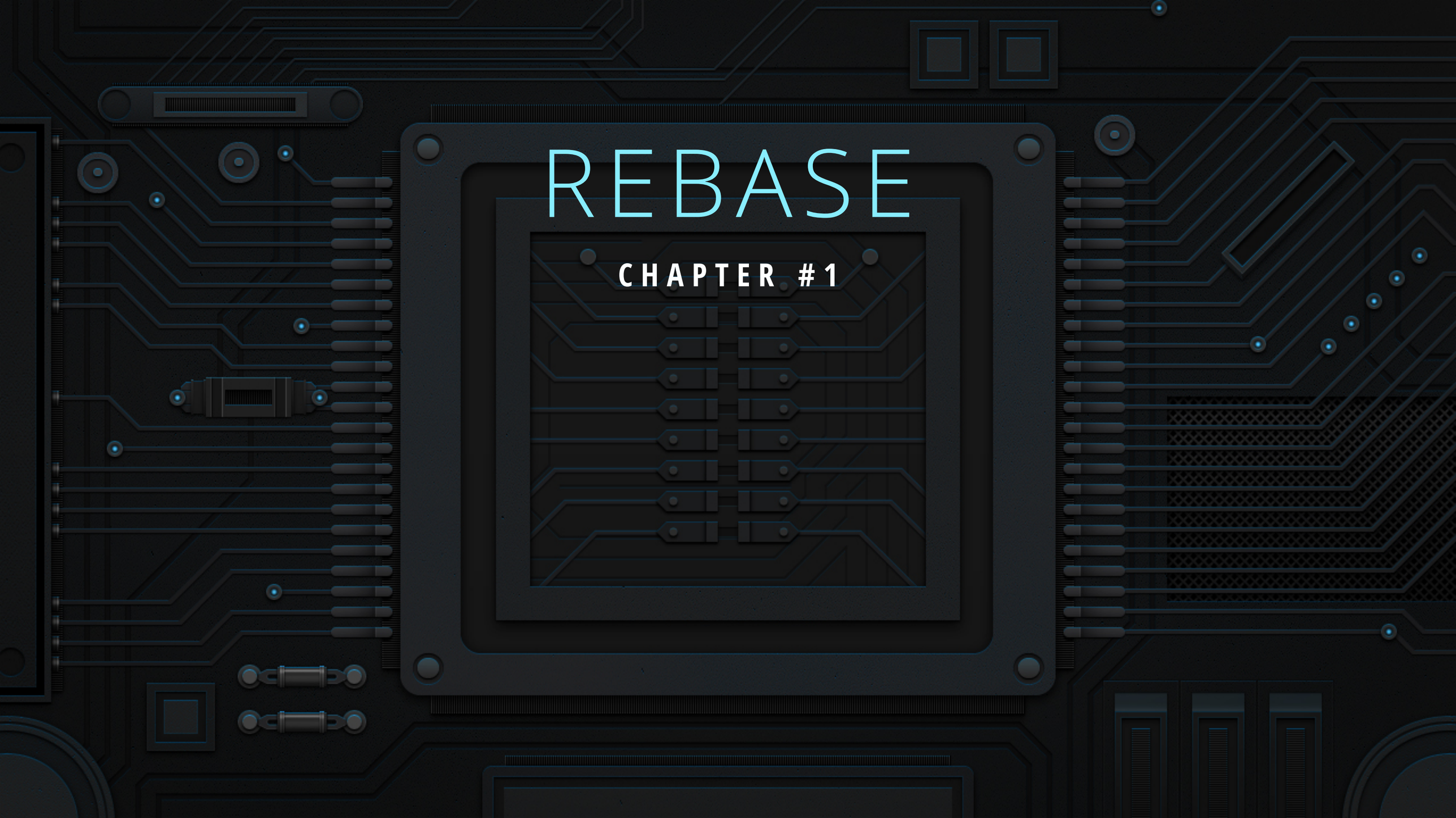

GIT REAL
2


REBASE
CHAPTER #1

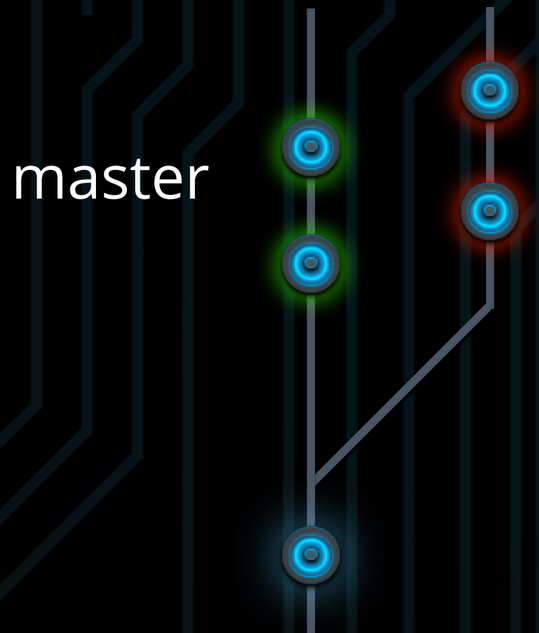
REBASE RECAP

"master" branch

```
$ git log --oneline
cd853a0 Fix pluralization.
b37fb82 Add index.
c60de92 Catalog pages
```

"unicorns" branch

```
$ git log --oneline
b36b0b1 Add rainbows.
423b4fb Add unicorns.
c60de92 Catalog pages
```



REBASE RECAP

```
$ git checkout unicorns
Switched to branch 'unicorns'
$ git rebase master
First, rewinding head to replay your work on top of it...
```

"master" branch

```
cd853a0 Fix pluralization.
b37fb82 Add index.
c60de92 Catalog pages
```

"unicorns" branch

```
b36b0b1 Add rainbows.
423b4fb Add unicorns.
c60de92 Catalog pages
```

temp

REBASE RECAP

```
$ git checkout unicorns
Switched to branch 'unicorns'
$ git rebase master
First, rewinding head to replay your work on top of it...
```

"master" branch

```
cd853a0 Fix pluralization.
b37fb82 Add index.
c60de92 Catalog pages
```

"unicorns" branch

```
c60de92 Catalog pages
```

temp

```
Add rainbows.
Add unicorns.
```

Work that's on "unicorns", but not "master", is moved to a temp area

