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 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

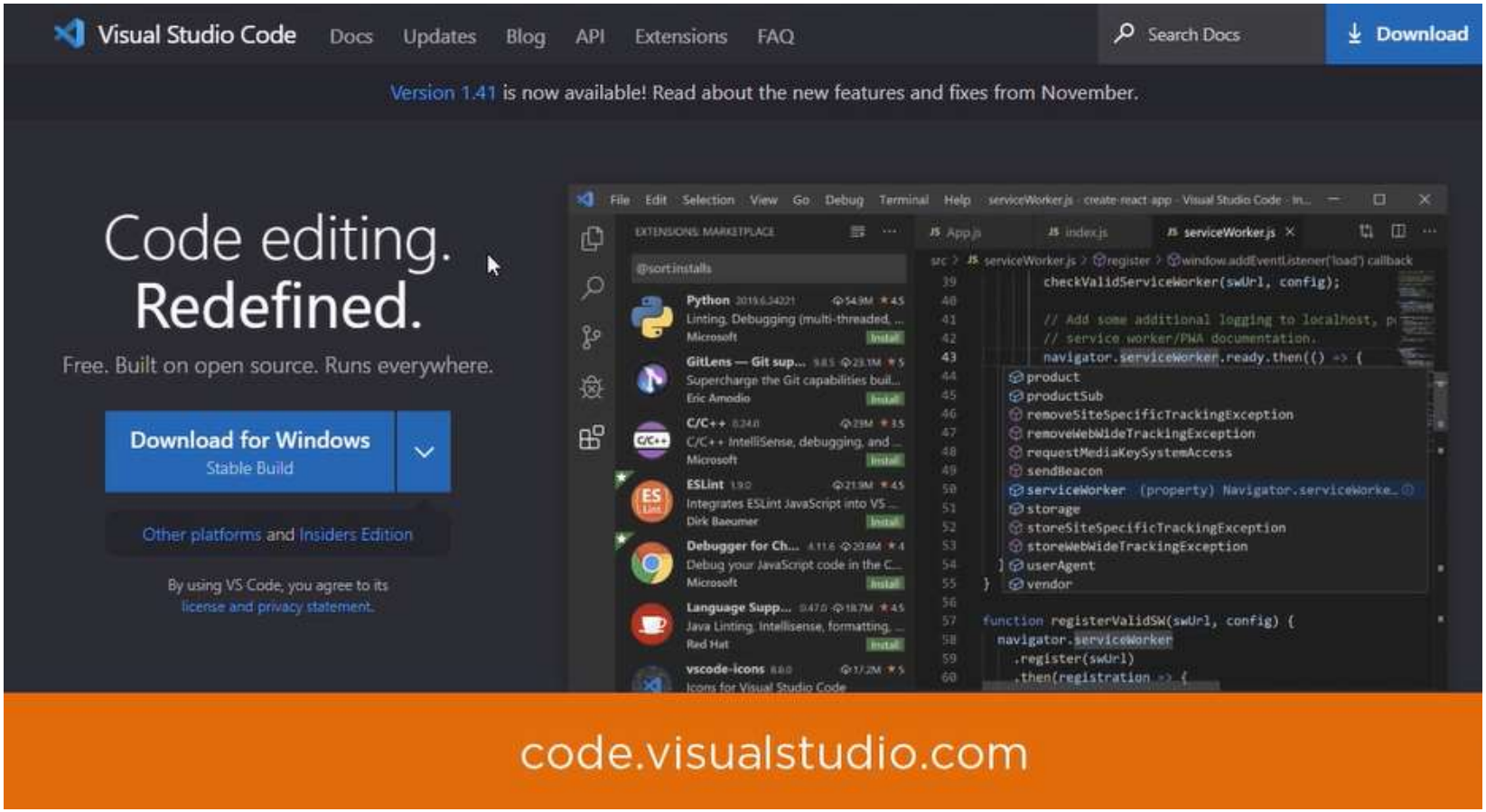
What you need

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

- Setting up the environment

git-scm.com/downloads

- Setting up the environment



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

```
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 **<u>.gitconfig</u>**

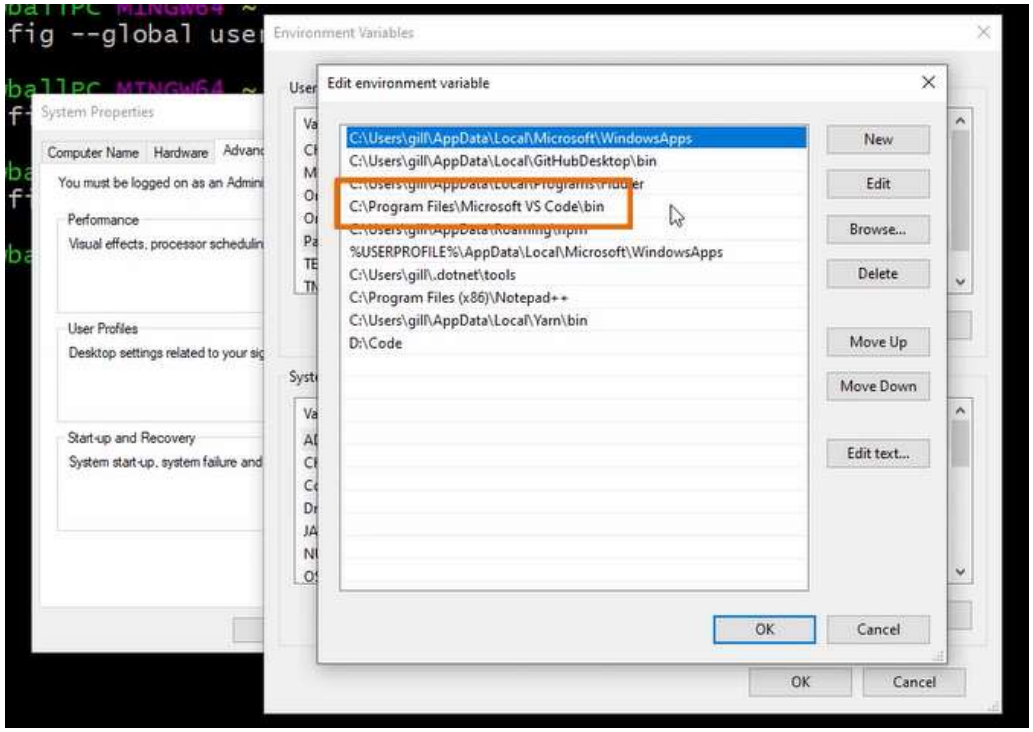
- <u>Setting up the environment</u>
- To set any editor as the default editor we have to install the editor of our choice and set the path in environment variables as below.
- After setting the path then

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

