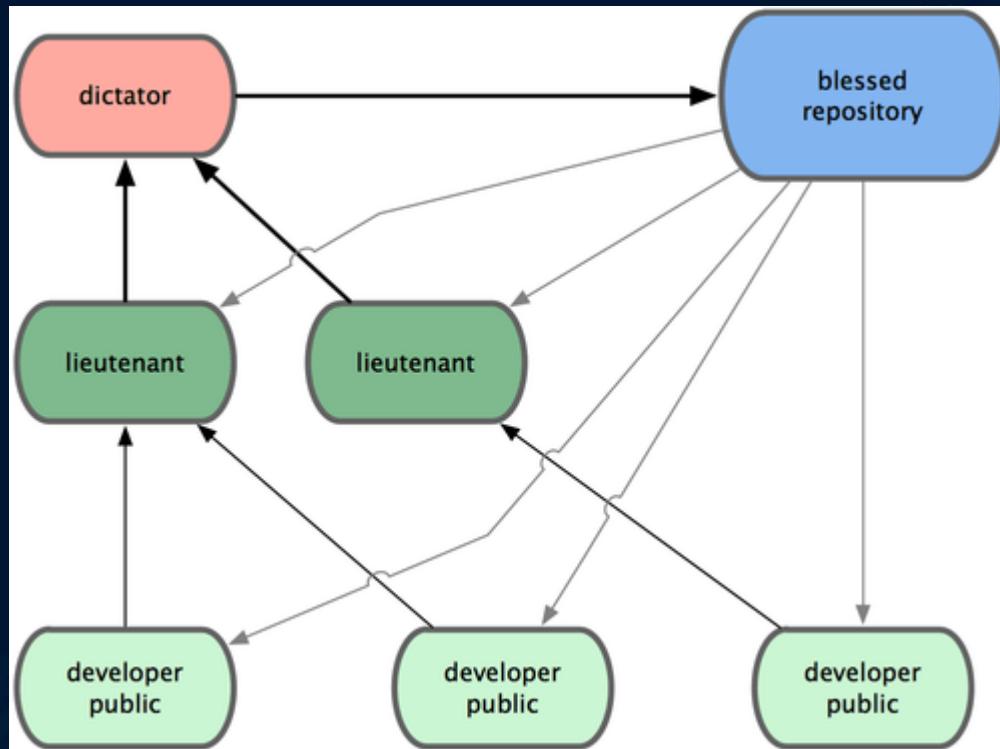


# Adopting a workflow - a wise act

## Linux kernel workflow

- The Linux kernel uses a workflow that refers to the traditional way Linus Torvalds has driven its evolution during these years, based on a military like hierarchy.
- Simple kernel developers work on their personal branches, rebasing the master branch on the reference repository.
- Then push their branches to the lieutenant developers master branch.

# Adopting a workflow - a wise act



# 13. Searching for Events in Your Code

# What is Git bisect?

- The git bisect command is used to discover the commit that has introduced a bug in the code.
- It helps track down the commit where the code works and the commit where it does not, hence, tracking down the commit that introduced the bug into the code.

# What is Git bisect?

- Suppose there are many commits, and we have to find the commit that introduced the bug.
- The normal way to do this would be to check out each commit one by one and build it until we reach the commit where the code does not compile.



git bisect performs a binary search to find the faulty commit.

# 1. Initializing the repository

```
mkdir git_bisect_example  
cd git_bisect_example  
git init
```

## 2. Creating commits to demonstrate git bisect

```
commit 9598ae96a6bba2cbd4eda693f8aa6cd92fd4c295
Author: Example <example@example.com>
Date:   Fri Apr 9 11:01:57 2021 +0000
```

Adding 4

```
commit b75591336fe4f603fd36eaf6354da8a652025186
Author: Example <example@example.com>
Date:   Fri Apr 9 11:01:57 2021 +0000
```

Changing the 2 to 9

```
commit a59e5dd4632361015856367535d95cbb493a07d3
Author: Example <example@example.com>
Date:   Fri Apr 9 11:01:57 2021 +0000
```

Adding 3

Adding 3

```
commit 67495ba9c2dec29abc00af776e6cd042fe09a477
Author: Example <example@example.com>
Date:   Fri Apr 9 11:01:57 2021 +0000
```

Adding 2

```
commit 548c695b9d5f6906e30507de8d672b82f088f5a3
Author: Example <example@example.com>
Date:   Fri Apr 9 11:01:57 2021 +0000
```