REBASE RECAP

```
$ git checkout unicorns
Switched to branch 'unicorns'
$ git rebase master
First, rewinding head to replay your work on top of it...
```

"master" branch

```
cd853a0 Fix pluralization.
b37fb82 Add index.
c60de92 Catalog pages
```

"unicorns" HEAD moved to match "master" HEAD

"unicorns" branch

```
cd853a0 Fix pluralization.
b37fb82 Add index.
c60de92 Catalog pages
```

temp

```
Add rainbows.
Add unicorns.
```

REBASE RECAP

```
$ git checkout unicorns
Switched to branch 'unicorns'
$ git rebase master
First, rewinding head to replay your work on top of it...
```

"master" branch

```
cd853a0 Fix pluralization.
b37fb82 Add index.
c60de92 Catalog pages
```

"unicorns" branch

```
cd853a0 Fix pluralization.
b37fb82 Add index.
c60de92 Catalog pages
```

temp

```
Add rainbows.
Add unicorns.
```

Commits replayed,
one at a time

REBASE RECAP

```
$ git checkout unicorns
Switched to branch 'unicorns'
$ git rebase master
First, rewinding head to replay your work on top of it...
```

"master" branch

```
cd853a0 Fix pluralization.
b37fb82 Add index.
c60de92 Catalog pages
```