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

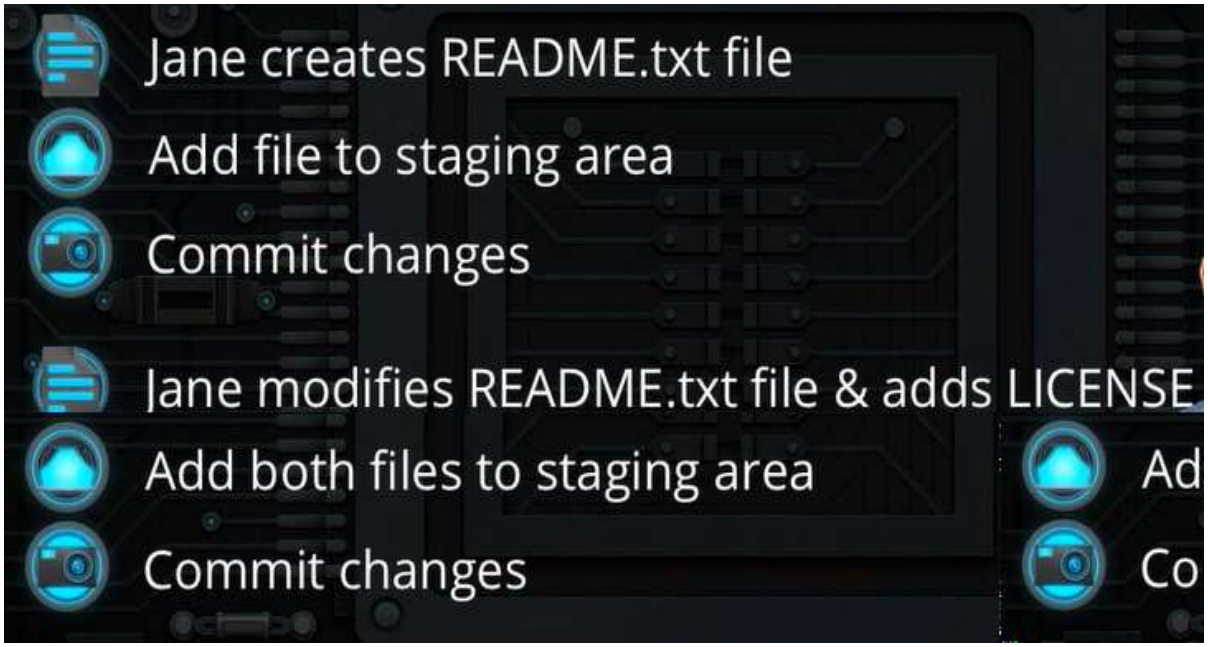
Add file to staging area

Getting ready to take a picture

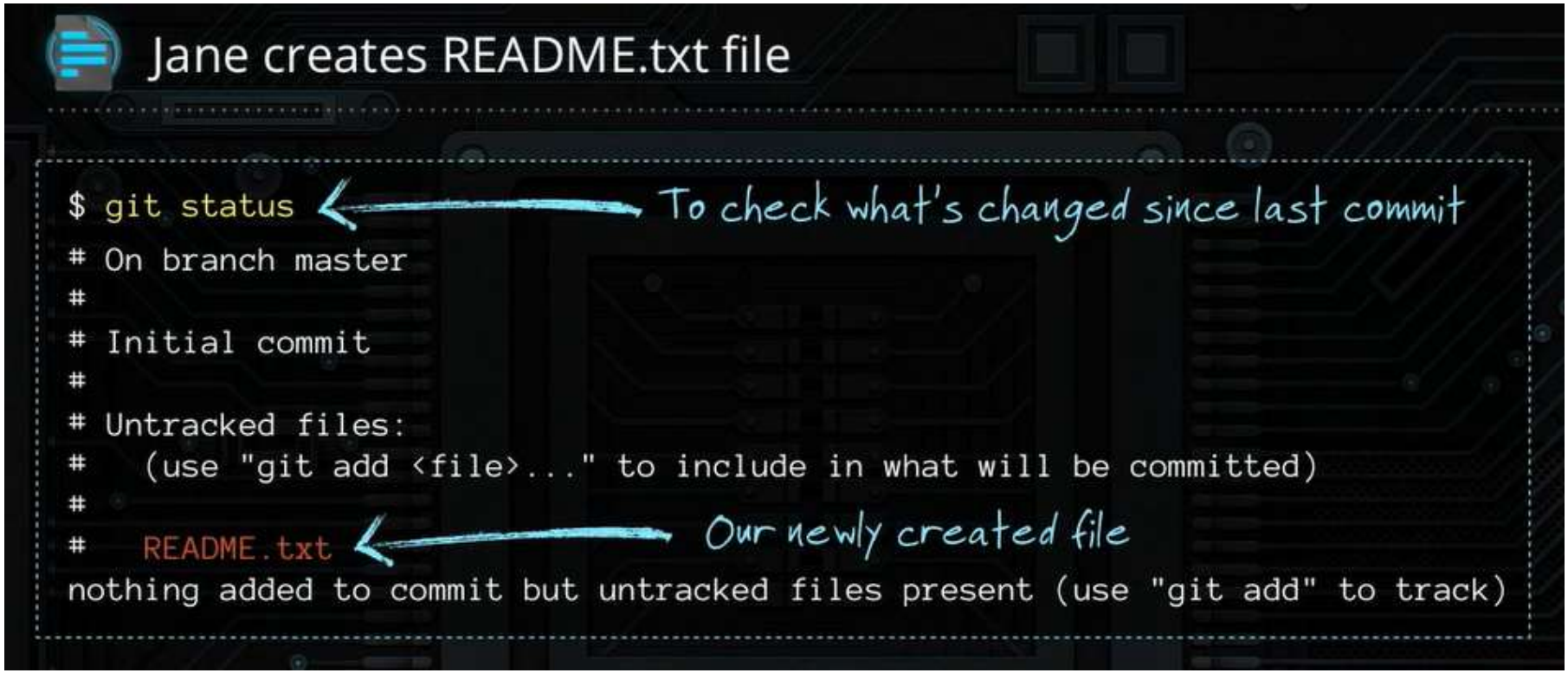
Commit changes

A snapshot of those on the stage

- Git Workflow



- Git Workflow



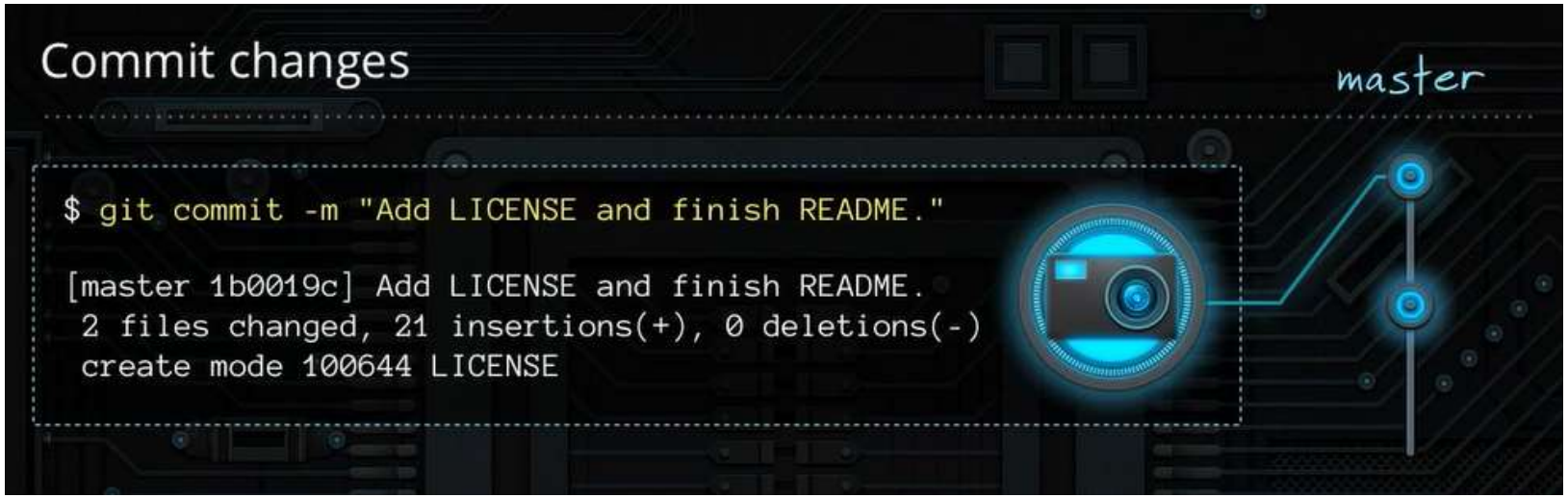
- Git Workflow

- Git Workflow

- Git Workflow

- Git Workflow

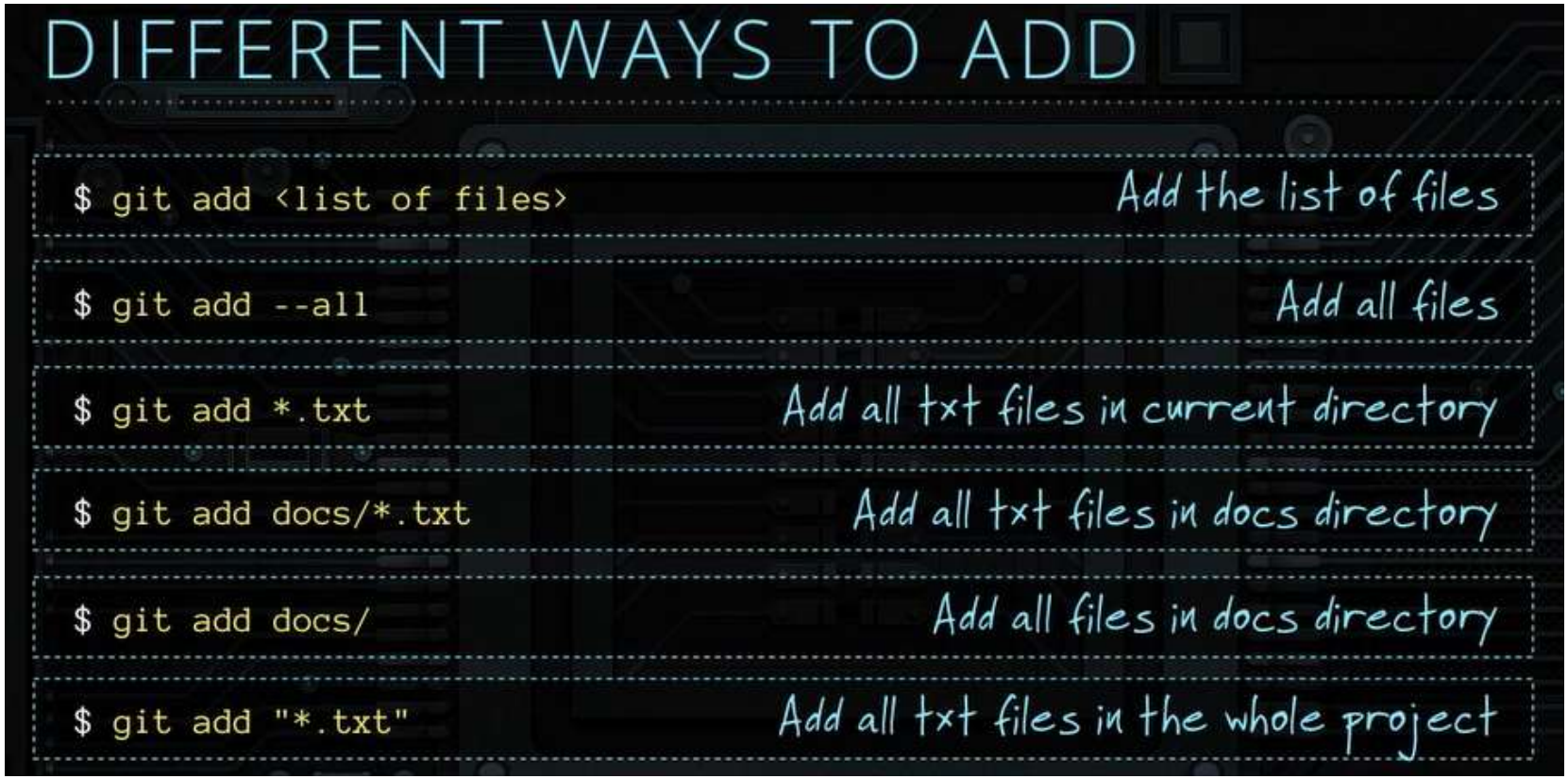
- Git Workflow



- Git Timeline

COMMIT MESSAGES SHOULD ALWAYS BE IN PRESENT TENSE

- Different ways to Add



- Staging and Remotes

- Staging And Remotes
- Suppose you mad 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

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

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.

