

GitHub – Getting Started

Feb-2020 –Gill Green

- **Overview**

- GitHub is a social coding platform used by millions of developers every single day.
- GitHub is used with git as the source management system.
- Apart from being a source management system, it is used to manage the entire project.
- In this session we will see how to work with Git and GitHub in an efficient way.
- We will see how to work with GitHub repositories, branches, forks and pull requests.
- The topics we will discuss are
 - Understanding Git Basics and commonly used commands of Git
 - Deep understanding of GitHub repositories
 - Create branches and forks
 - GitHub flow and pull request. Code reviews.
 - Using GitHub in an organization
 - Creating WIKIs
- Prerequisites:
 - No prior knowledge of anything is required for this course.

- Git and GitHub
- What is Git?
- Git is a popular source control system and is very widely adapted.
- Git allows many users to work with large and small software projects in a simplified way.
- Git is a distributed source management system as opposed to TFS or SVN which are centralized source management control system.
- Distributed source control system means instead of having a single centralized place for all the sources, every developer has the full history of all changes.
- Git is free and open source.
- Git was created by **Linus Torvalds** in 2005 for development of the Linux kernel, with other kernel developers contributing to its initial development.
- Its current maintainer since 2005 is Junio Hamano.

- Why Use Git?
- Git is fast and scalable.
- History is maintained locally with every user in a team
- Git is distributed and hence we can work with Git in disconnected mode.
- All the work in Git is done in the local branch and only when we need to merge the local branch with remote branch connection is required.
- Git is very powerful and yet very easy to use.
- Using branching is another benefit of Git.
- The whole Git flow is based on branching.
- Development is done in feature branch and only when we are sure about the data the feature branch is merged with the master branch, this the master branch always contains quality code.
- Pull requests are really not a feature of Git

- Why Use Git?
- Pull requests are really not a feature of Git
- Pull requests are typically added on top of the core Git feature set by sites such as GitHub.
- Pull requests enables on collaborating with code.
- Using pull requests the developer can ask for the review or a merge on other branch on the changes the developer has done.

- What is GitHub?
- GitHub is a hosting service(website) based on Git.
- GitHub runs on top of Git so it much more than just a source control for our code.
- GitHub offers both free and paid options.

- Getting the machine ready.
- Git works on all operating systems – Windows, Mac or Linux



What you need

- Download Git from <https://git-scm.com/downloads>
- An editor
 - Visual Studio Code
- GitHub account

- Setting up the environment



The screenshot shows the Git website's Downloads page. At the top left is the Git logo with the tagline "--distributed-is-the-new-centralized". To the right is a search bar. On the left sidebar, there are links for "About", "Documentation", "Downloads" (highlighted), "GUI Clients", "Logos", and "Community". Below these is a note about the "Pro Git book". The main content area is titled "Downloads" and features a box with links for "Mac OS X", "Windows", and "Linux/Unix". To the right of this is a monitor graphic displaying the "Latest source Release 2.24.1" and a button to "Download 2.24.1 for Windows". Below the platform links, it states "Older releases are available and the Git source repository is on GitHub." At the bottom of the main area, there are two sections: "GUI Clients" with a link to "View GUI Clients" and "Logos" with a link to "View Logos". The entire page is framed by an orange banner at the bottom containing the URL "git-scm.com/downloads".

git --distributed-is-the-new-centralized

Search entire site...

About
Documentation
Downloads
GUI Clients
Logos
Community

The entire **Pro Git book** written by Scott Chacon and Ben Straub is available to read online for free. Dead tree versions are available on Amazon.com.

Downloads

Mac OS X Windows Linux/Unix

Older releases are available and the Git source repository is on GitHub.

Latest source Release
2.24.1
Release Notes (2019-12-06)
Download 2.24.1 for Windows

GUI Clients

Git comes with built-in GUI tools (**git-gui**, **gitk**), but there are several third-party tools for users looking for a platform-specific experience.

[View GUI Clients →](#)

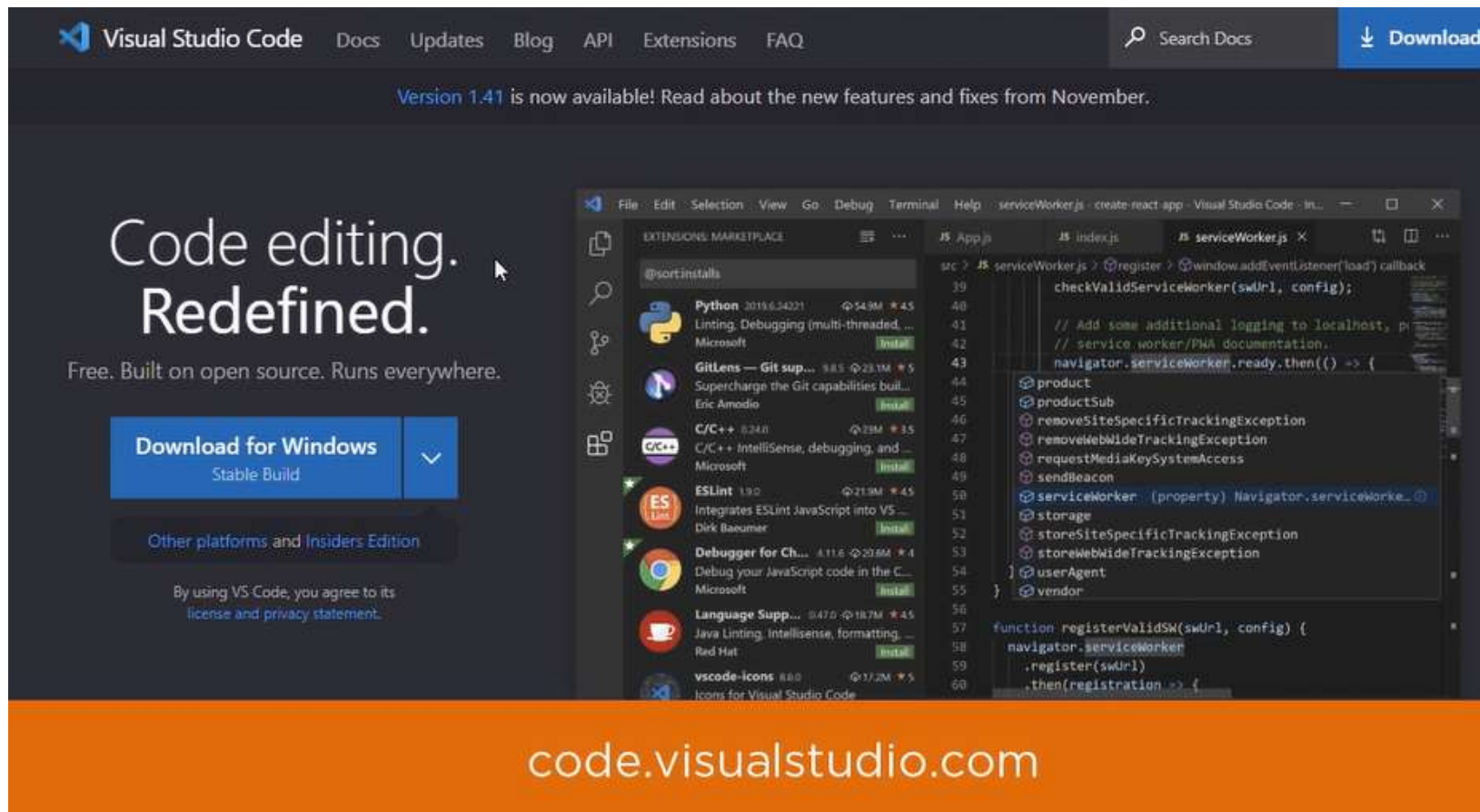
Logos

Various Git logos in PNG (bitmap) and EPS (vector) formats are available for use in online and print projects.

[View Logos →](#)

git-scm.com/downloads

- Setting up the environment



The image shows the Visual Studio Code website and a screenshot of the code editor interface. The website header includes the Visual Studio Code logo, navigation links (Docs, Updates, Blog, API, Extensions, FAQ), a search bar, and a download button. A banner below the header announces "Version 1.41 is now available! Read about the new features and fixes from November." The main content area features the text "Code editing. Redefined." and "Free. Built on open source. Runs everywhere." Below this is a "Download for Windows" button with a dropdown arrow, and a link to "Other platforms and Insiders Edition". A disclaimer at the bottom states: "By using VS Code, you agree to its license and privacy statement."