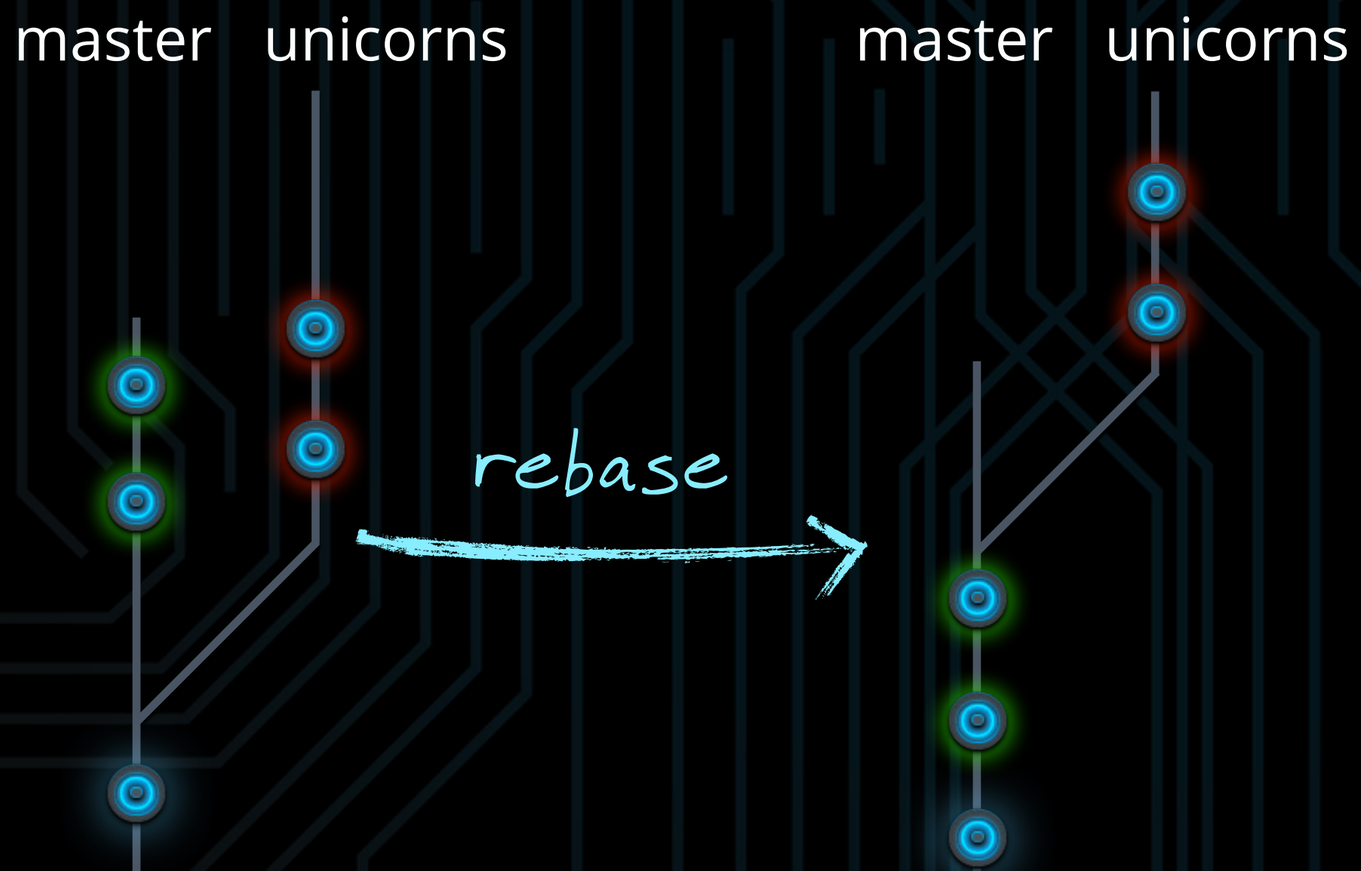
"unicorns" branch

```
30fadd8 Add rainbows.
f1616a7 Add unicorns.
cd853a0 Fix pluralization.
b37fb82 Add index.
c60de92 Catalog pages
```

temp

REBASE RECAP

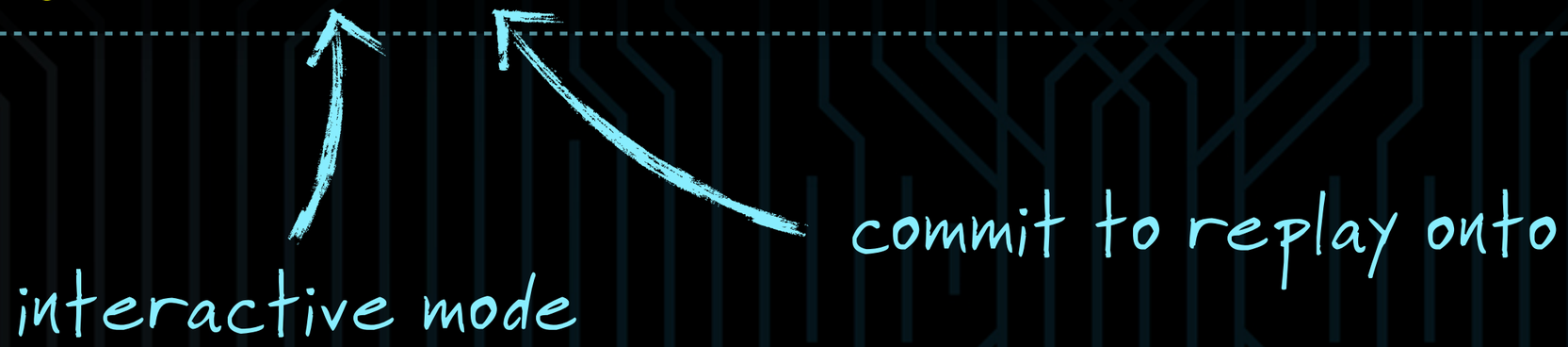
```
$ git checkout unicorns
Switched to branch 'unicorns'
$ git rebase master
First, rewinding head to replay your work on top of it...
```