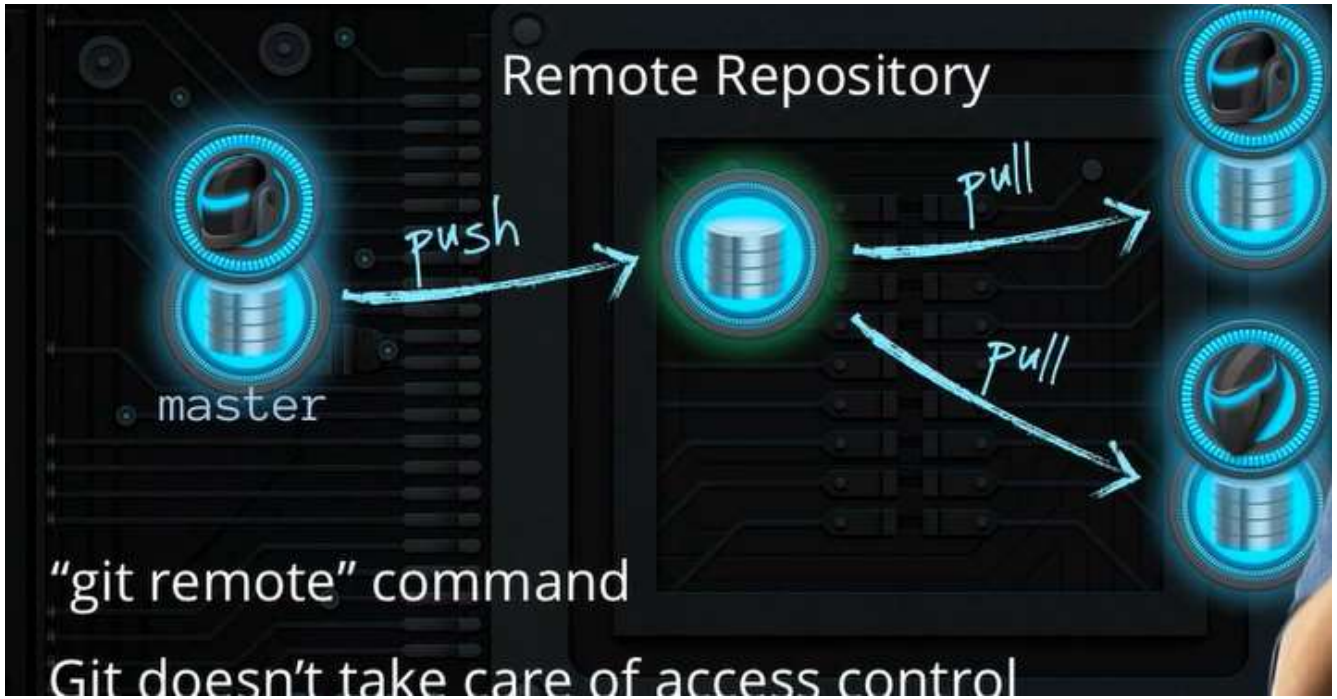
- Staging and Remotes

- Staging and Remotes

- Staging and Remotes

- 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 Gitosis or Gitorious

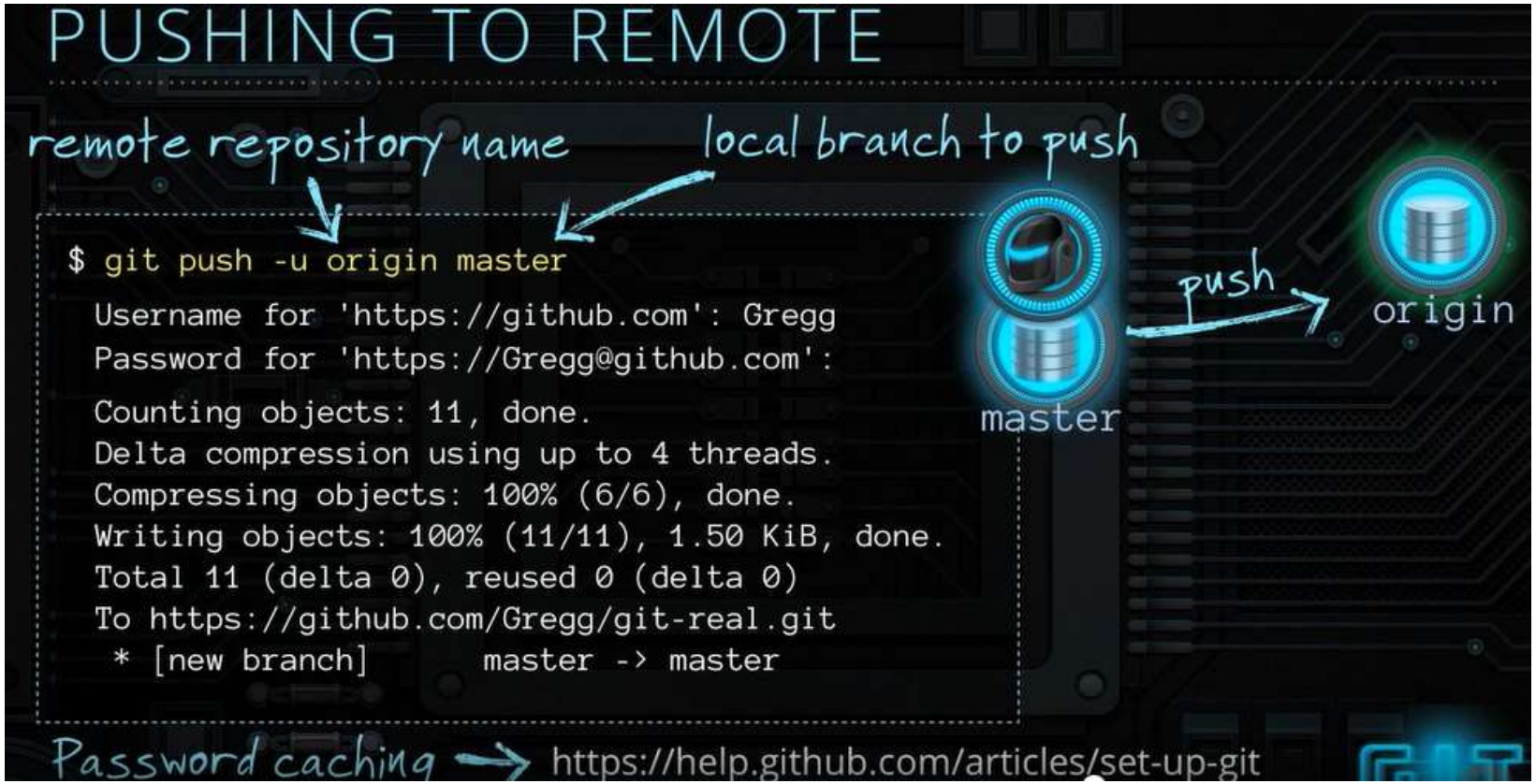
- Staging and Remotes



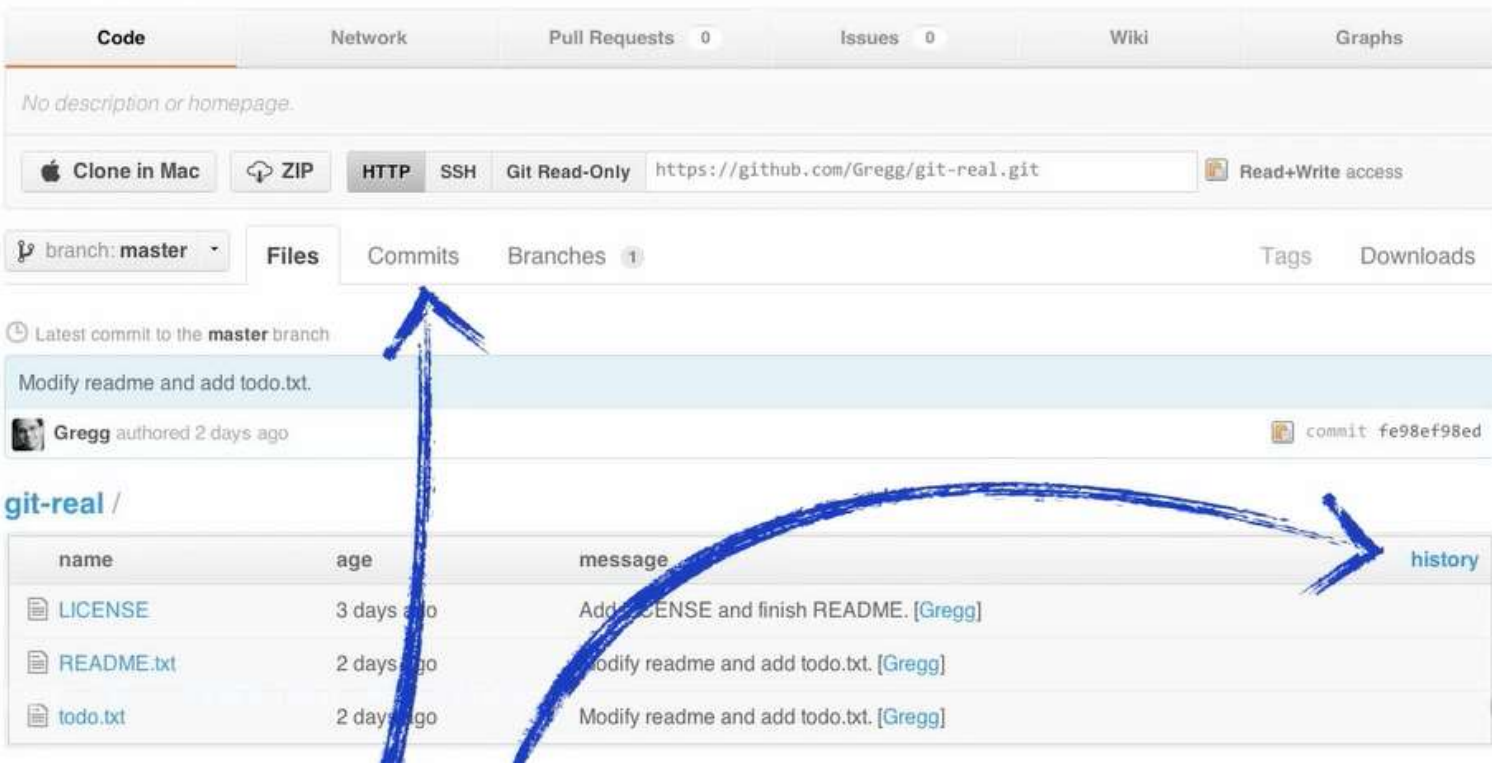
- Staging and Remotes

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

- Staging and Remotes

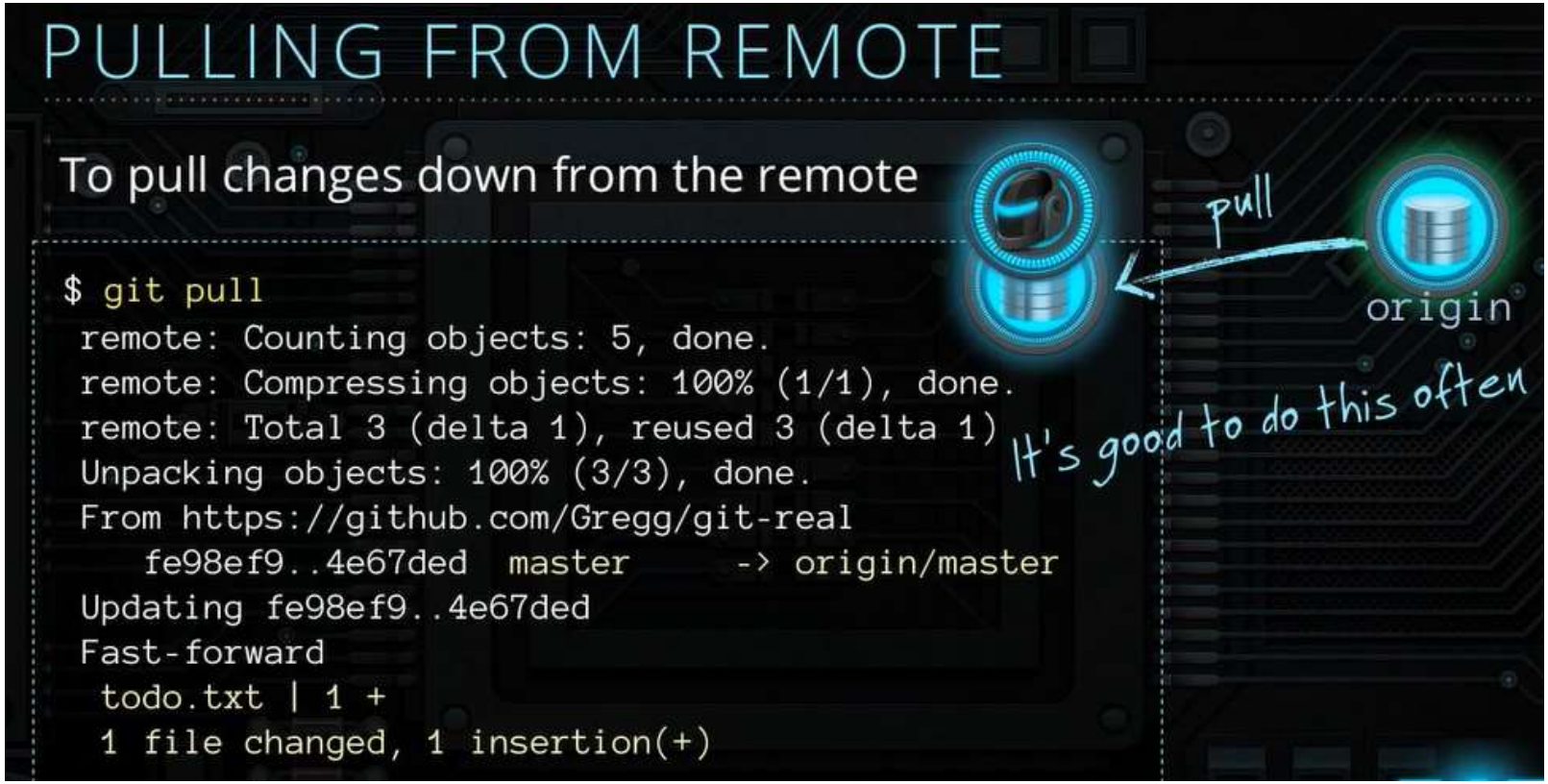


- Staging and Remote
- After pushing we will see the output



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.