The screenshot of the code editor shows the "EXTENSIONS: MARKETPLACE" sidebar on the left, displaying a list of installed and available extensions. The main editor area shows a JavaScript file named "serviceWorker.js" with the following code:

```
src > JS serviceWorker.js > register > @window.addEventListener('load') callback
39
40 checkValidServiceWorker(swUrl, config);
41
42 // Add some additional logging to localhost, p
43 // service worker/PWA documentation.
44 navigator.serviceWorker.ready.then(() => {
45   @product
46   @productSub
47   @removeSiteSpecificTrackingException
48   @removeWebWideTrackingException
49   @requestMediaKeySystemAccess
50   @sendBeacon
51   @serviceWorker (property) Navigator.serviceWorke
52   @storage
53   @storeSiteSpecificTrackingException
54   @storeWebWideTrackingException
55   @userAgent
56   @vendor
57
58 function registerValidSW(swUrl, config) {
59   navigator.serviceWorker
60     .register(swUrl)
61     .then(registration => {
```

- Setting up the environment
- Next we need to configure git.
- Any git command starts with the word **git**

```
gill@SnowballPC MINGW64 ~  
$ git config --global user.name "Gill Cleeren"  
  
gill@SnowballPC MINGW64 ~  
$ git config --global user.email "gill@snowball.be"  
  
gill@SnowballPC MINGW64 ~  
$ git config --edit --global
```

- In the above commands we are setting up the user, email globally.
- The 3rd command is for opening the global configuration in the default editor.
- The configuration information will be saved in a file called **.gitconfig**

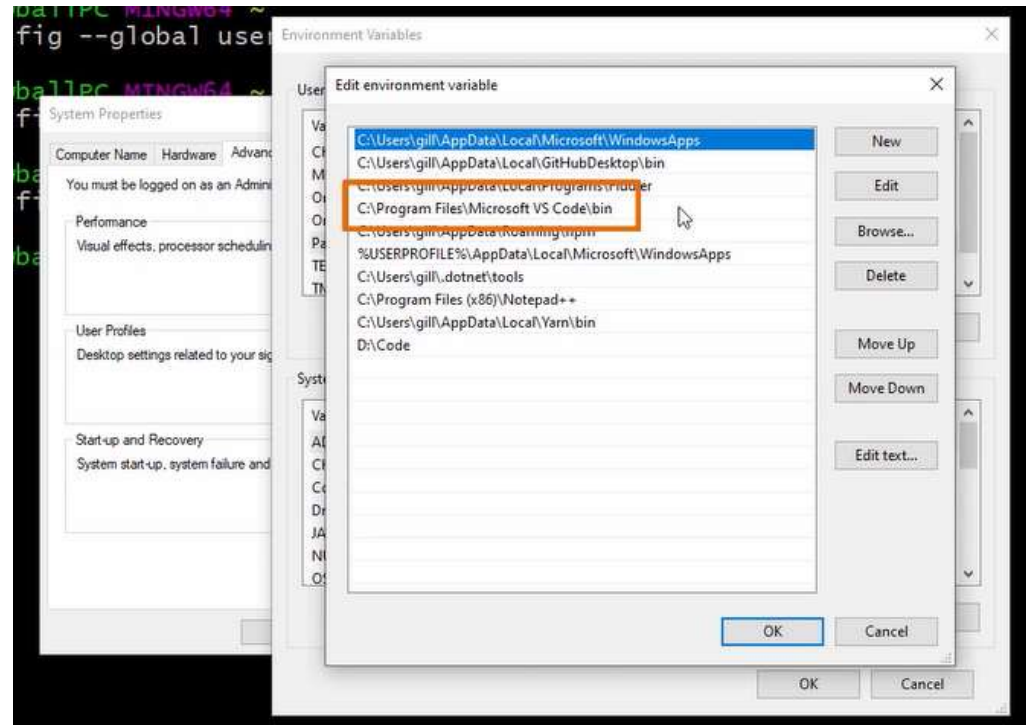
- Setting up the environment

- To set any editor as the default editor we have to install the editor of our choice and set the path in environment variables as below.

- After setting the path then

Type the following command to
Set the editor as the default
editor

```
$ git config --global core.editor "code --wait --new-window"
```



- Setting up the environment

SETTING UP GIT

```
$ git config --global user.name "Gregg Pollack"    Who gets credit for changes  
$ git config --global user.email gregg@codeschool.com    What email you use  
$ git config --global color.ui true                Pretty command line colors
```

STARTING A REPO

```
$ mkdir store
```

```
$ cd store
```

```
$ git init
```

```
Initialized empty Git repository in /Users/gregg/store/.git/
```

git metadata is stored here



GIT WORK FLOW



Jane creates README.txt file

Starts as untracked



Add file to staging area

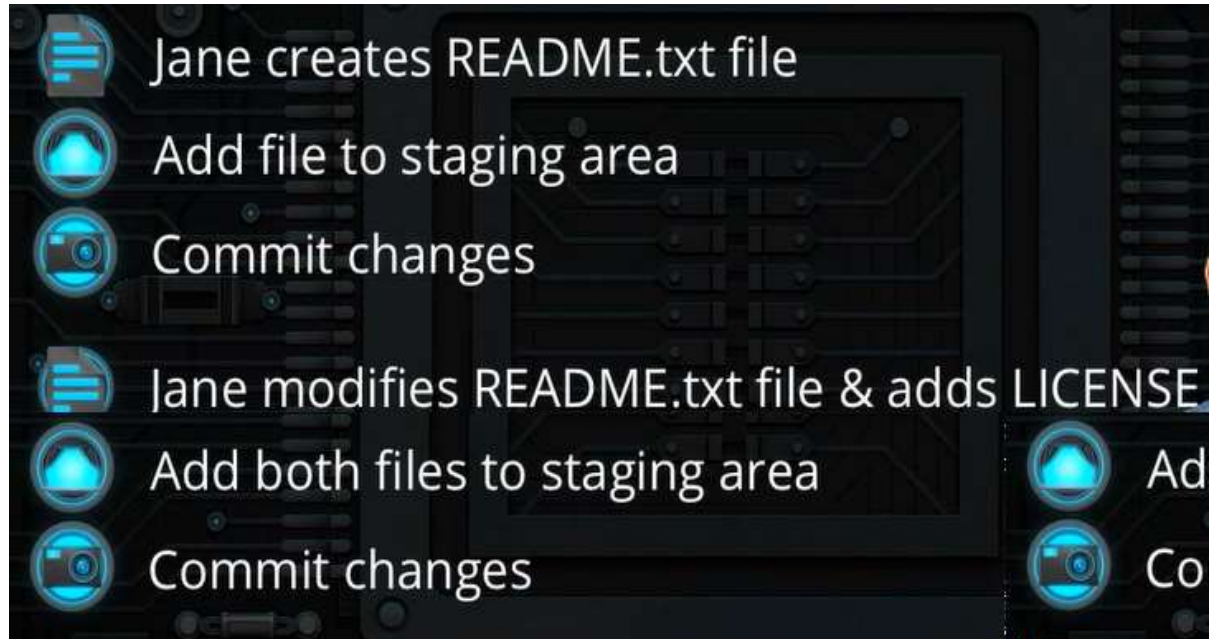
Getting ready to take a picture



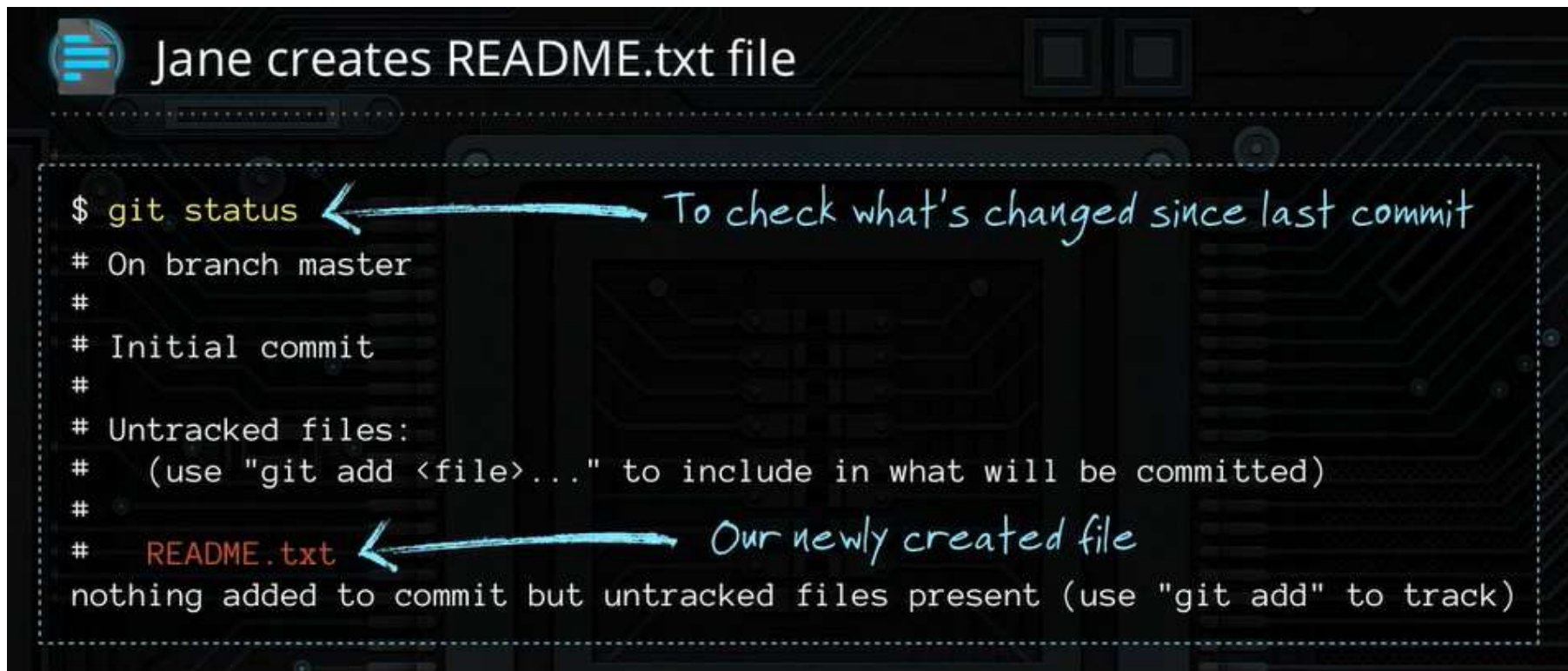
Commit changes

A snapshot of those on the stage

- Git Workflow



- Git Workflow



The image shows a terminal window with a dark background and a circuit-like pattern. At the top left is the Git logo. The title bar reads "Jane creates README.txt file". The terminal output shows the result of a `git status` command. Handwritten blue arrows point from the text "To check what's changed since last commit" to the `git status` command, and from "Our newly created file" to the `README.txt` file name in the output. The output indicates that the file is untracked and needs to be added to the commit.

```
$ git status
# On branch master
#
# Initial commit
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#   README.txt
nothing added to commit but untracked files present (use "git add" to track)
```