Adding 1

# Finding the bug in your project

- We can now categorize the first commit as good and the last commit as bad.
- We check the contents of the test file and, since the number 2 is still present, we categorize the commit as good.

```
root@educative:/git_bisect_example# git bisect start
root@educative:/git_bisect_example# git bisect good 548c
root@educative:/git_bisect_example# git bisect bad 9598ae
Bisecting: 1 revision left to test after this (roughly 1 step)
[a59e5dd4632361015856367535d95cbb493a07d3] Adding 3
root@educative:/git_bisect_example# cat test.txt
1
2
3
root@educative:/git_bisect_example# git bisect good
```

# Finding the bug in your project

- git bisect now takes us to another commit.
- We repeat the process and find the new commit to be bad since it contains the number 9 instead of the number 2.

```
root@educative:/git_bisect_example# git bisect good
Bisection: 0 revisions left to test after this (roughly 0 steps)
[b75591336fe4f603fd36eaf6354da8a652025186] Changing the 2 to 9
root@educative:/git_bisect_example# cat test.txt
1
9
3
root@educative:/git_bisect_example# git bisect bad
```

# Finding the bug in your project

- git bisect finally returns the commit to us where the first error took place.

```
root@educative:/git_bisect_example# git bisect bad
b75591336fe4f603fd36eaf6354da8a652025186 is the first bad commit
commit b75591336fe4f603fd36eaf6354da8a652025186
Author: Example <example@example.com>
Date:   Fri Apr 9 11:01:57 2021 +0000

        Changing the 2 to 9

:100644 100644 01e79c32a8c99c557f0757da7cb6d65b3414466d 27eb24fce3f28aadc
c947c438c0ab12373093c8d M      test.txt
root@educative:/git_bisect_example#
```

# Atomic Commits

# 14. Reverting Commits

# How Commits are made

- It is possible to correct mistakes made in Git the with git push context (without exposing them if the mistake is found before sharing or publishing the change).
- If the mistake is already pushed, it is still possible to undo the changes made to the commit that introduced the mistake.
- We'll also look at the reflog command and how we can use that and git fsck to recover lost information.

# How Commits are made

In this, we'll explore the possibilities to undo a commit in several ways depending on what we want to achieve. We'll explore four ways to undo a commit:

- Undo everything; just remove the last commit like it never happened.
- Undo the commit and unstage the files; this takes us back to where we were before we started to add the files.
- Undo the commit, but keep the files in the index or staging area so we can just perform some minor modifications and then complete the commit.
- Undo the commit with the dirty work area.

# The art of committing

- While working with Git, committing seems the easiest part of the job: you add files, write a short comment, and then you're done.
- But it is just for its simplicity that often, especially at the very beginning of your experience.
- You acquire the bad habit of doing terrible commits: too late, too big, too short, or simply equipped with bad messages.

# The art of committing

## Building the right commit

- One of the harder skills to acquire while programming in general is to split the work into small and meaningful tasks.
- Too often, I have experienced this scenario: you start to fix a small issue in a file; then you see another piece of code that can be easily improved, even if it's not related to what you are working on now - you can't resist, and you fix it.

# The art of committing

	COMMENT	DATE
O	CREATED MAIN LOOP & TIMING CONTROL	14 HOURS AGO
O	ENABLED CONFIG FILE PARSING	9 HOURS AGO
O	MISC BUGFIXES	5 HOURS AGO
O	CODE ADDITIONS/EDITS	4 HOURS AGO
O	MORE CODE	4 HOURS AGO
O	HERE HAVE CODE	4 HOURS AGO
O	AAAAAAA	3 HOURS AGO
O	ADKFJSLKDFJSOKLFJ	3 HOURS AGO
O	MY HANDS ARE TYPING WORDS	2 HOURS AGO
O	HAAAAAAAAANDS	2 HOURS AGO

AS A PROJECT DRAGS ON, MY GIT COMMIT MESSAGES GET LESS AND LESS INFORMATIVE.