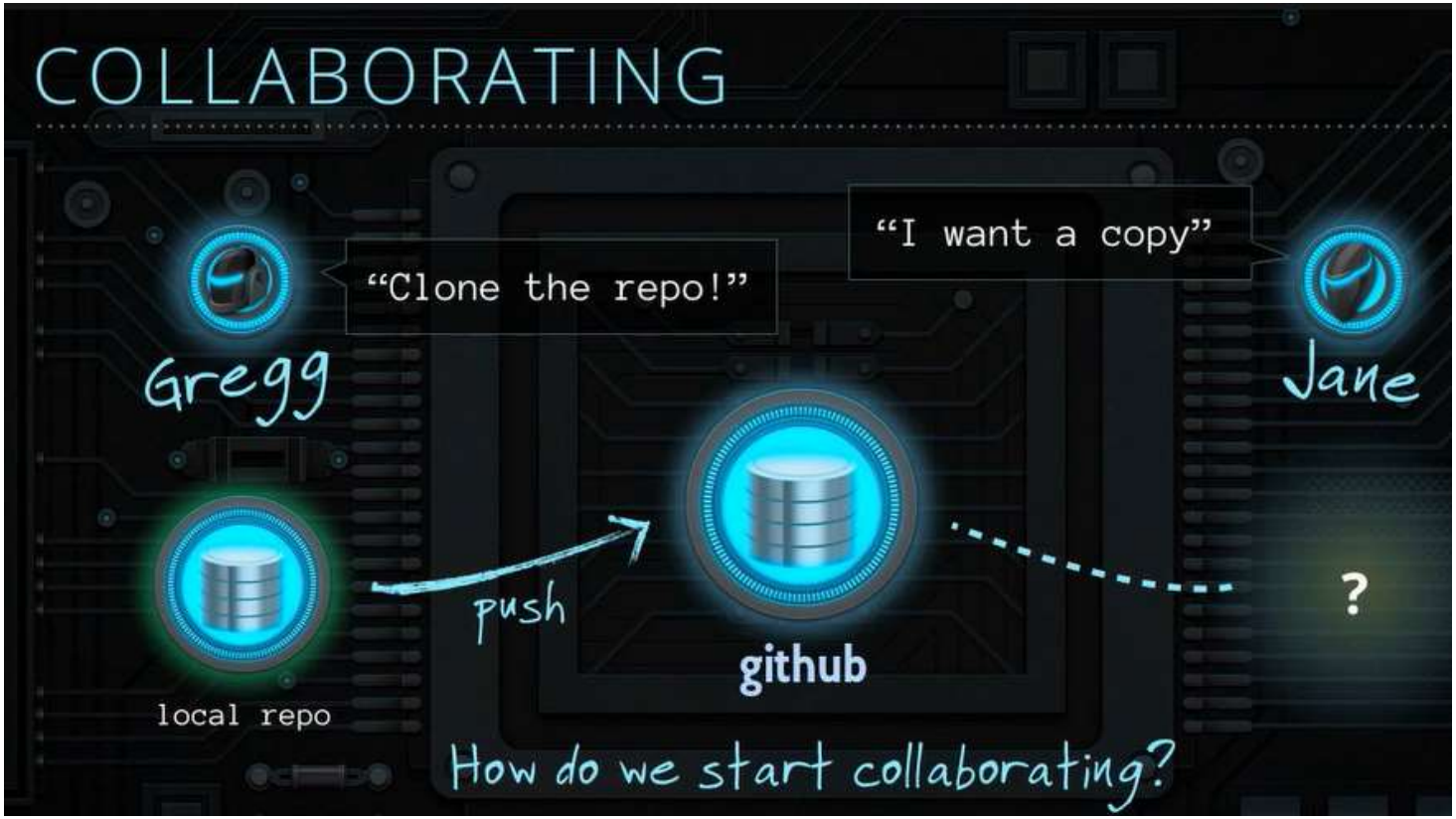


- Staging and Remote

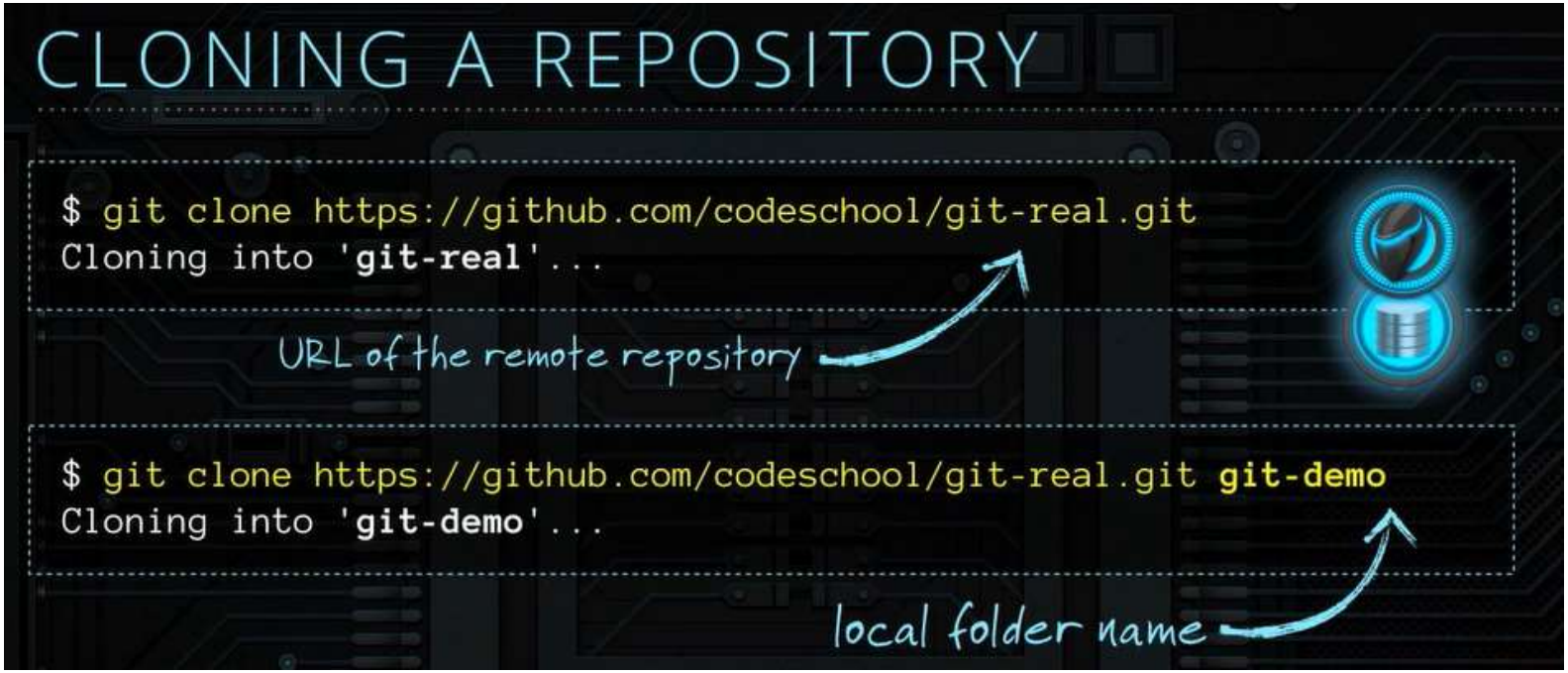
-u option in the git push command above , is if you run git push command again we don't have to specify the name of the branch , just run "git push"

- Staging and Remotes

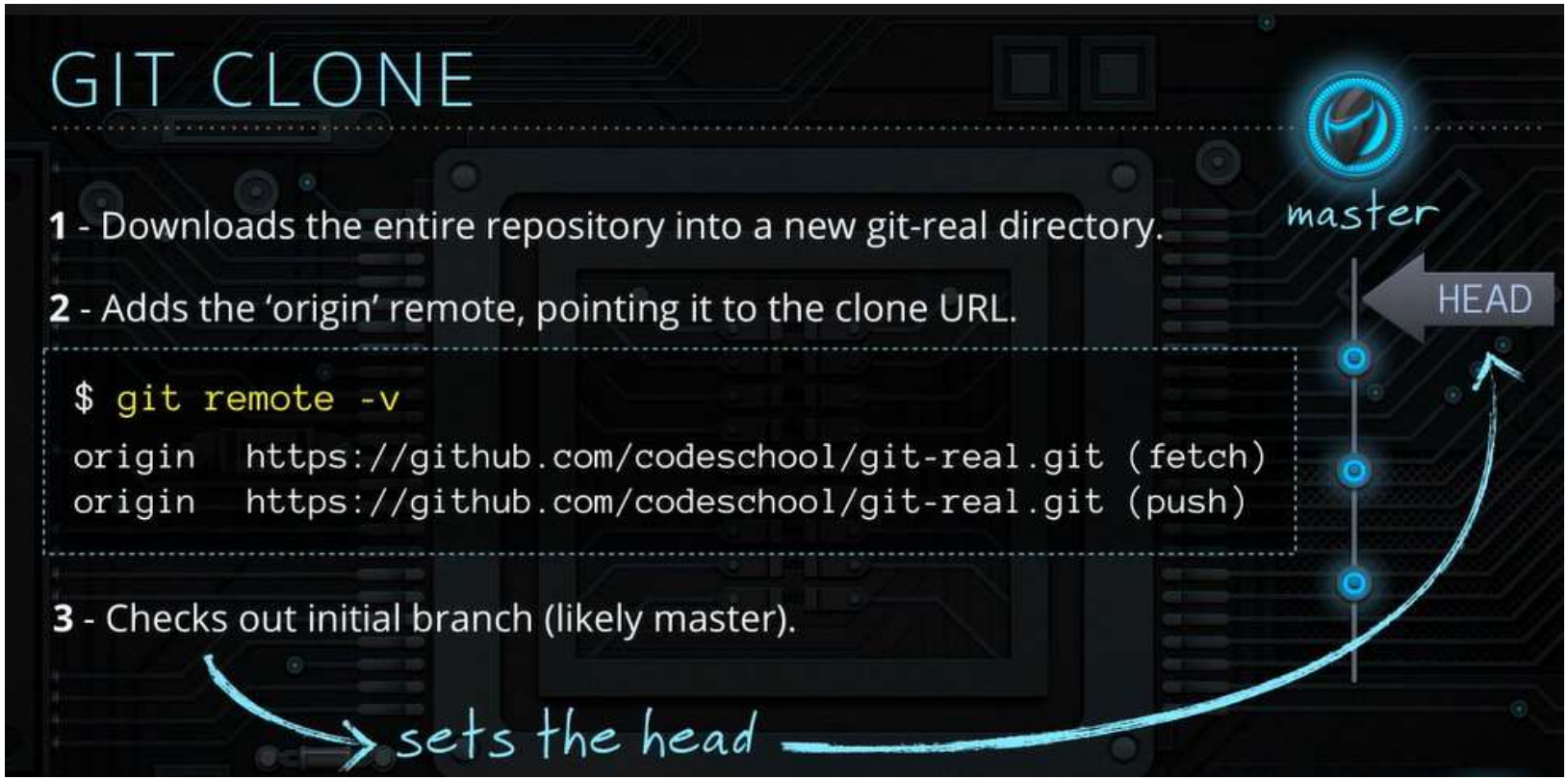
- Cloning and Branching



- Cloning and Branching

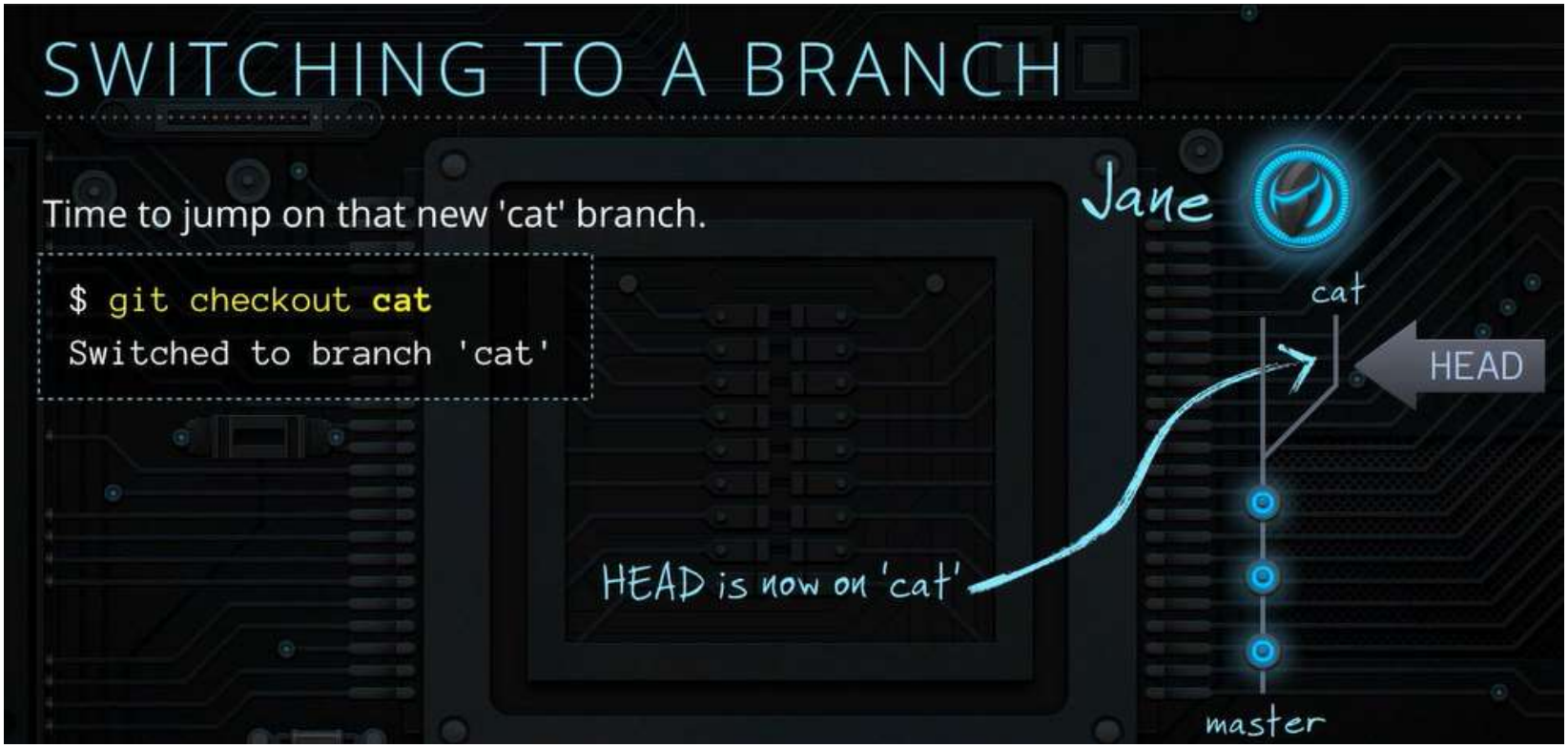


- Cloning and Branching