INTERACTIVE REBASE

What if we need to alter commits in the SAME branch?

```
$ git rebase -i HEAD~3
```



Pops up an editor with the rebase script

INTERACTIVE REBASE

```
$ git rebase -i HEAD~3
```

editor

```
pick 9629e9b Add capybaras to index.
pick 21e37b1 Add capybaras page.
pick eb7d5a0 Actually, the plural is 'capybara'.

# Rebase 4202447..eb7d5a0 onto 4202447
#
# 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
```

these commands will execute when you close the editor

you can replace them with any of these

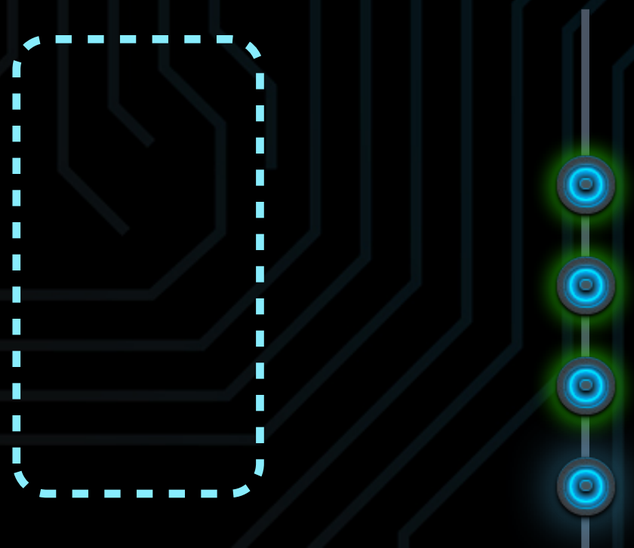
INTERACTIVE REBASE

editor

```
pick 9629e9b Add capybaras to index.
pick 21e37b1 Add capybaras page.
pick eb7d5a0 Actually, the plural is 'capybara'.
```

We didn't make any changes...

temp



The default script just replays the same commits

INTERACTIVE REBASE

Alters every commit AFTER the one you specify

```
$ git rebase -i HEAD
```

editor

```
noop

# Rebase 21e37b1..21e37b1 onto 21e37b1
```

There's no commits after HEAD!

Does nothing

```
$ git rebase -i HEAD^
```

editor

```
pick 21e37b1 Add capybaras page.

# Rebase 9629e9b..21e37b1 onto 9629e9b
```

Did you mean the parent of HEAD?

REORDER COMMITS

```
$ git log --oneline
9afe987 Actually, the plural is 'capybara'.
1ee9572 Add capybaras page.
74e6f3e Add capybaras to index.
```

```
$ git rebase -i HEAD~3
```

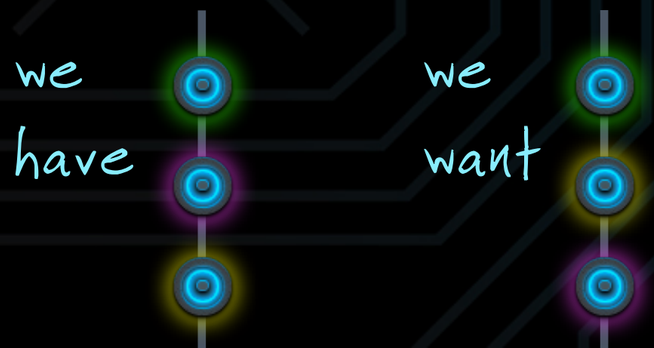
editor

```
pick 74e6f3e Add capybaras to index.
pick 1ee9572 Add capybaras page.
pick 9afe987 Actually, the plural is 'capybara'.
```