# The art of committing

## **Making only one change per commit**

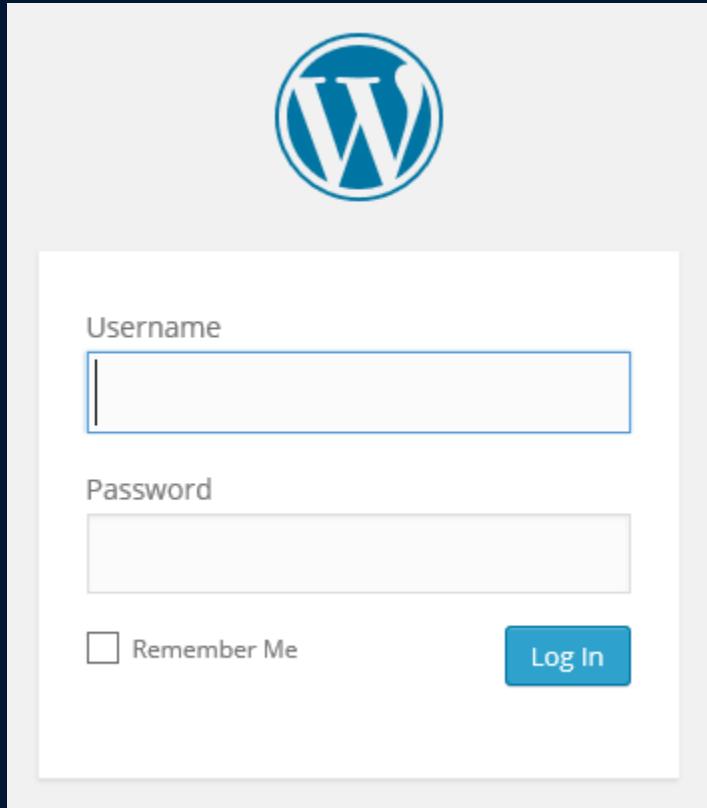
- After the routine morning coffee, you open your editor and then you start to work on a bug: BUG42.
- Working around fixing the bug in the code, you realize that fixing BUG79 will require tweaking just a single line of code,
- So you fix it, but you not only change that awful class name, but also add a good-looking label to the form and make a few more changes.
- The damage is done now.

# The art of committing

## **Splitting up features and tasks**

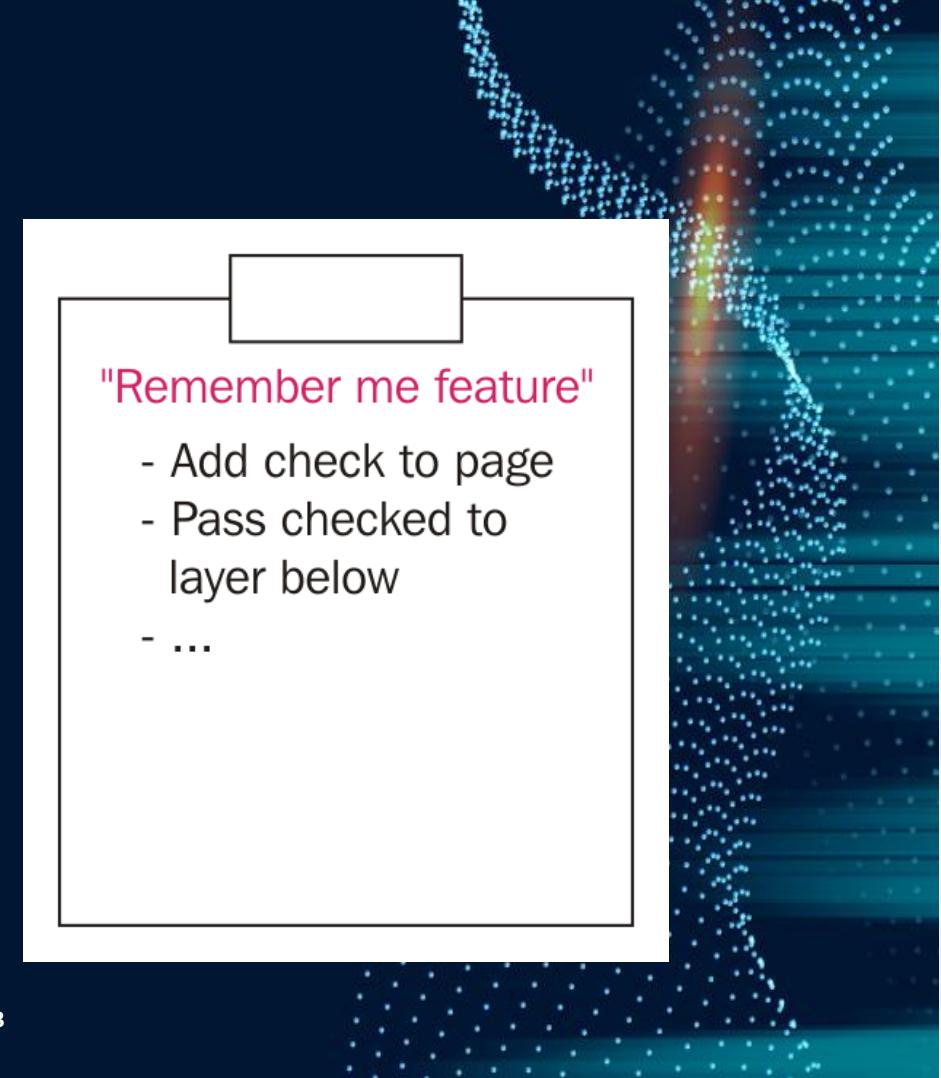
- As said before, breaking up things to do is a fine art.
- If you know and adopt some Agile Movement techniques, you have probably faced these problems already, so you have an advantage.
- Otherwise you will need to make a little more effort, but it isn't anything you can't achieve.