- Git Workflow

Add file to staging area

```
$ git add README.txt
```

```
$ git status
```

```
# On branch master
```

```
#
```

```
# Initial commit
```

```
#
```

```
# Changes to be committed:
```

```
#   (use "git rm --cached <file>..." to unstage)
```

```
#
```

```
#   new file:   README.txt
```

```
#
```

← Our staged file




- Git Workflow

Commit changes

Commit message
what work was done?

```
$ git commit -m "Create a README."
```

[master abe28da] Create a README.
1 files changed, 1 insertions(+), 0 deletions(-)
create mode 100644 README.txt



timeline

master

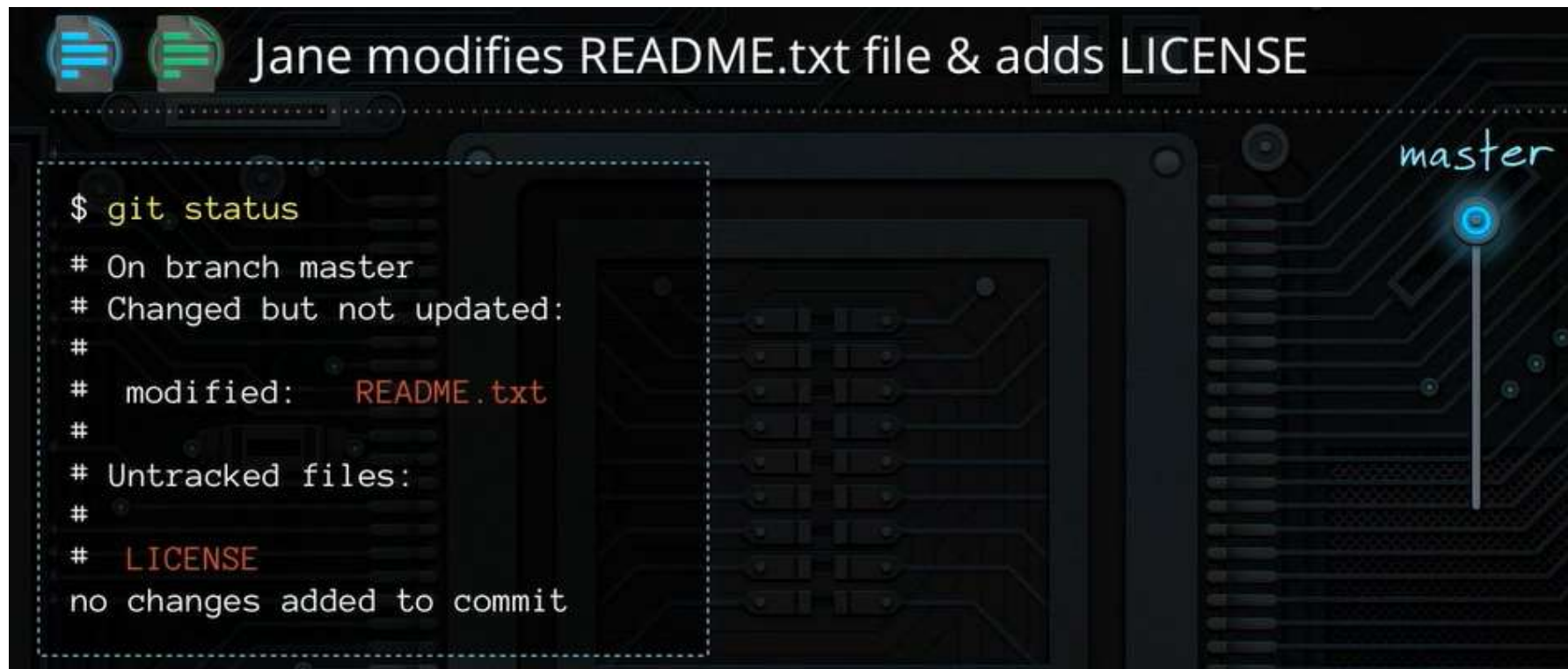
```
$ git status
```

```
# On branch master
```

```
nothing to commit (working directory clean)
```

No new or modified files since last commit

- Git Workflow



```
Jane modifies README.txt file & adds LICENSE

$ git status
# On branch master
# Changed but not updated:
#
#   modified:   README.txt
#
# Untracked files:
#
#   LICENSE
no changes added to commit
```

- Git Workflow

Add both files to staging area

```
$ git add README.txt LICENSE
```

OR

```
$ git add --all
```

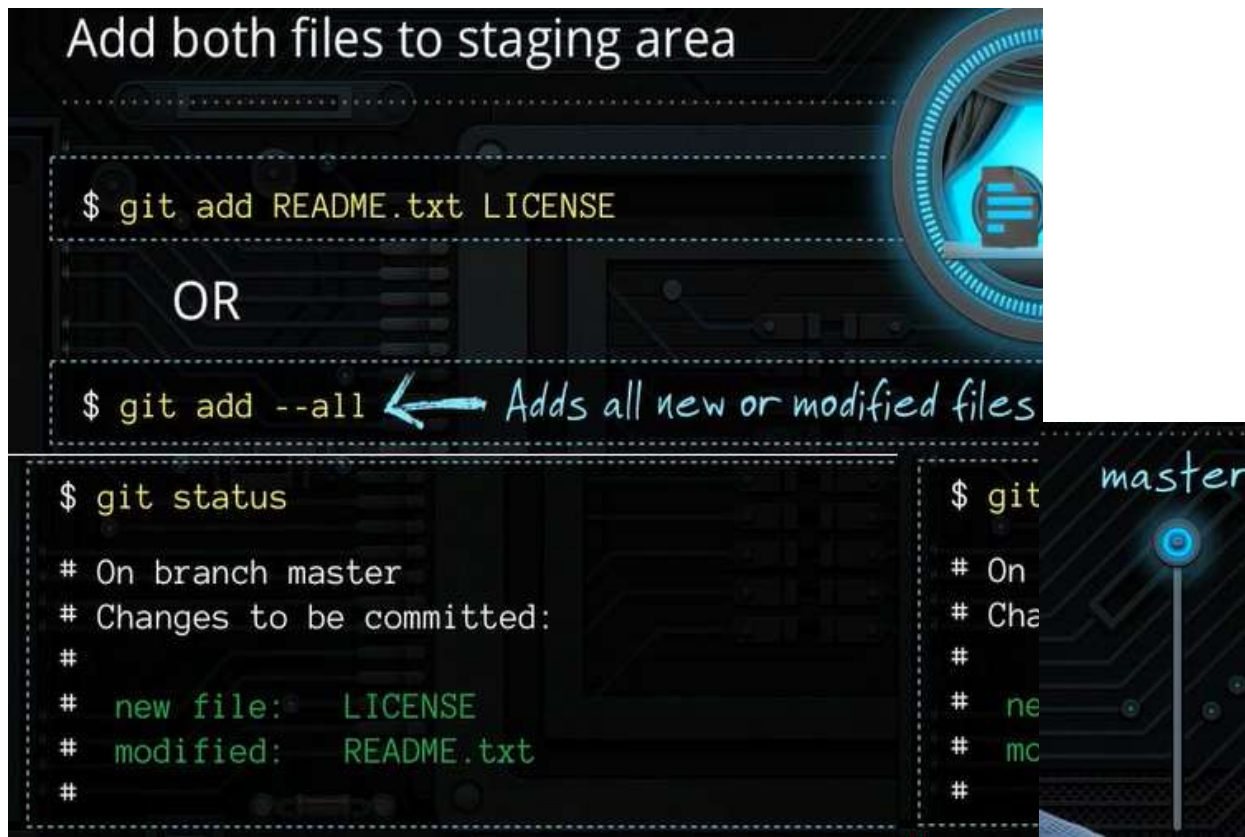
← Adds all new or modified files

```
$ git status
```

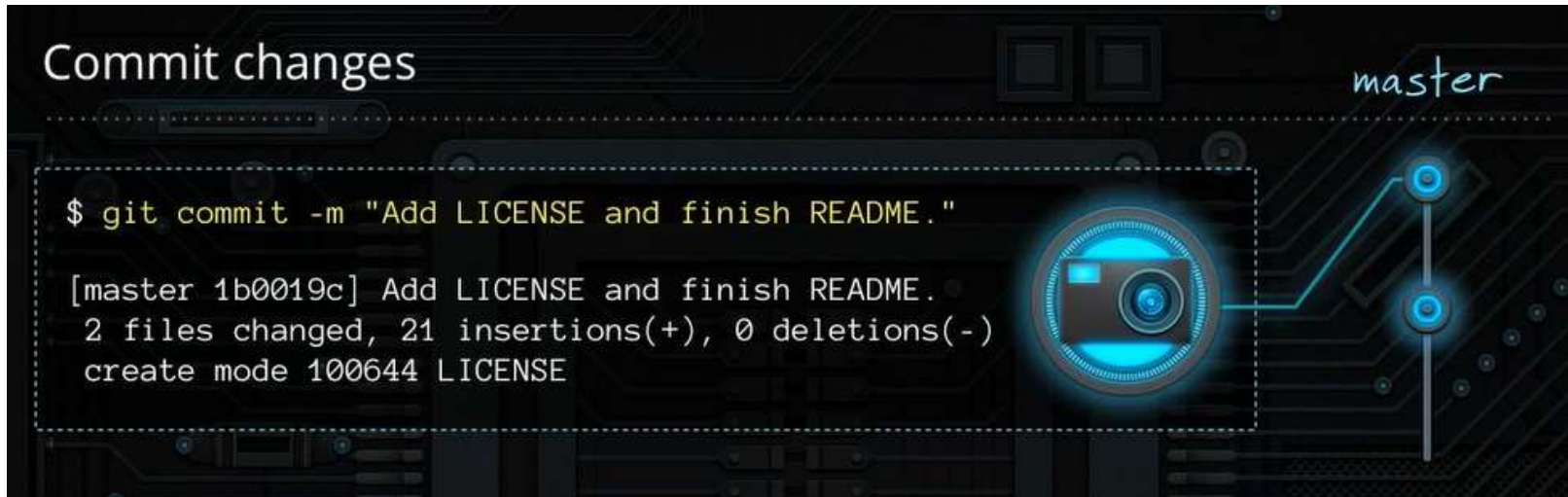
```
# On branch master
# Changes to be committed:
#
#   new file:   LICENSE
#   modified:   README.txt
#
```

```
$ git
```