We want to re-order these two commits

"pick" chooses a commit for replay

Default script is in the original chronological order



REORDER COMMITS

```
$ git log --oneline
9afe987 Actually, the plural is 'capybara'.
1ee9572 Add capybaras page.
74e6f3e Add capybaras to index.
```

```
$ git rebase -i HEAD~3
```

editor

```
pick 1ee9572 Add capybaras page.
pick 74e6f3e Add capybaras to index.
pick 9afe987 Actually, the plural is 'capybara'.
```

Swap the order, save, and close the editor

```
$ git log --oneline
0511ab7 Actually, the plural is 'capybara'.
7d2edea Add capybaras to index.
44d59fa Add capybaras page.
```

Commits get replayed in new order

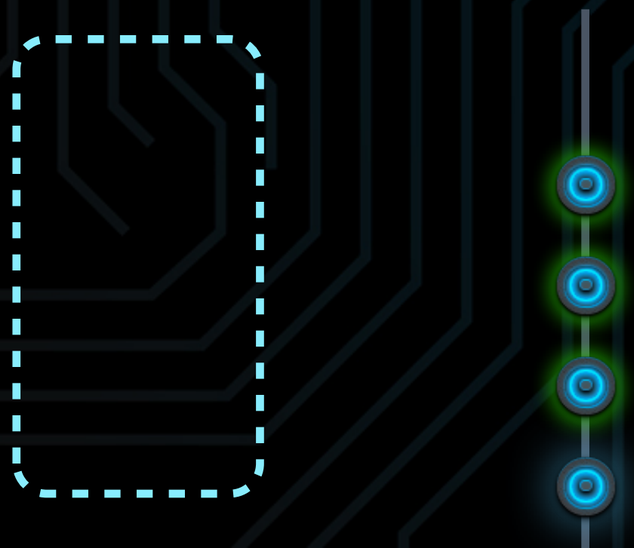
INTERACTIVE REBASE

editor

```
pick 1ee9572 Add capybaras page.
pick 74e6f3e Add capybaras to index.
pick 9afe987 Actually, the plural is 'capybara'.
```

With the order swapped...

temp



Commits will be replayed
(picked) in the specified order

HOW TO CHANGE MESSAGES

editor

```
pick 1ee9572 Add capybaras page.
pick 74e6f3e Add capybaras to index.
pick 9afe987 Actually, the plural is 'capybara'.
```

CHANGE MESSAGES

editor

```
pick 1ee9572 Add capybaras page.
pick 74e6f3e Add capybaras to index.
reword 9afe987 Actually, the plural is 'capybara'.
```

Change "pick" to "reword", then save and exit

Another editor pops up, with the existing message as the default

editor

```
Actually, the plural is 'capybara'.

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# Not currently on any branch.
# Changes to be committed:
#   (use "git reset HEAD^1 <file>..." to unstage)
#
#   modified:   capybara.html
#   modified:   index.html
```

CHANGE MESSAGES

editor

```
pick 1ee9572 Add capybaras page.
pick 74e6f3e Add capybaras to index.
reword 9afe987 Actually, the plural is 'capybara'.
```

editor

```
Change plural on index and details pages to 'capybara'.

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# Not currently on any branch.
# Changes to be committed:
#   (use "git reset HEAD^1 <file>..." to unstage)
#
#   modified:   capybara.html
#   modified:   index.html
```

Enter new message, save, and exit

NOW THE MESSAGE IS UPDATED

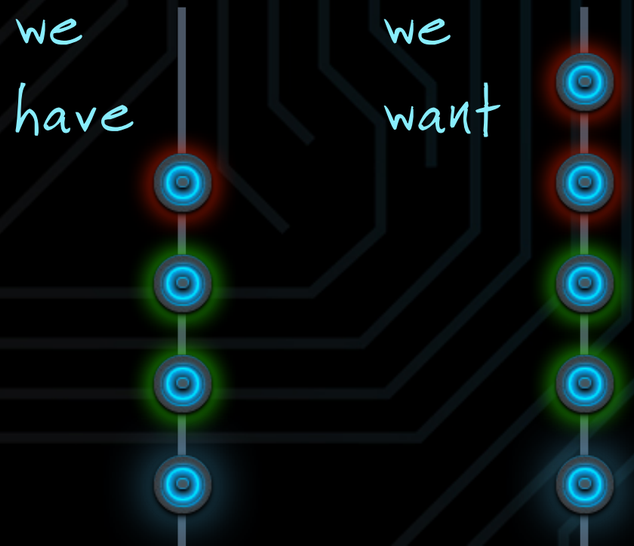
```
$ git log --oneline
9afe987 Change plural on index and details pages to 'capybara'.
1ee9572 Add capybaras page.
74e6f3e Add capybaras to index.
```