# The art of committing



# The art of committing

- Make a little notebook, like the one in the following picture - it will become one of your most precious tools:



# The art of committing

## **Including the whole change in one commit**

- Making more than one change per commit is a bad thing.
- Even splitting a single change into more than one commit is considered harmful.
- As you may already know, in some trained teams you do not simply push your code to production.
- First you have to pass code quality reviews, where someone else tries to understand what you did to decide if your code is good or not (that is why there are pull requests, indeed).

# The art of committing

## Describing the change, not what have you done

- Too often I read (and more often I wrote) commit messages like "Removed this", "Changed that", "Added that one" and so on.
- Imagine you are going to work on the common "lost password" feature on your website; you'll probably find a message like this adequate: "Added the lost password retrieval link to the login page".

# The art of committing

"Implement the password retrieval mechanism

- Add the "Lost password?" link into the login page
- Send an email to the user with a link to renew the password"

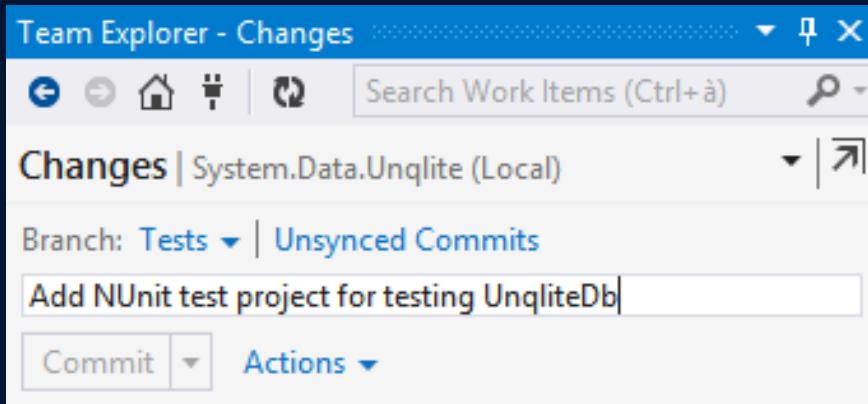
# The art of committing

## **Don't be afraid to commit**

- Fear is one of the most powerful of emotions; it can drive a person to do the craziest things on Earth.
- One of the most common reactions to fear is the breakdown: you don't know what to do, so you end up doing nothing.

# The art of committing

- To help myself get used to committing this way, I followed this simple trick: write the commit message in Visual Studio before starting to write any code:



# The art of committing

## **Adding bulleted details lines when needed**

- Often you can say all that you want in 50 chars; in that case, use details lines.
- In this situation, the common rule is to leave a blank line after the subject, use a dash and go no longer than 72 chars:

"Add the newsletter signup in homepage  
- Add textbox and button on homepage- Implement email address validation- Save email in database"

# The art of committing

## Tying other useful information

- If you use issue and project tracking systems, write down the issue number, bug id's, or anything else helpful:

"Add the newsletter signup in homepage  
- Add textbox and button on homepage- Implement email address validation- Save email in database  
#FEAT-123: closed"

# Reverting Commits

Undo – remove a commit completely

- In this example, we'll learn how we can undo a commit as if it had never happened.
- We'll learn how we can use the reset command to effectively discard the commit and thereby reset our branch to the desired state.

# Reverting Commits

## Getting ready

- In this example, we'll use the example of the hello world repository, clone the repository, and change your working directory to the cloned one:

```
$ git clone  
https://github.com/dvaske/hello_world_cookbook.git  
$ cd hello_world_cookbook
```

# How to do it...

- We'll make sure our working directory is clean, no files are in the modified state, and nothing is added to the index:

```
$ git status
```

```
On branch master
```

```
Your branch is up-to-date with 'origin/master'.
```

nothing to commit, working directory clean

# Reverting Commits

- Also, check what is in our working tree:

```
$ ls
```

```
HelloWorld.java Makefile      hello_world.c
```

- If all works well, we'll check the log to see the history of the repository.
- We'll use the --oneline switch to limit the output:

```
$ git log --oneline
```

```
3061dc6 Adds Java version of 'hello world'
```

```
9c7532f Fixes compiler warnings
```

```
5b5d692 Initial commit, K&R hello world
```

# Reverting Commits

- The most recent commit is the 3061dc6 Adds Java version of 'hello world' commit.
- We will now undo the commit as though it never happened and the history won't show it:

```
$ git reset --hard HEAD^  
HEAD is now at 9c7532f Fixes compiler warnings
```