- What git clone does?

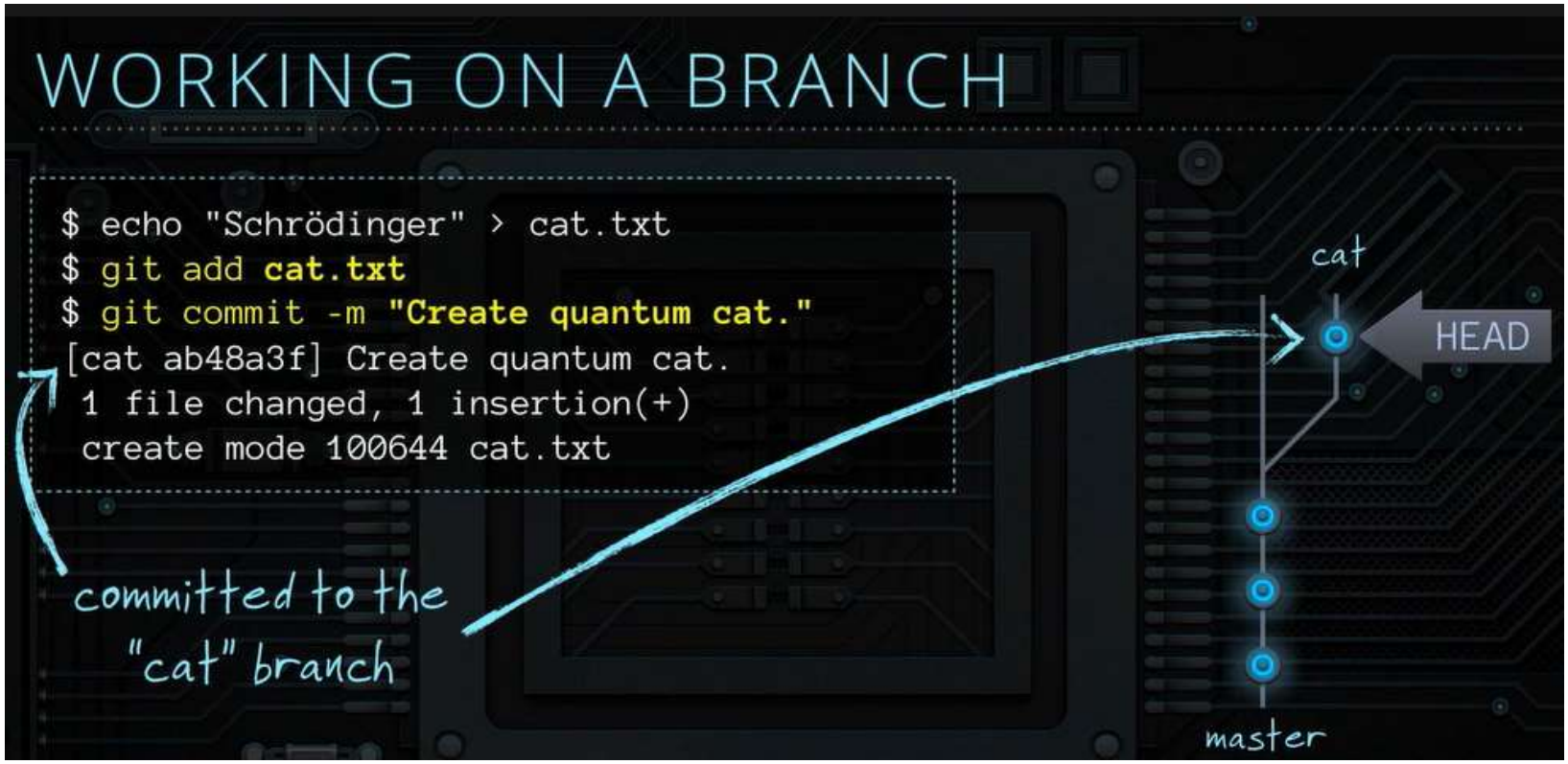


- Cloning and Branching
- After cloning the remote repo locally now f I want to create a feature branch locally then

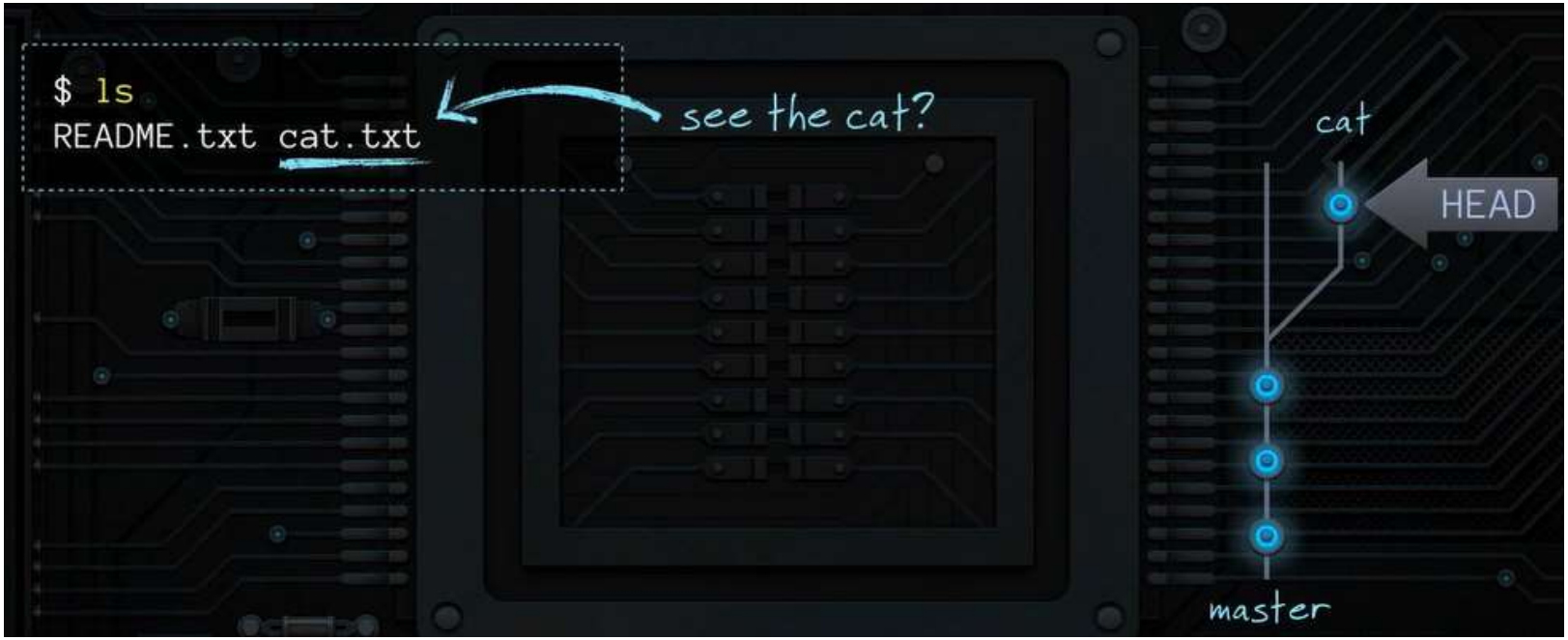
- Cloning and Branching
- To switch to the feature branch



- Cloning and Branching
- Working on the feature branch



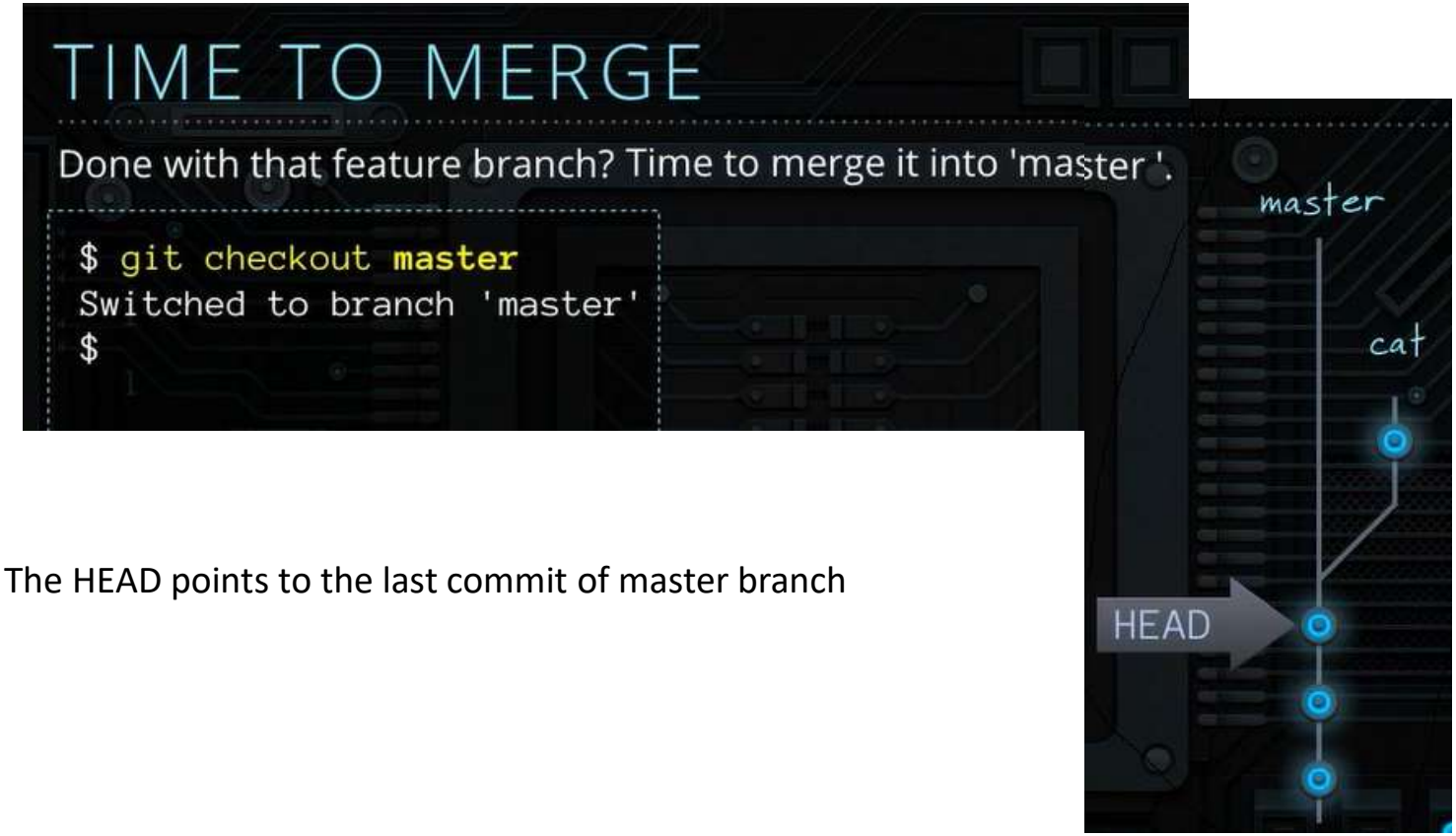
- Cloning and Branching
- Listing the contents of the feature branch