master



- Git Workflow



- Git Timeline



COMMIT MESSAGES SHOULD ALWAYS BE IN PRESENT TENSE

- Different ways to Add

DIFFERENT WAYS TO ADD

```
$ git add <list of files>
```

Add the list of files

```
$ git add --all
```

Add all files

```
$ git add *.txt
```

Add all txt files in current directory

```
$ git add docs/*.txt
```

Add all txt files in docs directory

```
$ git add docs/
```

Add all files in docs directory

```
$ git add "*.txt"
```

Add all txt files in the whole project

- Staging and Remotes

GIT DIFF

 LICENSE

Copyright (c) 2012 Envy Labs LLC
...
Permission is hereby granted,
free of charge, to any person
obtaining a copy

edit →

 LICENSE

Copyright (c) 2012 Code School LLC
...
Permission is hereby granted, free
of charge, to any person obtaining
a copy

```
$ git diff
```

← Show unstaged differences since last commit

diff --git a/LICENSE b/LICENSE
index 7e4922d..442669e 100644
--- a/LICENSE
+++ b/LICENSE
@@ -1,4 +1,4 @@
← Line removed
-Copyright (c) 2012 Envy Labs LLC
← Line added
+Copyright (c) 2012 Code School LLC



- Staging And Remotes
- Suppose you made change to a file in the local repo and you don't remember what that change was.
- To figure out this change you can run the **Git Diff** command.
- The **git diff** command shows the unstaged differences since the last commit.
- Once the changes are staged the git diff command will not display any response as the file is staged now.


- Staging and Remotes

VIEWING STAGED DIFFERENCES

```
$ git add LICENSE
$ git diff
$ git diff --staged
diff --git a/LICENSE b/LICENSE
index 7e4922d..442669e 100644
--- a/LICENSE
+++ b/LICENSE
@@ -1,4 +1,4 @@
-Copyright (c) 2012 Envy Labs LLC
+Copyright (c) 2012 Code School LLC
```

No differences, since all changes are staged

View staged differences




- Staging and Remotes
- What if we don't want to commit the files that are staged

UNSTAGING FILES

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   modified:   LICENSE
#
$ git reset HEAD LICENSE
Unstaged changes after reset:
M  LICENSE
```

Unstage Tip (with arrow pointing to the command in the status output)

Refers to last commit (with arrow pointing to HEAD in the command)



A diagram on the right side of the slide illustrates the Git branch structure. It shows a vertical line representing a branch, with two blue circular commit markers. The top marker is labeled 'master' and has a grey arrow pointing to it labeled 'HEAD'. The bottom marker is also labeled 'master'. A blue arrow points from the text 'Refers to last commit' to the bottom 'master' marker.

The HEAD in the “git reset HEAD [filename]” is the last commit on the current branch.

- Staging and Remotes

DISCARD CHANGES

```
$ git status
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#   modified:   LICENSE
#
$ git checkout -- LICENSE
$ git status
nothing to commit (working directory clean)
```

← Blow away all changes since last commit

- Staging and Remotes



Instead of running two command “git add” and “git commit –m [commit message]” we can use a single command “git commit –a –m [commit message]”

- Staging and Remotes
- Suppose you made a commit and then you don't want that commit anymore.
- To undo the last commit, we use the git reset command.
- The git reset command will reset the last commit and undo the staging
 - `Git reset --soft HEAD^`
- The caret symbol after HEAD will tell move the commit one before the current HEAD.
- Now if we execute the git status command the changes from the last commit are staged.
- Now we can make changes or add files and re commit.

- Staging and Remotes

UNDOING A COMMIT

Whoops, we forgot something on that commit.

```
$ git reset --soft HEAD^
```

```
$ git status
```

```
# On branch master
```

```
# Changes to be committed:
```

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

```
#
```

```
#   modified:   README.txt
```

```
#
```

Now I can make changes, and re-commit

master

HEAD



- Staging and Remotes

ADDING TO A COMMIT

Maybe we forgot to add a file

Add to the last commit

```
$ git add todo.txt
```



New commit message



```
$ git commit --amend -m "Modify readme & add todo.txt."
```

```
[master fe98ef9] Modify readme and add todo.txt.  
2 files changed, 2 insertions(+), 1 deletions(-)  
create mode 100644 todo.txt
```

master

HEAD



Whatever has been staged is added to last commit

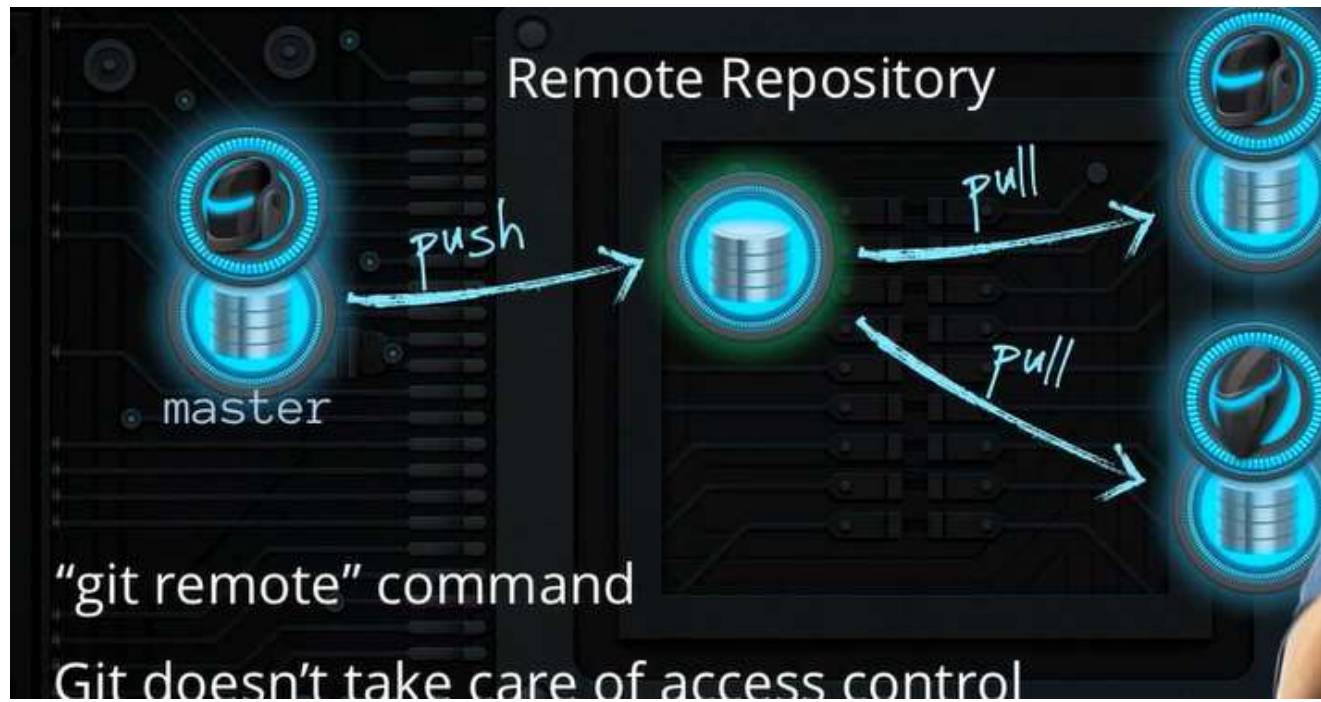
- Staging and Remotes

USEFUL COMMANDS

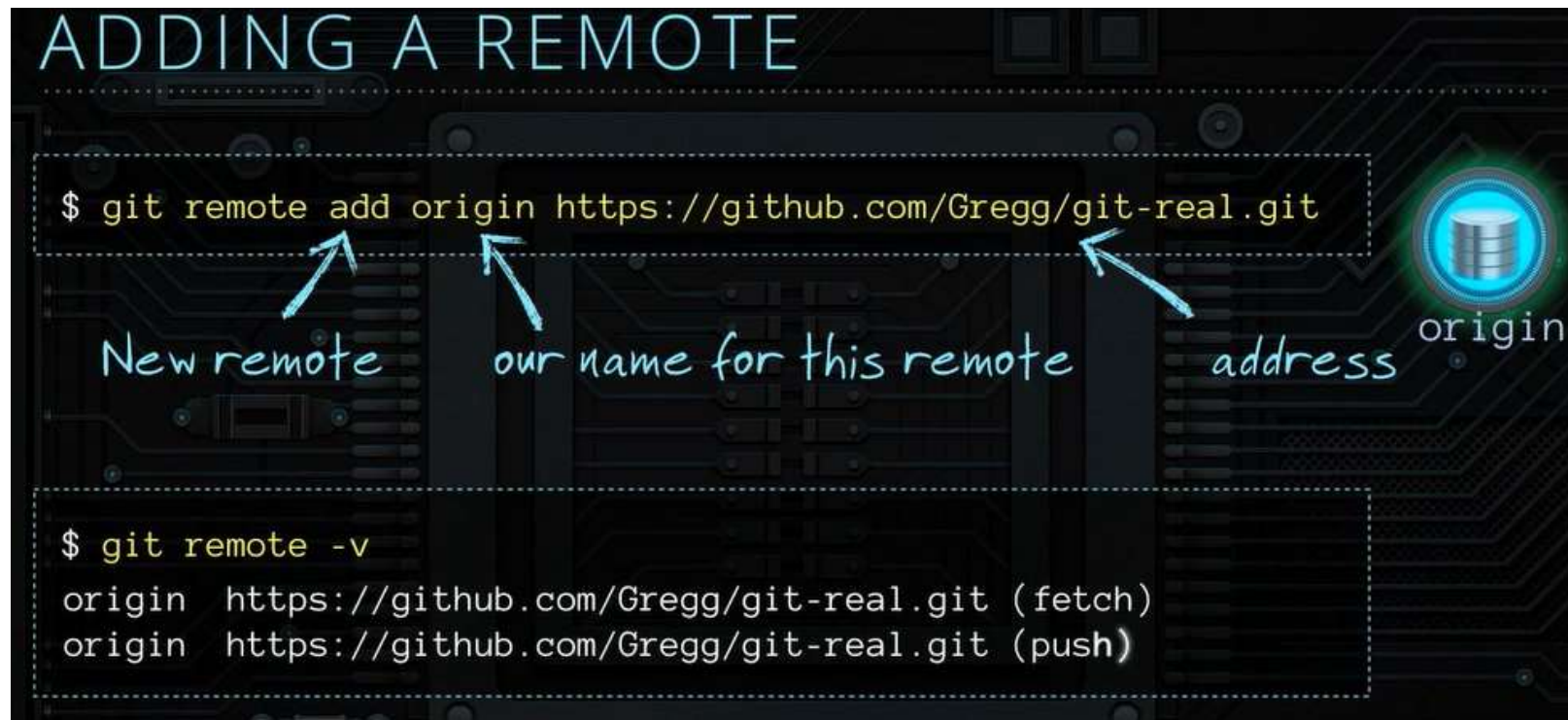
<code>\$ git reset --soft HEAD^</code>	Undo last commit, put changes into staging
<code>\$ git commit --amend -m "New Message"</code>	Change the last commit
<code>\$ git reset --hard HEAD^</code>	Undo last commit and all changes
<code>\$ git reset --hard HEAD^^</code>	Undo last 2 commits and all changes

- Staging and Remotes
- Now if we want to push the repository to the other people so that they can contribute to it.
- This is where the push and pull command is used.
- So from the local master we can push it to the remote repository and then the other people can pull it to their local repository.
- These pushing and pulling can be done using the “git remote” command.
- Git does not take care of the access control.
- For access control you need hosted solutions like GitHub, BitBucket or self managed solutions like Gitis or Gitorious

- Staging and Remotes



- Staging and Remotes



At this point we are not pushing the repository to the remote repo but we are just book marking it.
To push to remote repository we must create a GitHub account.

- Staging and Remotes

PUSHING TO REMOTE

remote repository name local branch to push