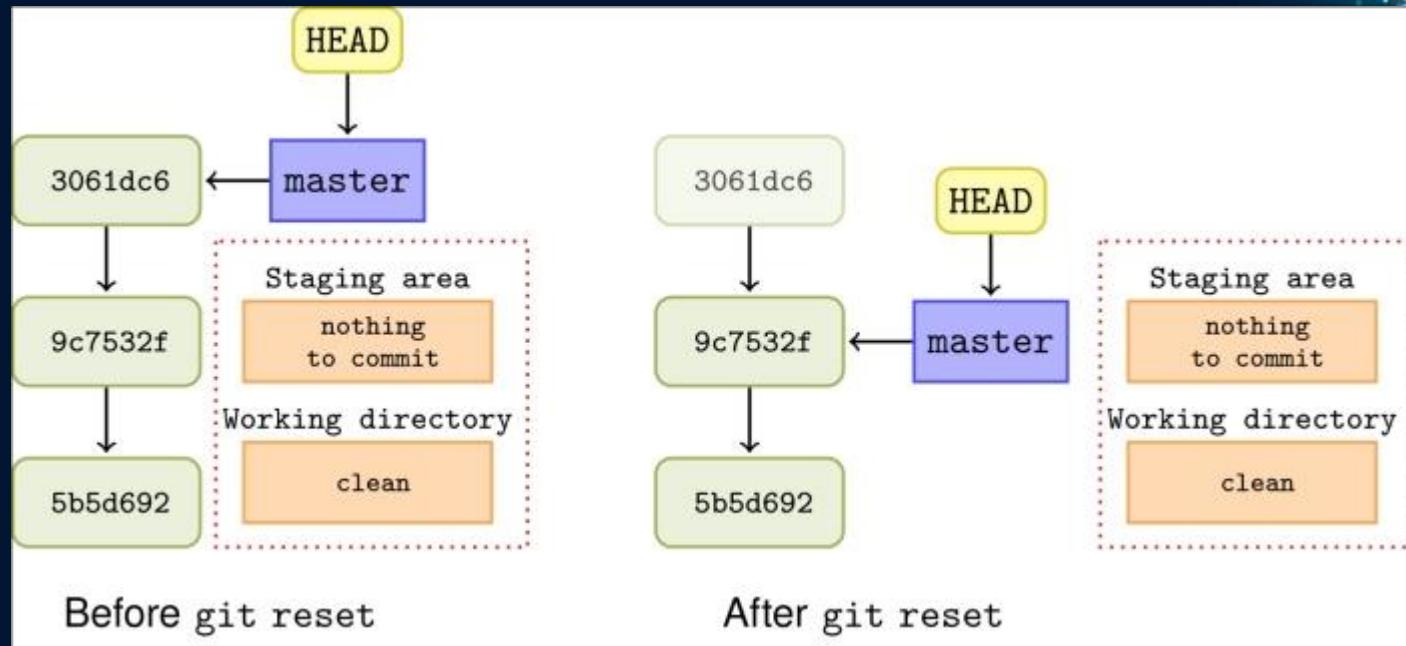
# Reverting Commits

- Check the log, status, and filesystem so that you can see what actually happened:

```
$ git log --oneline
9c7532f Fixes compiler warnings
5b5d692 Initial commit, K&R hello world
$ 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 directory clean
$ ls
hello_world.c
```

# How it works...



# 15. Helpful Git Commands

# Moving and Renaming Files with Git

- Git keeps track of changes to files in the working directory of a repository by their name
- When you move or rename a file, Git doesn't see that a file was moved; it sees that there's a file with a new filename, and the file with the old filename was deleted (even if the contents remain the same).
- As a result, renaming or moving a file in Git is essentially the same operation; both tell Git to look for an existing file in a new location.

# Problem

- In your Git working directory, you wish to rename a previously committed file named `mycoolclass.cs` to `myCoolClass.cs` and commit the newly renamed file.



# SOLUTION

- Change to the directory containing your repository: for example,  
`cd /Repo/MyProject/.`

- Run the following git command:

```
git mv mycoolclass.cs myCoolClass.cs
```

There will be no output.

- Run the following command to commit the change:

```
git commit --message 'Rename the cool class'
```



# SOLUTION

- The output will like the following:

```
# git commit --message 'Rename the cool class'  
[master c6eed66] Rename the coolclass  
1 file changed, 0 insertions(+), 0 deletions(-)  
rename thecoolclass.cs => theCoolClass.cs (100%)
```

You've renamed thecoolclass.cs to theCoolClass.cs and committed it.