- 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

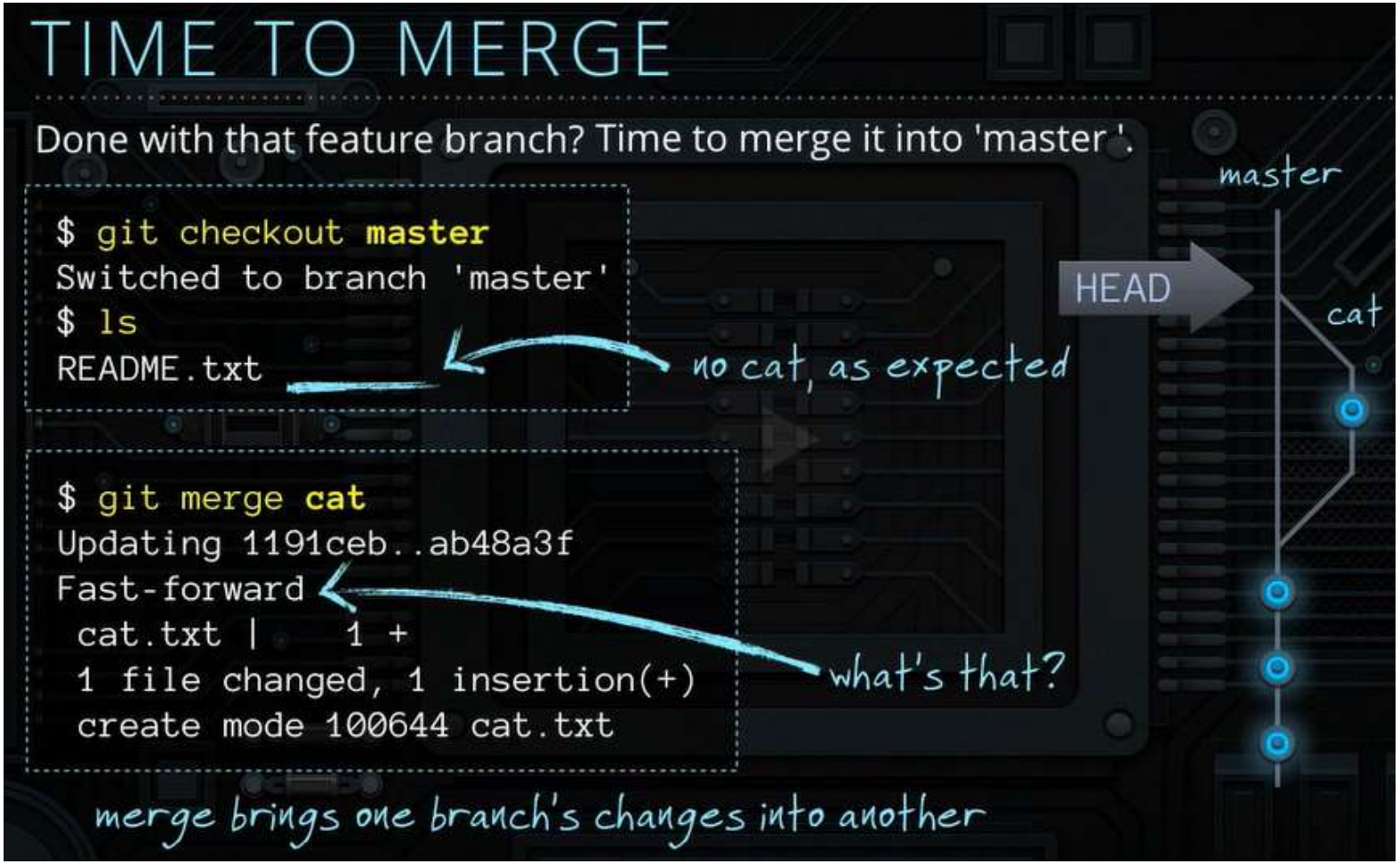
- Cloning & Branching

- Cloning & Branching



The HEAD points to the last commit of master branch

- Cloning & Branching

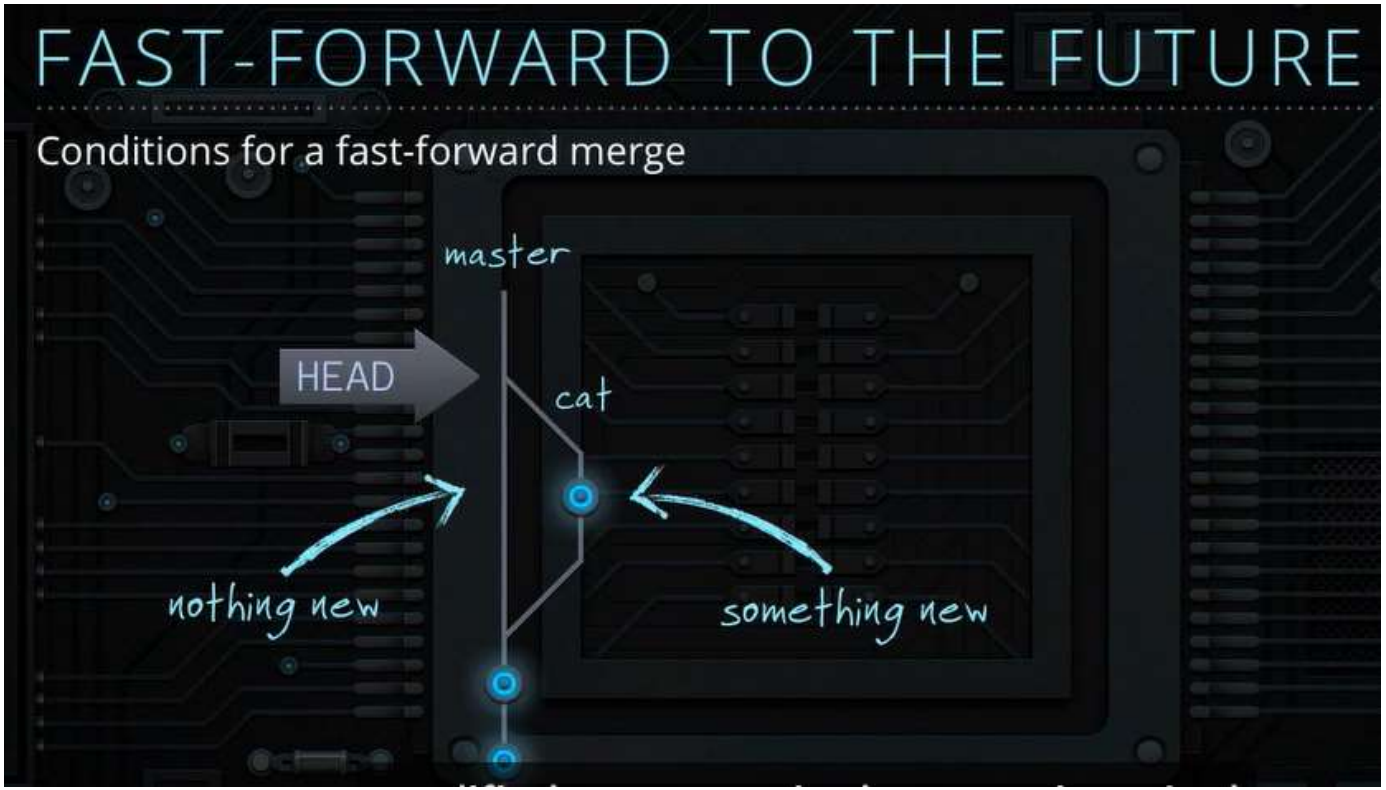


- Cloning & Branching
- In the previous slide you see when you fire the command “git merge cat”

You are merging the cat branch with the master branch.