```
$ git push -u origin master
```

Username for 'https://github.com': Gregg
Password for 'https://Gregg@github.com':
Counting objects: 11, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (11/11), 1.50 KiB, done.
Total 11 (delta 0), reused 0 (delta 0)
To https://github.com/Gregg/git-real.git
* [new branch] master -> master

push

origin

master

Password caching → <https://help.github.com/articles/set-up-git>

- Staging and Remote
- After pushing we will see the output

The screenshot shows the GitHub interface for a repository named 'git-real'. The 'Commits' tab is selected, displaying the latest commit by Gregg. Below the commit details, there is a table of commit history. A blue arrow points from the 'Commits' tab to the 'history' link in the table header.

name	age	message	history
LICENSE	3 days ago	Add LICENSE and finish README. [Gregg]	
README.txt	2 days ago	Modify readme and add todo.txt. [Gregg]	
todo.txt	2 days ago	Modify readme and add todo.txt. [Gregg]	


When I click on the history or commits link we will see the same info what we see after we execute the command "git log"

- Staging and Remote
- To pull the repository from GitHub to local repository use the pull command.

PULLING FROM REMOTE

To pull changes down from the remote

```
$ git pull
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (1/1), done.
remote: Total 3 (delta 1), reused 3 (delta 1)
Unpacking objects: 100% (3/3), done.
From https://github.com/Gregg/git-real
   fe98ef9..4e67ded  master    -> origin/master
Updating fe98ef9..4e67ded
Fast-forward
 1 file changed, 1 insertion(+)
```



It's good to do this often

- Staging and Remote

WORKING WITH REMOTES

To add new remotes

```
$ git remote add <name> <address>
```

To remove remotes

```
$ git remote rm <name>
```

To push to remotes

```
$ git push -u <name> <branch>
```

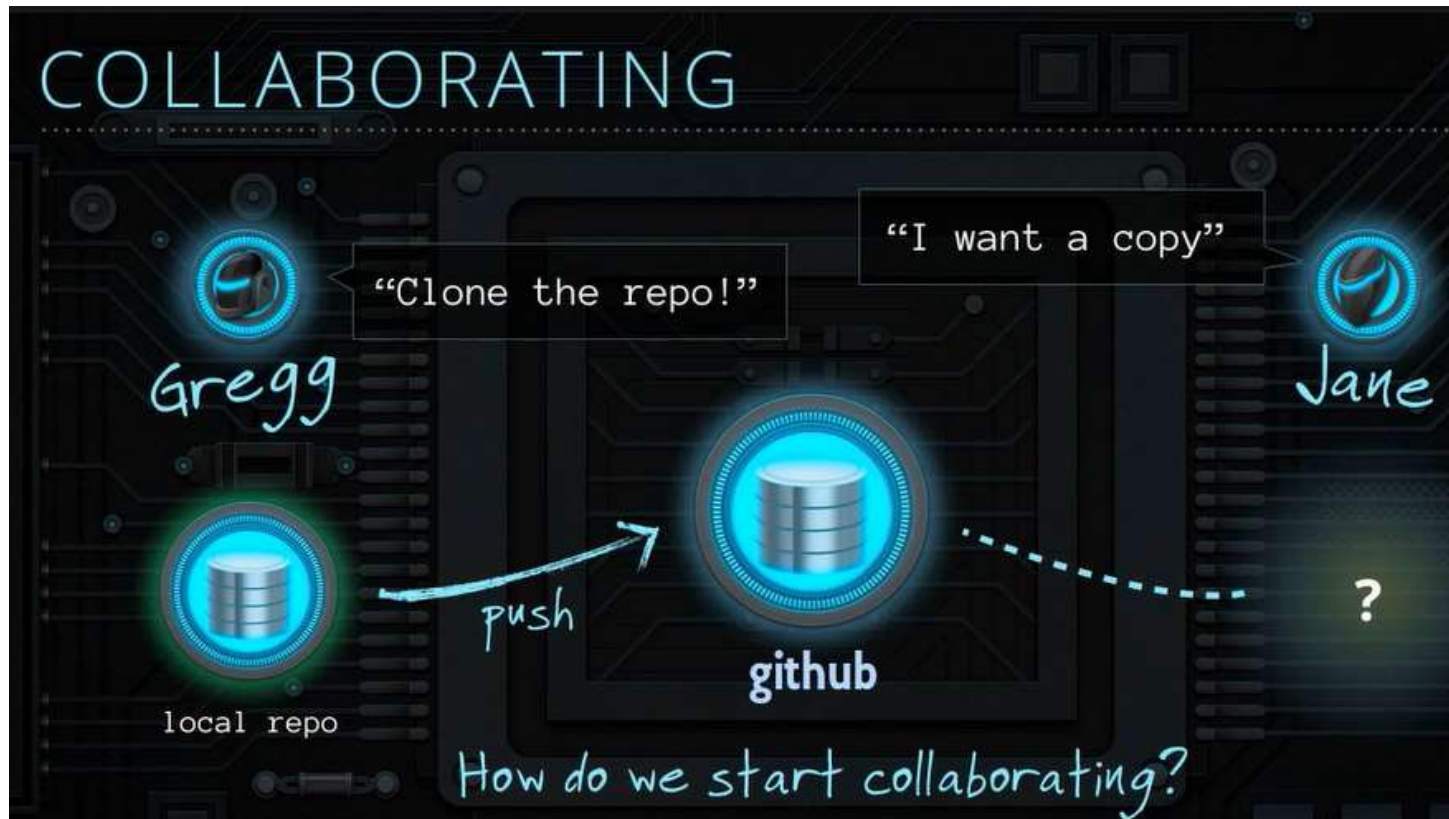
usually master

-u option in the git push command above , is if you run git push command again we don't have to specify the name of the branch , just run "git push"

- Staging and Remotes



- Cloning and Branching



- Cloning and Branching


CLONING A REPOSITORY

```
$ git clone https://github.com/codeschool/git-real.git  
Cloning into 'git-real'...
```

URL of the remote repository

```
$ git clone https://github.com/codeschool/git-real.git git-demo  
Cloning into 'git-demo'...
```

local folder name




- Cloning and Branching
- What git clone does?

GIT CLONE

- 1 - Downloads the entire repository into a new git-real directory.
- 2 - Adds the 'origin' remote, pointing it to the clone URL.