# Rename file

`git mv options oldFilename newFilename`

- `oldFilename`: The name of the file that we rename
- `newFilename`: The new name of the file



# Move file

`git mv filename foldername`

- filename: The name of the file that is moved
- foldername: The name of the folder where the file is moved



# Staging Hunks of Changes

- It's worth noting that staging has little to do with 'files' themselves and everything to do with the changes within each given file.
- We stage files that contain changes, and git tracks the changes as commits (even when the changes in a commit are made across several files).
- The distinction between files and commits may seem minor, but understanding this difference is fundamental to understanding essential functions like cherry-pick and diff.

# Add changes by hunk

## Example

- You can see what "hunks" of work would be staged for commit using the patch flag:

`git add -p`

- or

`git add --patch`

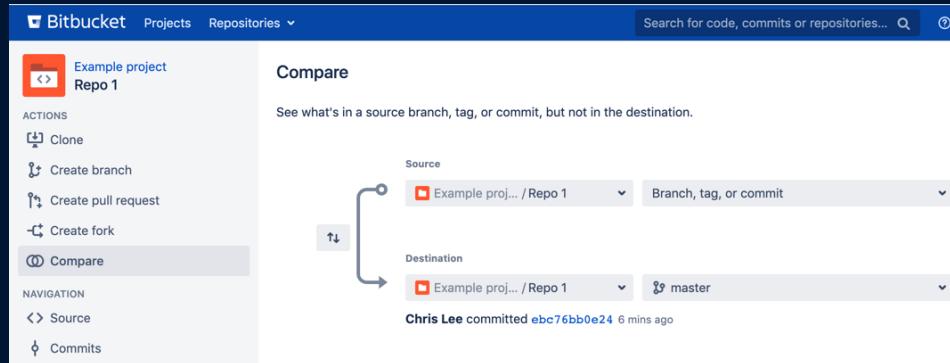
- This opens an interactive prompt that allows you to look at the diffs and let you decide whether you want to include them or not.

`Stage this hunk [y,n,q,a,d/,s,e,?]`

# 16. Viewing Local Changes

# Comparing changes with the Repository

- You can compare the state of your repository across branches, tags, commits, forks, and dates.
- To compare different versions of your repository, append /compare to your repository's path.
- We'll demonstrate the power of Compare by looking at the compare page for a fork of the Linguist repo, which is at <https://github.com/octocat/linguist/compare/master...octocat:master>.



# Comparing branches

- The most common use of Compare is to compare branches, such as when you're starting a new pull request.
- You'll always be taken to the branch comparison view when starting a new pull request.
- To compare branches, you can select a branch name from the compare drop down menu at the top of the page.

# Comparing changes with the Repository

## Comparing commits

- You can also compare two arbitrary commits in your repository or its forks on GitHub in a two-dot diff comparison.
- To quickly compare two commits or Git Object IDs (OIDs) directly with each other in a two-dot diff comparison on GitHub, edit the URL of your repository's "Comparing changes" page.

# Comparing across forks

- You can compare your base repository and any forked repository.
- This is the view that's presented when a user performs a Pull Request to a project.
- To compare branches on different repositories, preface the branch names with user names.
- For example, by specifying octocat:main for base and octo-org:main for compare, you can compare the main branch of the repositories respectively owned by octocat and octo-org.

# Git Fundamentals - Niche Concepts, Configurations, and Commands

A photograph showing a person's hands typing on a white laptop keyboard. The background is slightly blurred, showing a window with a view of the outdoors and a blue patterned fabric, possibly a bedsheet or a sofa cover, in the foreground.

# Git Fundamentals - Niche Concepts, Configurations, and Commands

- This lesson is a collection of short but useful tricks to make our Git experience more comfortable.
- In the first three lessons, we learned all the concepts we need to take the first steps into versioning systems using the Git tool; now it's time to go a little bit in depth to discover some other powerful weapons in the Git arsenal, and how to use them (without shooting yourself in the foot, preferably).