SPLIT COMMITS

editor

```
pick 1ee9572 Add capybaras page.
pick 74e6f3e Add capybaras to index.
pick 39b23ce Change plural on index and details pages to 'capybara'.
```

We want to split this into 2 commits



SPLIT COMMITS

editor

```
pick 1ee9572 Add capybaras page.
pick 74e6f3e Add capybaras to index.
edit 39b23ce Change plural on index and details pages to 'capybara'.
```

Change "pick" to"edit"

"edit" replays changes and stops after this commit

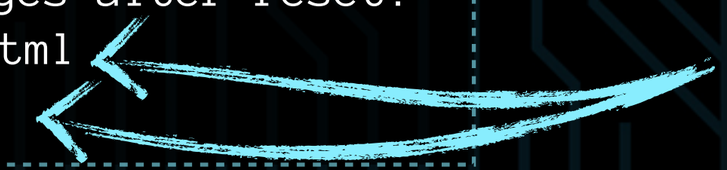
```
Stopped at 39b23ce... Change plural on index
and details pages to 'capybara'.
You can amend the commit now, with

    git commit --amend

Once you are satisfied with your changes, run

    git rebase --continue
$
```

Returns to prompt

SPLIT COMMITS

To split commits, we need to undo the changes that were just replayed

```
$ git reset HEAD^
Unstaged changes after reset:
M   capybara.html
M   index.html
```



since we don't specify "--hard",
files stay in working directory

```
$ git add capybara.html
$ git commit -m "Change plural on detail page to 'capybara'."
[detached HEAD 6e8e5d6] Change plural on detail page to 'capybara'.
 1 file changed, 1 insertion(+), 1 deletion(-)
$ git add index.html
$ git commit -m "Change plural on index page to 'capybara'."
[detached HEAD e8005f4] Change plural on index page to 'capybara'.
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Commit them separately

SPLIT COMMITS

New commits are in place, now we need to finish the rebase

```
$ git rebase --continue
Successfully rebased and updated refs/heads/master.
```

Resumes replaying
changes from temp area

```
$ git log --oneline
e8005f4 Change plural on index page to 'capybara'.
6e8e5d6 Change plural on detail page to 'capybara'.
7d2edea Add capybaras to index.
44d59fa Add capybaras page.
```

One commit
is now two!

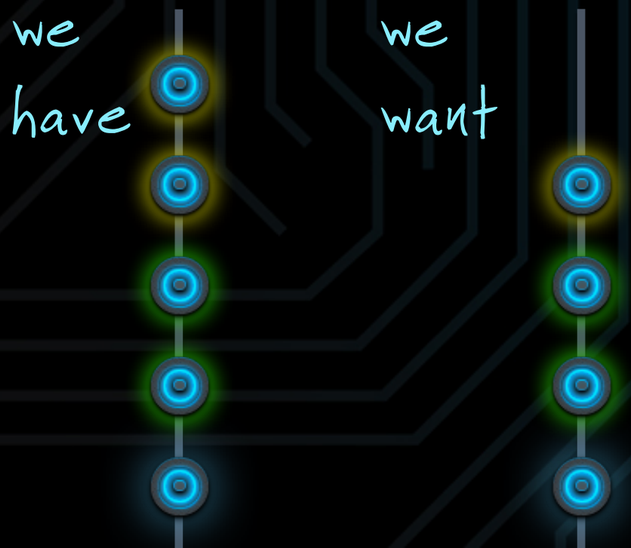
SQUASH COMMITS

No, wait... they were better off as one commit.

```
$ git rebase -i HEAD~4
```

editor

```
pick 44d59fa Add capybaras page.
pick 7d2edea Add capybaras to index.
pick 6e8e5d6 Change plural on detail page to 'capybara'.
pick e8005f4 Change plural on index page to 'capybara'.
```



SQUASH COMMITS

No, wait... they were better off as one commit.

```
$ git rebase -i HEAD~4
```

editor

```
pick 44d59fa Add capybaras page.
pick 7d2edea Add capybaras to index.
pick 6e8e5d6 Change plural on detail page to 'capybara'.
squash e8005f4 Change plural on index page to 'capybara'.
```

"squash" merges a commit with the previous commit

SQUASH COMMITS

editor

```
# This is a combination of 2 commits.
# The first commit's message is:

Change plural on detail page to 'capybara'.