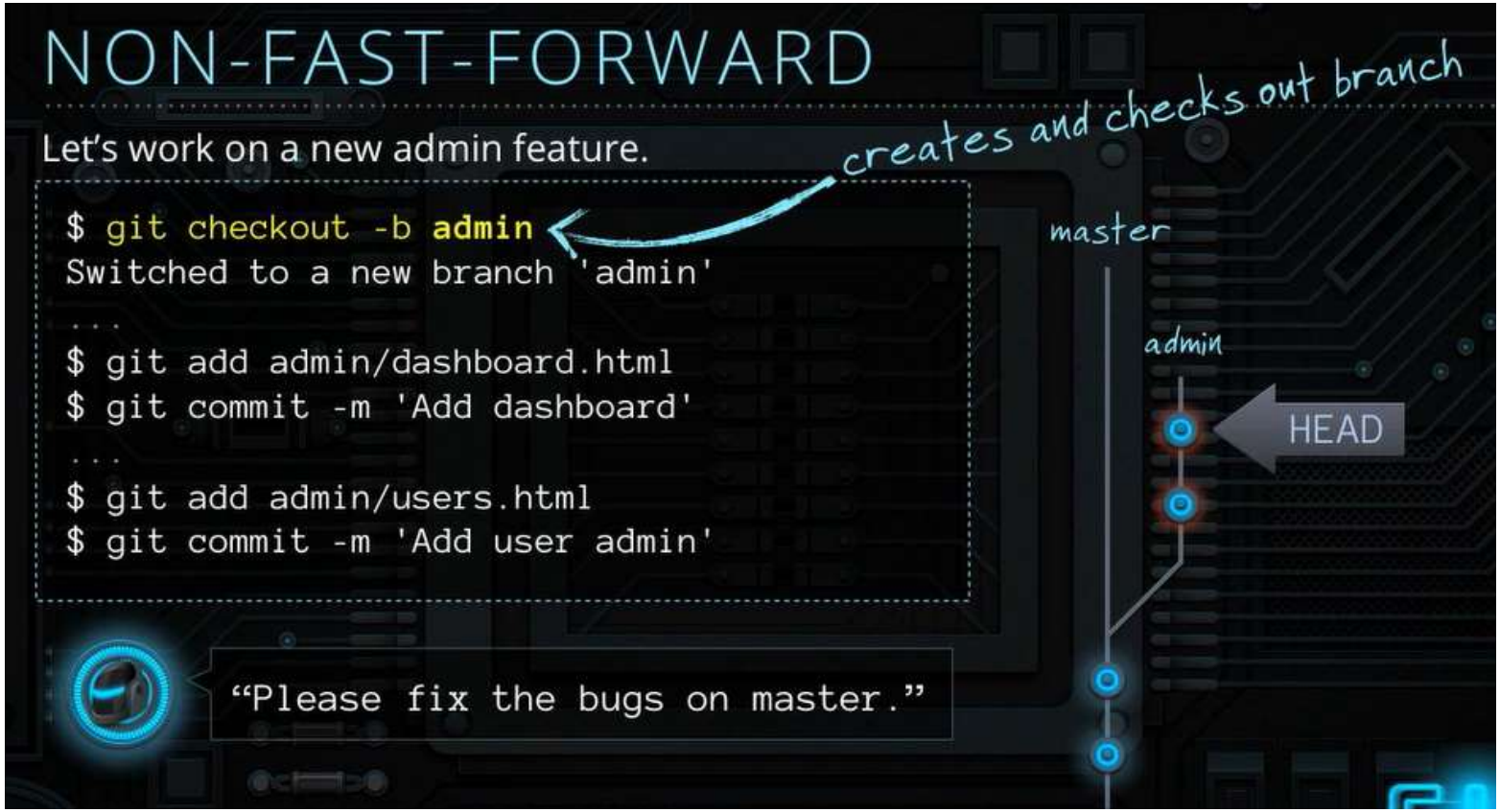
- After merging it says that it is Fast-forward.
- What does this Fast-forward mean here?
- So when we create a branch and make one commit or several commits, and do nothing on the other branch it is very easy for git to merge the cat branch to master because nothing was modified in the master in the meantime.

- Cloning & Branching



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

- Cloning & Branching



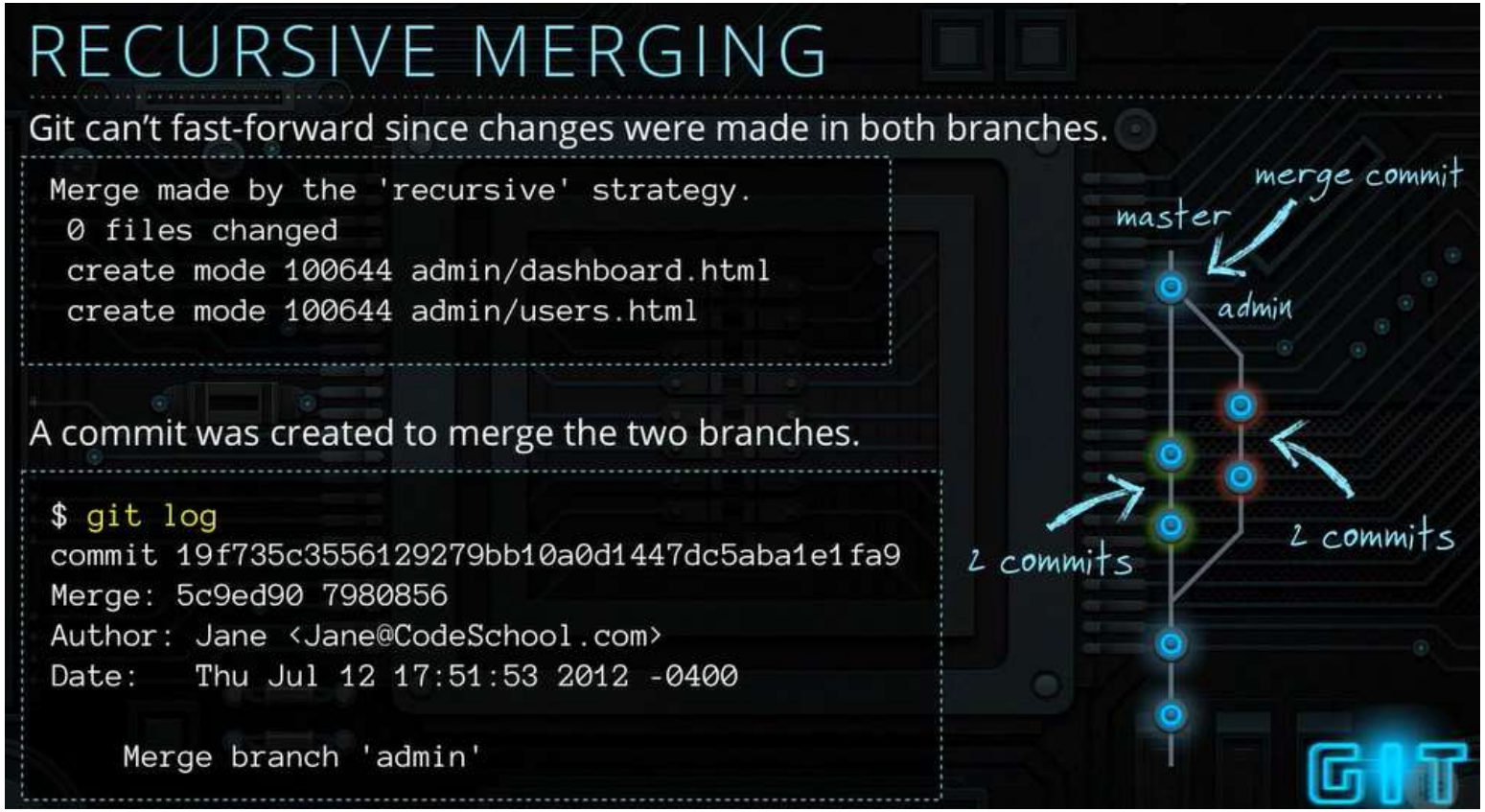
- Cloning & Branching
- While you are in the admin branch you have a message from your team member that there is a bug in the master branch and you need to fix it.

- 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.