# Dissecting Git Configuration

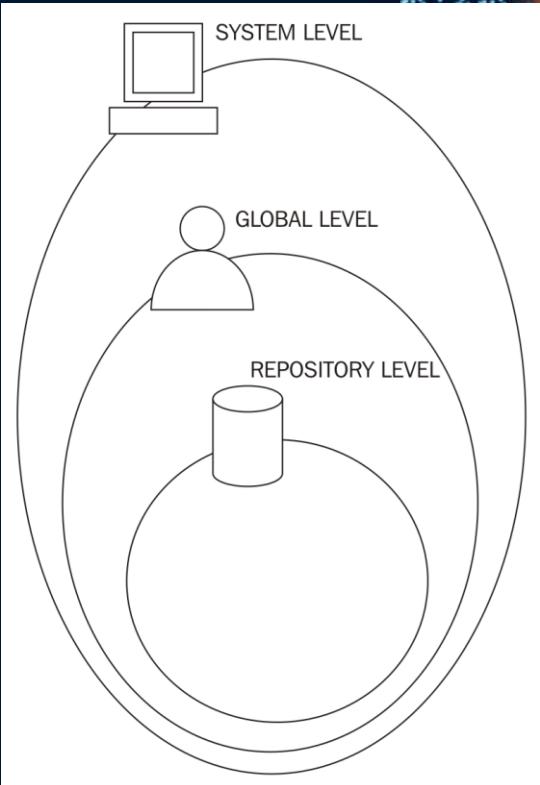
## Configuration levels

In Git, we have three configuration levels:

1. System
1. Global (user-wide)
1. Repository

# Dissecting Git configuration

- The following figure will help you to better understand these levels:



# Dissecting Git Configuration

## System level

This configuration is stored in the `gitconfig` file usually located in:

- Windows: `C:\Program Files\Git\etc\gitconfig`
- Linux: `/etc/gitconfig`
- macOS: `/usr/local/git/etc/gitconfig`

# Dissecting Git Configuration

## Global level

This configuration is stored in the `.gitconfig` file usually located in:

- Windows: `C:\Users\<UserName>\.gitconfig`
- Linux: `~/.gitconfig`
- macOS: `~/.gitconfig`

# Dissecting Git Configuration

## Repository level

This configuration is stored in the config file located in the .git repository subfolder:

Windows: C:\<MyRepoFolder>\.git\config

Linux: ~/<MyRepoFolder>/.git/config

macOS: ~/<MyRepoFolder>/.git/config

# Dissecting Git Configuration

## Listing configurations

- To get a list of all the configurations currently in use, you can run the `git config --list` command; if you are inside a repository, it will show all the configurations, from repository to system level.
- To filter the list, append optionally `--system`, `--global` or `--local` options to obtain only the desired level configurations.

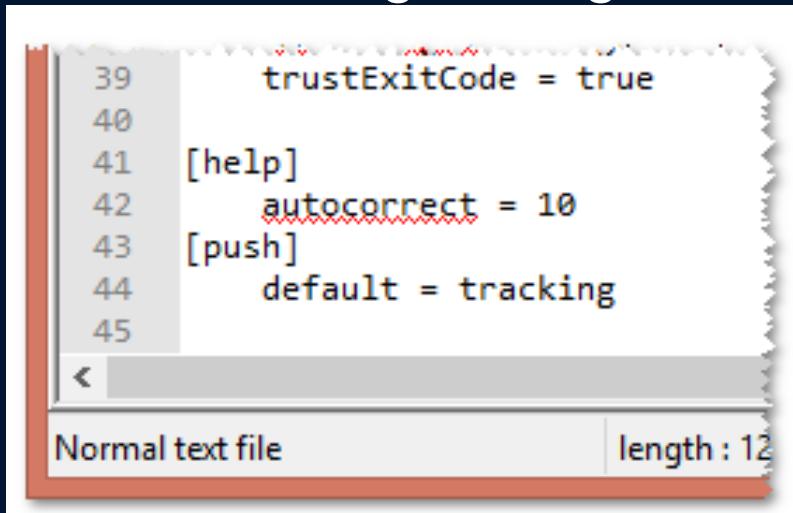
# Dissecting Git Configuration

## Typos autocorrection

- So, let's try to fix an annoying question about typing command: typos.
- I often find myself re-typing the same command two or more times; Git can help us with embedded autocorrection, but we first have to enable it.
- To enable it, you have to modify the `help.autocorrection` parameter, defining how many tenths of a second Git will wait before running the assumed command; so giving a `help.autocorrect 10`, Git will wait for a second:

# Dissecting Git Configuration

- You can see the section names within [] if you look in the configuration file; for example, in C:\Users\<UserName>\.gitconfig:



```
39     trustExitCode = true
40
41 [help]
42     autocorrect = 10
43 [push]
44     default = tracking
45
```

Normal text file length : 12

A screenshot of a terminal window displaying the contents of a file. The file contains several configuration sections: 'trustExitCode = true' (line 39), a '[help]' section (lines 41-42) containing 'autocorrect = 10', and a '[push]' section (lines 43-44) containing 'default = tracking'. The terminal interface shows line numbers on the left (39, 40, 41, 42, 43, 44, 45), a scroll bar on the right, and status information at the bottom: 'Normal text file' and 'length : 12'.