```
$ git remote -v
origin  https://github.com/codeschool/git-real.git (fetch)
origin  https://github.com/codeschool/git-real.git (push)
```
- 3 - Checks out initial branch (likely master).



sets the head

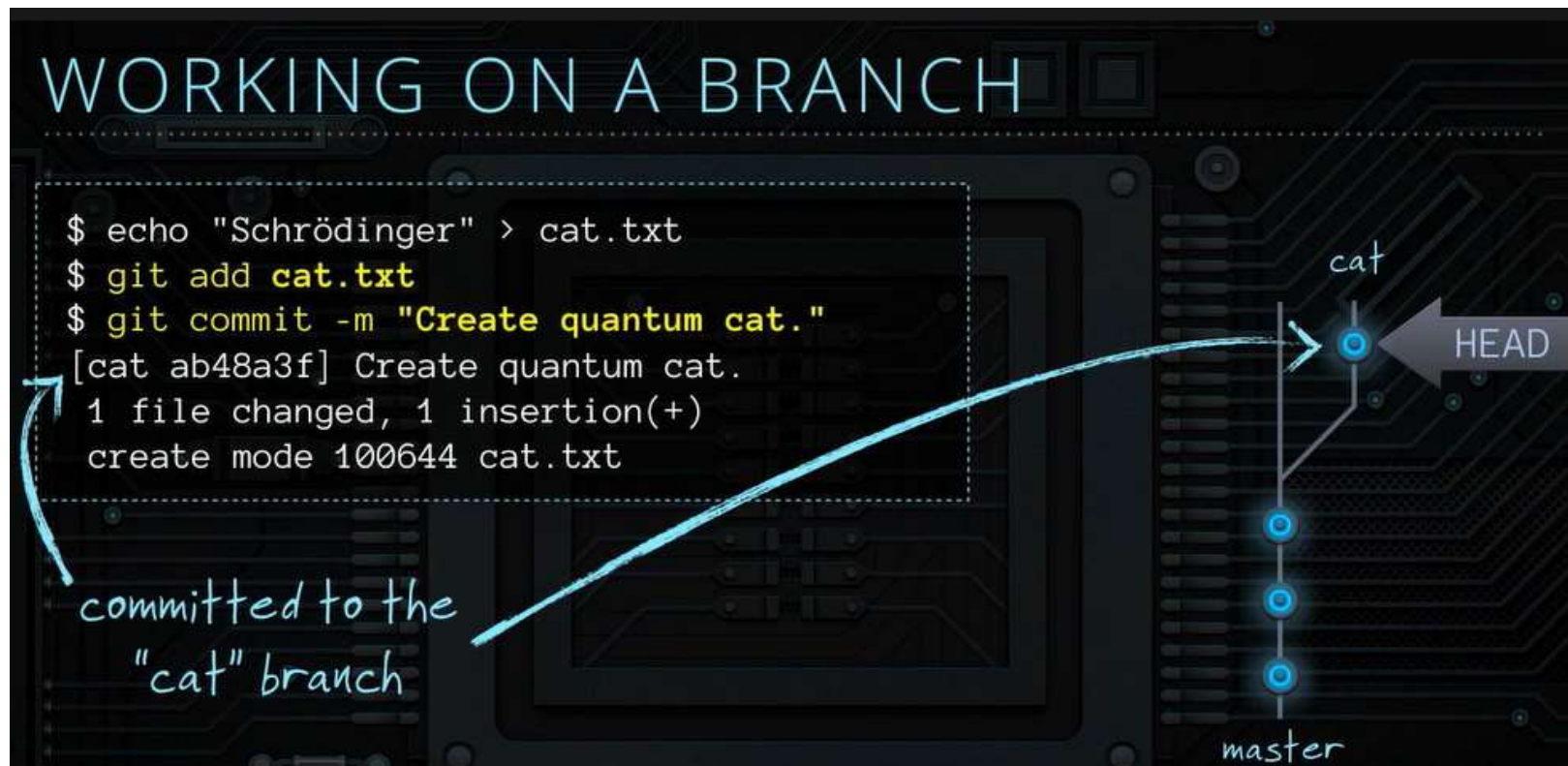
- Cloning and Branching
- After cloning the remote repo locally now if I want to create a feature branch locally then



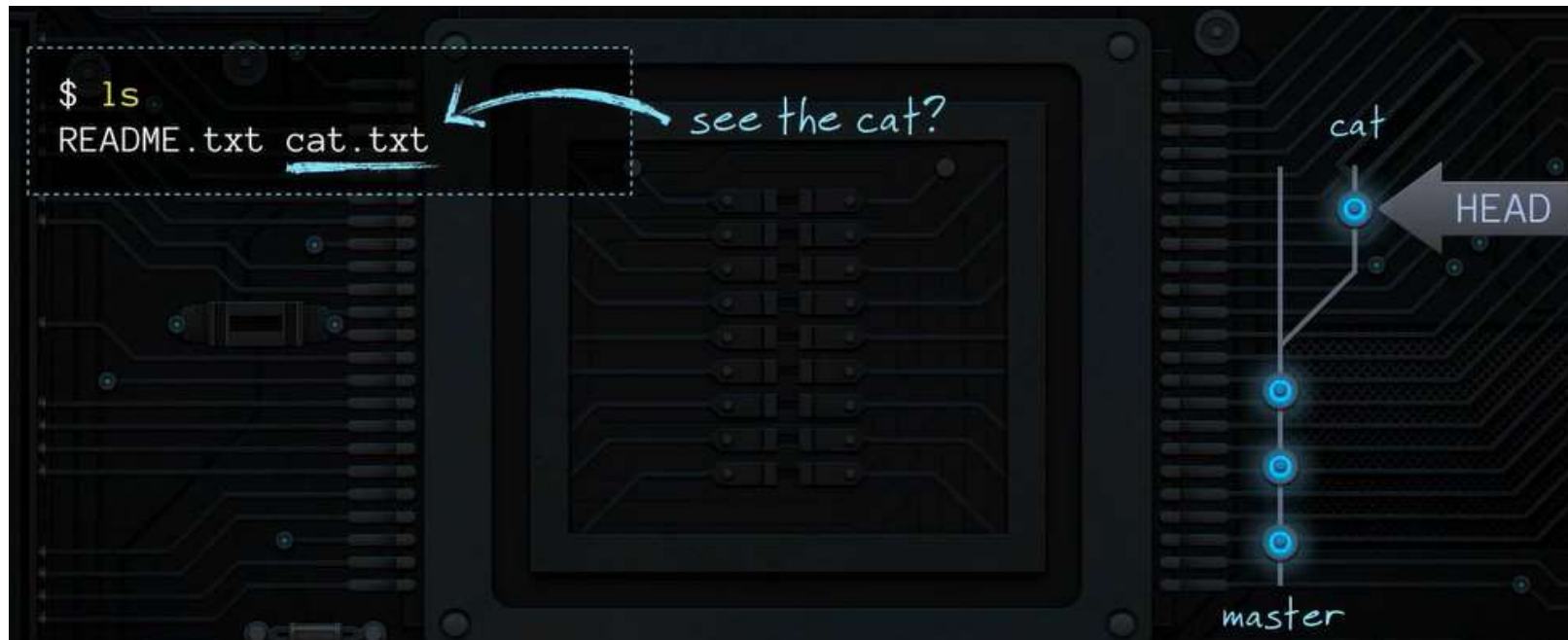
- Cloning and Branching
- To switch to the feature branch



- Cloning and Branching
- Working on the feature branch



- Cloning and Branching
- Listing the contents of the feature branch



- Cloning and Branching
- Now after checking out to master branch and listing the contents of the master branch will not show the contents of the feature branch

BACK TO MASTER

```
$ git checkout master
Switched to branch 'master'
$ ls
README.txt
```

cat.txt is gone!

and we're back on master

```
$ git log
commit 1191ceb7252c9d4b1e05c9969a55766a8adfce3b
Author: Gregg <gregg@codeschool.com>
Date: Wed Jun 27 23:11:20 2012 -0700

    Add README.
```

nothing in the log either

```
graph TD
    HEAD --> master
    master --- cat
```

- Cloning & Branching




- Cloning & Branching

TIME TO MERGE

Done with that feature branch? Time to merge it into 'master'.

```
$ git checkout master  
Switched to branch 'master'  
$
```



The diagram illustrates the Git branching model. It shows a vertical line representing the 'master' branch with four blue circular commit markers. A horizontal arrow labeled 'HEAD' points to the topmost commit on the 'master' branch. A branch named 'cat' is shown as a line branching off from the 'master' branch, with one blue circular commit marker. The background is a dark, stylized circuit board.

The HEAD points to the last commit of master branch

- Cloning & Branching

TIME TO MERGE

Done with that feature branch? Time to merge it into 'master'.

```
$ git checkout master
```

Switched to branch 'master'

```
$ ls
```

```
README.txt
```

no cat, as expected

```
$ git merge cat
```

Updating 1191ceb..ab48a3f

Fast-forward

```
cat.txt | 1 +
```

1 file changed, 1 insertion(+)