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

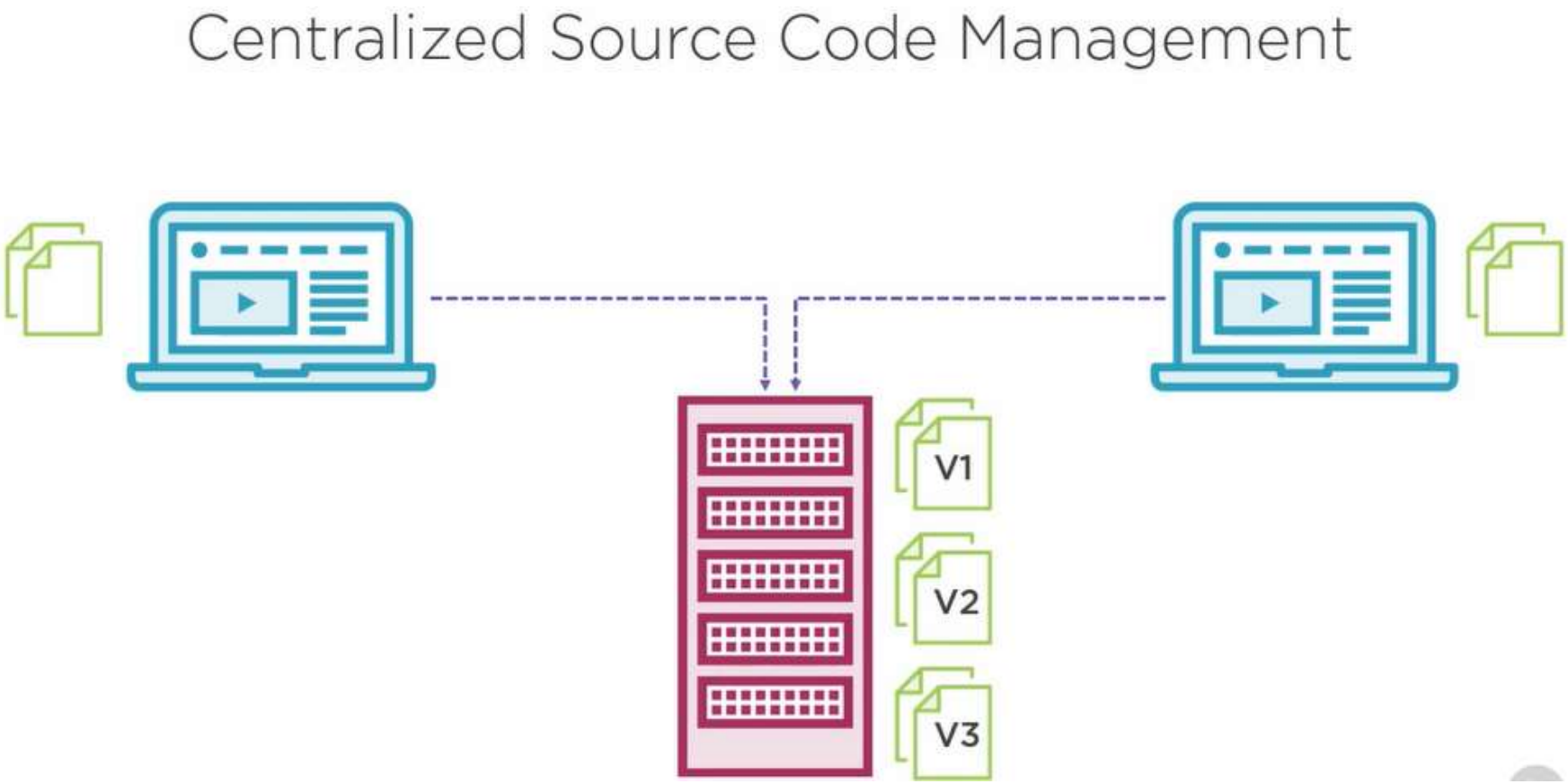
- Cloning & Branching



- 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.

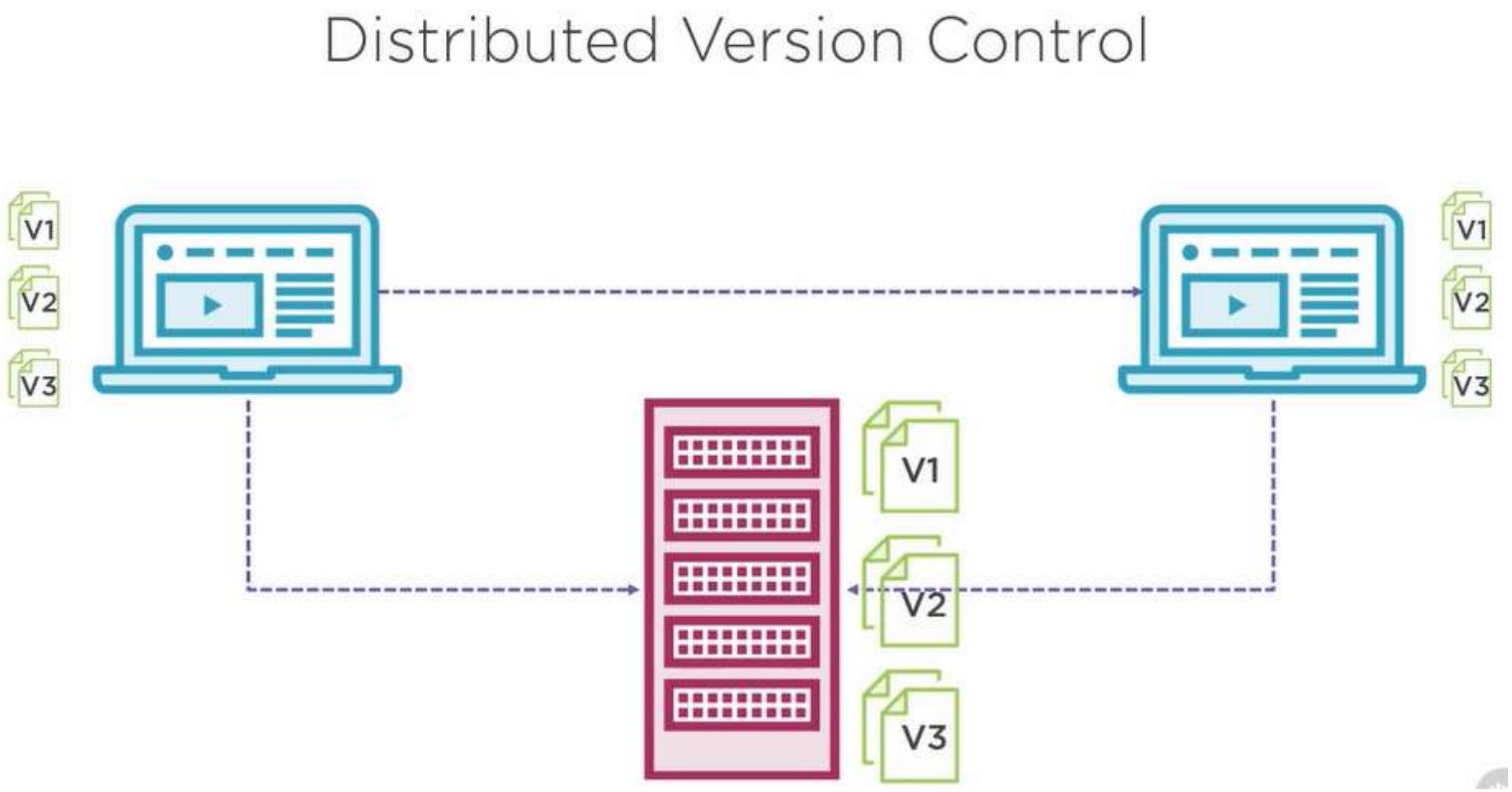
- Foundations of Git
- What is actually distributed source code management.
- In Centralized source code management there is a centralized server.
- On this Centralized server we have a copy of the source centrally on the server.
- Developers will save the copy of the code on this centralized server.
- 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



- 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



- 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>
```

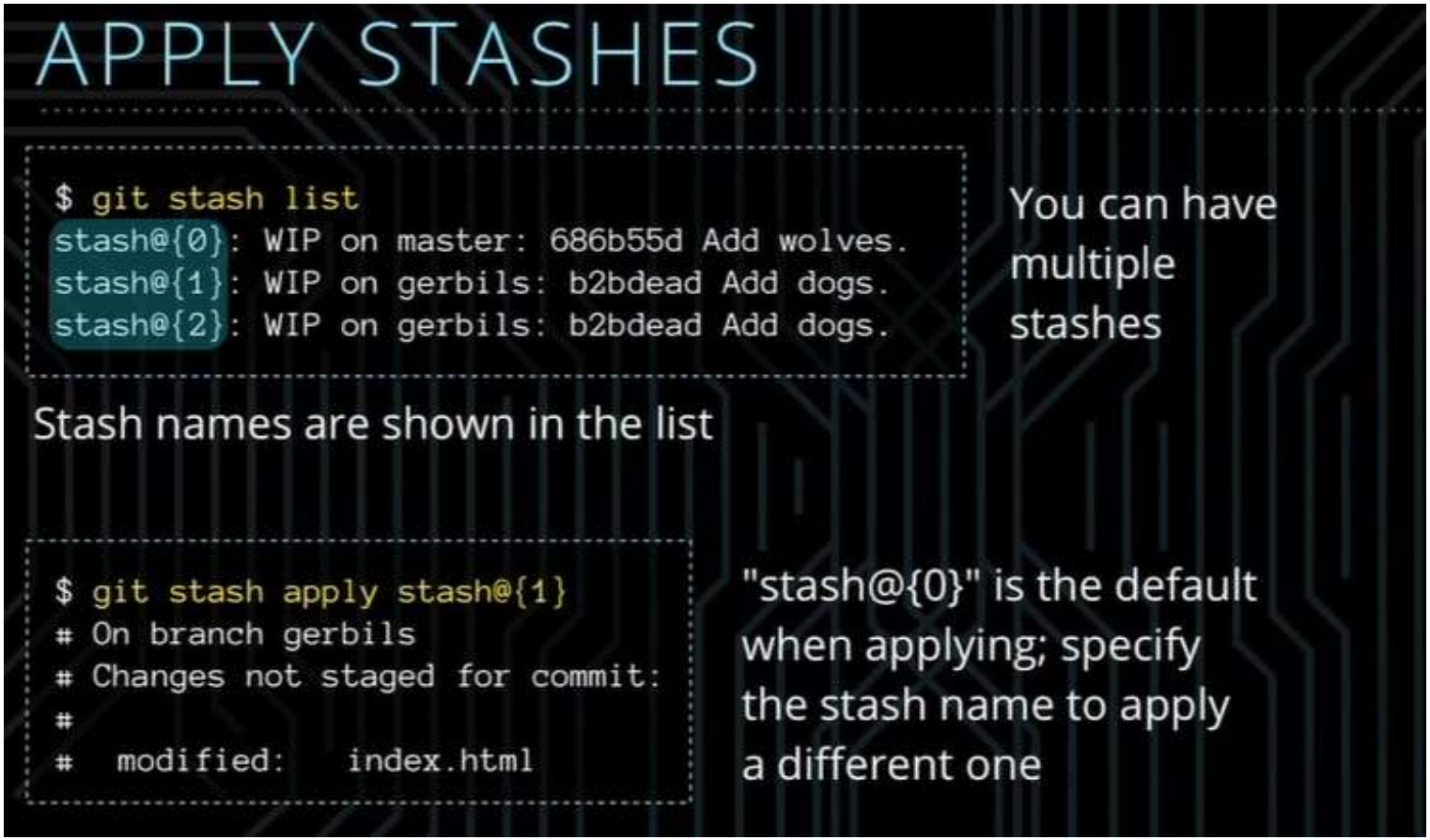
- Stashing
- So in our case, if we run **'git stash save**, ' it's going to take those files that haven't been completed yet and it's going to save them away in a temporary area.
- It's also going to restore the state from the last commit.
- So, in this case, if the command 'git diff, ‘ is executed there's 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


$ git stash save
Saved working directory and index state
WIP on gerbils: b2bdead Add dogs.
saves modified files
restores last commit
$ git diff
$ git status
# On branch gerbils
nothing to commit (working directory clean)
Modifications are "stashed" away

- 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
Old stash is gone!

- Stashing

SHORTCUTS

shortcut	same as
`git stash`	`git stash save`
`git stash apply`	`git stash apply stash@{0}`
`git stash drop`	`git stash drop stash@{0}`
`git stash pop`	`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.