# Dissecting Git Configuration

- There are two ways we can do this.
- First one: set Git to ask us the name of the branch we want to push every time, so a simple git push will have no effect.
- To obtain this, set push.default to nothing:

```
[1] ~/grocery-cloned (master)
```

```
$ git config --global push.default nothing
```

```
[2] ~/grocery-cloned (master)
```

```
$ git push
```

```
fatal: You didn't specify any refs to push, and push.default is  
"nothing".
```

# Dissecting Git Configuration

- Another way to save yourself from this kind of mistake is to set the push.default parameter to simple, allowing Git to push only when there is a remote branch with the same name as the local one:

```
[3] ~/grocery-cloned (master)
$ git config --global push.default simple
```

```
[4] ~/grocery-cloned (master)
$ git push
Everything up-to-date
```

# Dissecting Git Configuration

## Defining the default editor

- Some people really don't like vim, even only for writing commit messages; if you are one of them, there is good news: you can change it by setting the core.default config parameter:

```
[1] ~/grocery (master)
$ git config --global core.editor notepad
```

# Useful techniques

- Append a new fruit to the shopping list, then try to switch branch; Git won't allow you to do so, because with the checkout you would lose your local (not yet committed) changes to the shoppingList.txt file.
- So, type the git stash command; your changes will be set apart and removed from your current branch, letting you switch to another one (berries, in this case):

# Useful techniques

- Let's take a look at the actual situation in our repository using the git log command:

# Useful techniques

- OK, let's see what happened using the git log command:

# Useful techniques

## Git commit amend - modify the last commit

- This trick is for people that don't double-check what they're doing.
- If you have pressed the enter key too early, there's a way to modify the last commit message or add that file you forgot, using the git commit command with the --amend option:

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

# Useful techniques

## Git blame - tracing changes in a file

- Working on source code in a team, it is not uncommon to have the need to look at the last modifications made to a particular file to better understand how it evolved over time.
- To achieve this result, we can use the `git blame <filename>` command.
- Let's try it inside the Spoon-Knife repository to see changes made to the `README.md` file during a specific time:

# Useful techniques

- Suppose now you found that the modification you are looking for is the one made in the d0dd1f61 commit; to see what happened there, type the git show d0dd1f61 command:

# Useful techniques

- The last tip I want to suggest is to use the Git GUI [3] ~/Spoon-Knife (master)  
\$ git gui blame README.md

The commit details at the bottom of the viewer are:

```
commit bb4cc8d3b2e14b3af5df699876dd4ff3acd00b7f
Author: The Octocat <octocat@nowhere.com> Tue Feb 4 23:38:36 2014
Committer: The Octocat <octocat@nowhere.com> Thu Feb 13 00:18:55 2014
```

Below the viewer, there is a status message: 'Create styles.css and updated README'.

# Tricks

- If you want to set up a bare repository, you only have to use the --bare option:

```
$ git init --bare NewRepository.git
```

- As you may have noticed, I called it NewRepository.git, using a .git extension; this is not mandatory, but is a common way to identify bare repositories. If you pay attention, you will note that even in GitHub every repository ends with a .git extension.

# Tricks

## **Converting a regular repository to a bare one**

- It can happen that you start working on a project in a local repository, and then you feel the need to move it to a centralized server to make it available for other people or from other locations.
- You can easily convert a regular repository to a bare one using the git clone command with the same --bare option:

```
$ git clone --bare my_project my_project.git
```

# Tricks

## Archiving the repository

- To archive the repository without including versioning information, you can use the git archive command; there are many output formats but the classic one is the .zip one:

```
$ git archive master --format=zip --output=../repoBackup.zip
```

# Tricks

- Please note that using this command is not the same as backing up folders in a filesystem; as you will have noticed, the git archive command can produce archives in a smarter way, including only files in a branch or even in a single commit; for example, by doing this you are archiving only the last commit:

```
$ git archive HEAD --format=zip --output=../headBackup.zip
```

# Tricks

## Bundling the repository

- Another interesting command is the git bundle command. With git bundle, you can export a snapshot from your repository and then restore it wherever you want.
- Suppose you want to clone your repository on another computer, and the network is down or absent; with this command, you can create a repo.bundle file of the master branch:  
`$ git bundle create ../repo.bundle master`

# Tricks

- With this other command, we can restore the bundle in the other computer using the git clone command:

```
$ cd /OtherComputer/Folder  
$ git clone repo.bundle repo -b master
```

# Tricks

- In this lesson, we enhanced our knowledge about Git and its wide set of commands. We discovered how configuration levels work, and how to set our preferences using Git by, for example, adding useful command aliases to the shell.
- Then we looked at how Git deals with stashes, providing the way to shelve then and reapply changes.