```
create mode 100644 cat.txt
```

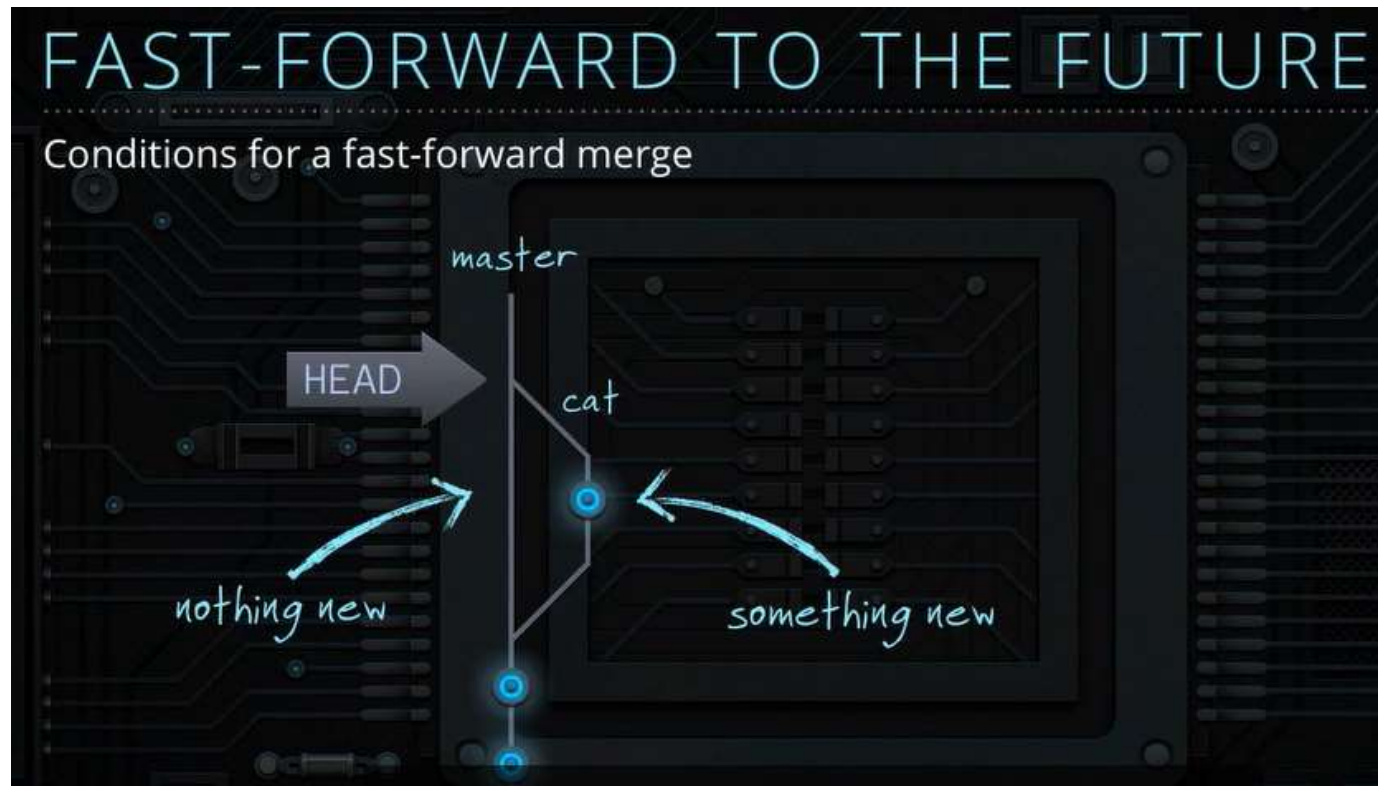
what's that?

merge brings one branch's changes into another



- Cloning & Branching
- In the previous slide you see when you fire the command “git merge cat”
You are merging the cat branch with the master branch.
- After merging it says that it is Fast-forward.
- What does this Fast-forward mean here?
- So when we create a branch and make one commit or several commits, and do nothing on the other branch it is very easy for git to merge the cat branch to master because nothing was modified in the master in the meantime.

- Cloning & Branching



- Cloning & Branching
- Now that we are done with merging we can delete the cat branch



- Cloning & Branching

NON-FAST-FORWARD

Let's work on a new admin feature.

```
$ git checkout -b admin  
Switched to a new branch 'admin'  
...  
$ git add admin/dashboard.html  
$ git commit -m 'Add dashboard'  
...  
$ git add admin/users.html  
$ git commit -m 'Add user admin'
```

creates and checks out branch

master

admin

HEAD

“Please fix the bugs on master.”

- Cloning & Branching
- While you are in the admin branch you have a message from your team member that there is a bug in the master branch and you need to fix it.

BUG FIXING ON MASTER

Time to put out the fire. We'll get back to that admin branch later.

```
$ git checkout master
Switched to branch 'master'
$ git branch
admin
* master
$ git pull

..
$ git add store.rb
$ git commit -m 'Fix store bug'

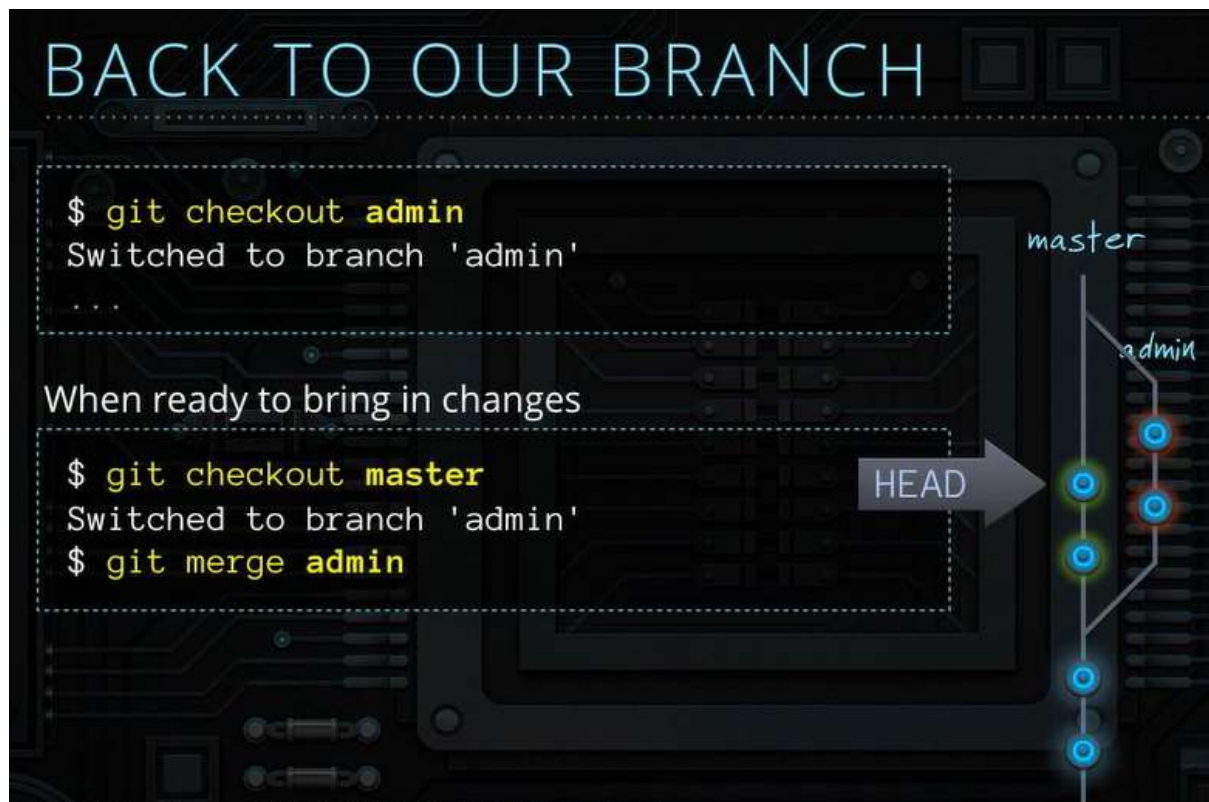
..
$ git add product.rb
$ git commit -m 'Fix product'

..
$ git push
```



The diagram illustrates the state of a Git repository. It shows two vertical lines representing branches: 'master' and 'admin'. The 'master' branch has four commit nodes (circles) stacked vertically. The top commit is highlighted with a blue glow and a grey arrow labeled 'HEAD' points to it. The 'admin' branch has two commit nodes, branching off from the second commit of the 'master' branch. The commit nodes on 'admin' are also highlighted with blue glows.

- Cloning & Branching
- So in the previous slide you switched to master branch and did some emergency bug fixing (or made changes) in the master branch.
- Now we need to checkout to the admin branch.



- Cloning & Branching
- In the previous slide you merged the admin branch to master but suddenly you will be sent to the default editor and asks the reason for merge.

AND SUDDENLY...

Git uses Vi if no default editor is set to edit commit messages.

DON'T PANIC

```
1 Merge branch 'admin'
2
3 # Please enter a commit message to explain why this merge is necessary,
4 # especially if it merges an updated upstream into a topic branch.
5 #
6 # Lines starting with '#' will be ignored, and an empty message aborts
7 # the commit.
```

:wq + hit Enter to write (save) & quit

Vi commands

j	down	k	up	ESC	leave mode	:wq	save & quit
h	left	l	right	i	insert mode	:q!	cancel & quit

This is because we merged two branches with two sets of changes git had to do a recursive merge.

- Cloning & Branching

RECURSIVE MERGING

Git can't fast-forward since changes were made in both branches.

```
Merge made by the 'recursive' strategy.  
0 files changed  
create mode 100644 admin/dashboard.html  
create mode 100644 admin/users.html
```

A commit was created to merge the two branches.

```
$ git log  
commit 19f735c3556129279bb10a0d1447dc5aba1e1 fa9  
Merge: 5c9ed90 7980856  
Author: Jane <Jane@CodeSchool.com>  
Date: Thu Jul 12 17:51:53 2012 -0400
```

Merge branch 'admin'

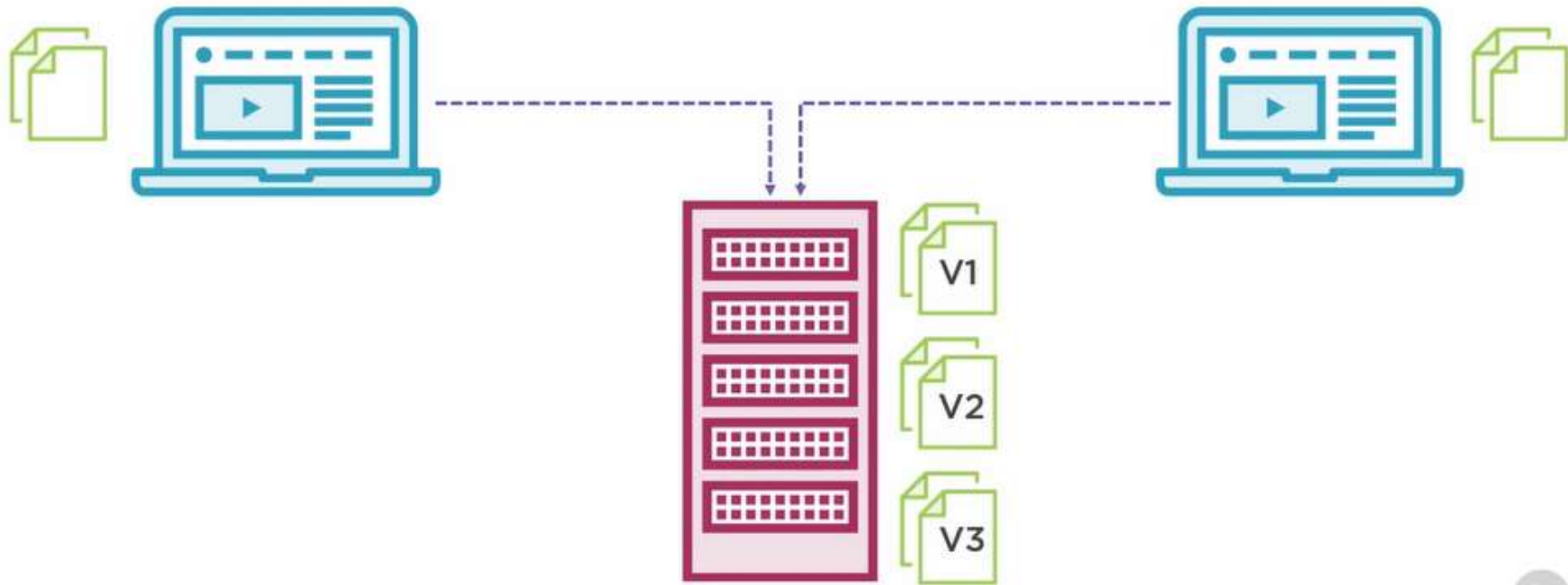


- Git can't fast-forward here since changes were made in both branches.
- When recursive merge happens git creates a commit right there when the branches are merged to gather.
- If you see "git log" you will see that a merge commit was actually created in the log.
- It doesn't contain any files, but it simply says that at this point master and admin became one branch.

- Foundations of Git
- What is actually distributed source code management.
- In Centralized source code management there is a centralized server.
- On this Centralized server we have a copy of the source centrally on the server.
- Developers will save the copy of the code on this centralized server.
- The other developers can now pull down the code from this centralized server.
- In a centralized system we work with changed sets.
- Changed set is the number of changes done as a whole.

- Foundations of Git

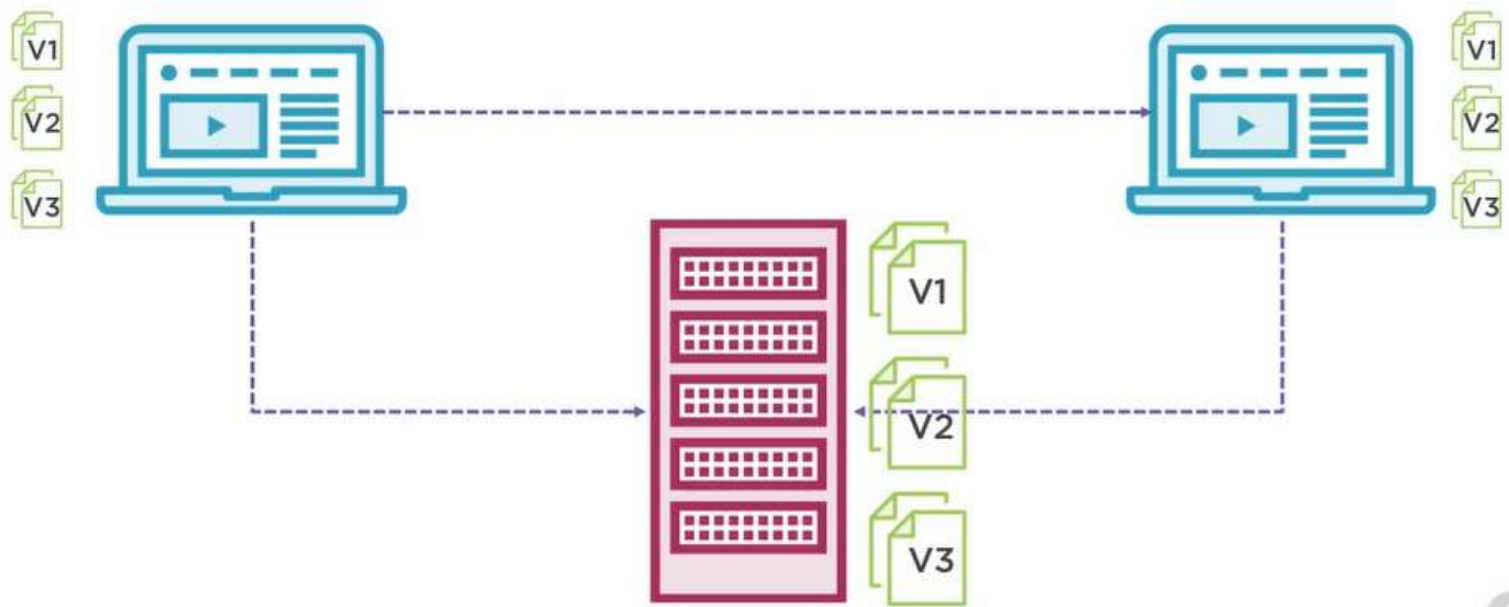
Centralized Source Code Management



- Foundations of Git
- In distributed source version control the developer will clone the entire repository and therefore gets the entire history on the local machine.
- This way of working therefore does not require a central store but typically there will be one.
- Since each developer will have the local copy of the source code and the history as well.
- We can create local branches and only when we are convinced with the changes we can send it to the central repository.
- The changes done locally can then be further distributed.
- The pushing and pulling only will require to stay connected with the central repository.
- The central store in our case will be GitHub.

- Foundations of Git

Distributed Version Control



- Foundations of Git
- The 3 states of Git are committed, modified and staged.
-

- Stashing in Git
- English Meaning of STASH : store (something) safely in a hidden or secret place.
- So let's say associate 'X' is working on a feature branch and he's about part way through a commit when X's boss calls and says, that something wrong with the server.
- X now has to make a commit right away to the master branch.
- But Mr. X is right in the middle of this html file, about halfway done with this "feature" branch, and can't go ahead and commit it halfway through.
- So, what X will do? Well, this is where stashing comes in.
- Stashing allows X to take some files that may not make up a full commit and store them away in a temporary area so that Mr. X can restore them at a later time.

- Stashing

Gregg is halfway done with work on the "gerbils" branch, but an issue with "master" needs fixing NOW

```
$ git diff
diff --git a/index.html b/index.html
index d36fac4..d2923a8 100644
--- a/index.html
+++ b/index.html
@@ -7,6 +7,7 @@
<body>
  <nav>
    <ul>
+    <li><a href="gerbil
      <li><a href="cat.html">Cats</a></li>
      <li><a href="dog.html">Dogs</a></li>
    </ul>
```

- Stashing
- So in our case, if we run 'git stash save, ' it's going to take those files that haven't been completed yet and it's going to save them away in a temporary area.
- It's also going to restore the state from the last commit.
- So, in this case, if the command 'git diff, ' is executed there's no feature branch.
- So now we can safely go back over to our master branch from here and make all the changes that we need.
- We can pull down updates, we can make commits, and we can push up the changes.
- Now when we're ready to resume working on our feature branch, we can go ahead and check out that particular branch and then run the command, 'git stash apply'
- This will rerun the changes that we stashed away before so that we can continue working on that code and eventually make a commit.
- Every time we run 'git stash save' it pushes that stash onto the stash stack.

- Stashing



- Stashing
- So if we run '**git stash list**' we'll see a list of all of the stashes that we've used.
- We'll see WIP as in work-in-progress on master, that's the branch where we stashed and it gives the last commit before we stashed.
- Because a stash is not a commit, it's giving the commit that was right before the time that we stashed.
- The stashes are each given a name that you can reference if you want to apply a certain stash.
- So, if we wanted to apply just the middle stash, stash number one, we could call '**git stash apply stash one**' (see the slide), and it would be applied into the code.
- Stash zero, is the one at the top of the stack, which is going to be applied by default if we don't specify a stash by name.
- When we run the stash apply command it's going to apply our stash but it's not going to pop our stash from the stash stack.

- Stashing

APPLY STASHES

```
$ git stash list
stash@{0}: WIP on master: 686b55d Add wolves.
stash@{1}: WIP on gerbils: b2bdead Add dogs.
stash@{2}: WIP on gerbils: b2bdead Add dogs.
```

You can have multiple stashes

Stash names are shown in the list

```
$ git stash apply stash@{1}
# On branch gerbils
# Changes not staged for commit:
#
#   modified:   index.html
```

"stash@{0}" is the default when applying; specify the stash name to apply a different one

- Stashing

DROP STASHES

"git stash drop" discards a stash

```
$ git stash list  
stash@{0}: WIP on gerbils: b2bdead Add dogs.
```

Stash has been applied
but it's still here

```
$ git stash drop  
Dropped (6dc716f...)
```

Delete it from list

```
$ git stash list  
[Empty]
```

Old stash is gone!

- Stashing

SHORTCUTS

shortcut

`git stash`

`git stash apply`

`git stash drop`

`git stash pop`

same as

`git stash save`

`git stash apply stash@{0}`

`git stash drop stash@{0}`

`git stash apply`

`git stash drop`

- Stashing
- There's some intelligent defaults for running stash commands for example,
- we can just run 'git stash' and it's the same thing as running 'git stash save. '
- If we run 'git stash apply' it's going to run the stash at the top of the stack, which is going to be stash zero.
- 'Git stash drop' is going to automatically drop the stash at the top of the stash stack.
- Lastly, there's the 'git stash pop' command, which runs 'git stash apply' and 'git stash drop. '
- So it actually applies the stash and then pops it off of the top of the stack.