# This is the 2nd commit message:

Change plural on index page to 'capybara'.

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
```

Another editor pops up, with the commits being squashed

"master" branch

SQUASH COMMITS

editor

```
# This is a combination of 2 commits.
# The first commit's message is:

Change plurals to 'capybara'.

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
```

Enter the message for the new commit, save, and exit

SQUASH COMMITS

```
$ git log --oneline
1c4614d Change plurals to 'capybara'.
7d2edea Add capybaras to index.
44d59fa Add capybaras page.
```

One commit, with the changes to both files

temp

The content of two commits is replayed, but only one commit is made


GIT REAL
2

STASHING

CHAPTER 2

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

```
$ 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

Gregg checks out "master", pulls changes, and makes fixes

```
$ git checkout master
Switched to branch 'master'
Your branch is behind 'origin/master' by 1 commit,
and can be fast-forwarded.

$ git pull
Updating b2bdead..686b55d
Fast-forward
 wolf.html | 7 +++++++
 1 file changed, 7 insertions(+)
 create mode 100755 wolf.html
```


LIST STASHES

Now he's ready to return to the feature branch

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


APPLY STASHES

```
$ git stash apply
# On branch gerbils
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes)
#
#  modified:   index.html
```

Bring stashed files back

```
$ git diff
diff --git a/index.html b/index.html
       <ul>
+        <li><a href="gerbil
         <li><a href="cat.html">Cats</a></li>
         <li><a href="dog.html">Dogs</a></li>
       </ul>
```

Here are the changes!

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

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!

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`

STASH CONFLICTS

```
$ git stash apply
error: Your local changes to the following files would be
overwritten by merge:
    index.html
Please, commit your changes or stash them before you can merge.
Aborting
```

Conflicts are possible when applying a stash

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

Commit or reset your local changes, as appropriate

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

success!

STASH CONFLICTS

```
$ git stash apply
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
```

You need to merge the conflicted lines as usual...

STASH CONFLICTS

```
$ git stash pop
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
```

If you're using the "pop" shortcut, and there's a conflict...

You *also* need to merge the conflicted lines as usual...

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

And the stash won't be dropped automatically

```
$ git stash drop
Dropped refs/stash@{0} (e4ba8a4...)
```

So be sure to do it manually

KEEP INDEX

Jane wants to commit some changes saved in the staging area

```
$ git status
# On branch gerbils
# Changes to be committed:
#
#   new file:   gerbil.html
#
# Changes not staged for commit:
#
#   modified:   index.html
```

we want to commit this

and stash everything else

```
$ git stash save
Saved working directory and index state...

$ git status
# On branch gerbils
nothing to commit (working directory clean)
```

Whoops, she stashed everything

KEEP INDEX

Jane pops the stash to get her staged changes back

```
$ git stash pop
# On branch gerbils
# Changes to be committed:
#
#   new file:   gerbil.html
#
# Changes not staged for commit:
#
#   modified:   index.html
#
Dropped refs/stash@{0} (de4105a...)
```

Stashed staging areas get restored later

KEEP INDEX

```
$ git stash save --keep-index
Saved working directory and index state
HEAD is now at b2bdead Add dogs.
```

"--keep-index" option causes the staging area not to be stashed

```
$ git status
# On branch gerbils
# Changes to be committed:
#
#   new file:   gerbil.html

$ git commit -m "Add gerbils section."
[gerbils 130a661] Add gerbils section.
 1 file changed, 7 insertions(+)
 create mode 100644 gerbil.html
```

Now Jane can commit the changes

KEEP INDEX

```
$ git stash pop
# On branch gerbils
# Changes not staged for commit:
#
#   modified:   index.html
#
no changes added to commit
Dropped refs/stash@{0} (db990c0...)
```

Unstaged changes get stashed and restored as usual

INCLUDE UNTRACKED

Jane needs to stash changes, but she has an untracked file

```
$ git stash save
Saved working directory and index state
HEAD is now at b2bdead Add dogs.

$ git status
# On branch gerbils
# Untracked files:
#
#   gerbil.html
```

Normally, only tracked files get stashed

INCLUDE UNTRACKED

```
$ git stash save --include-untracked
Saved working directory and index state
HEAD is now at b2bdead Add dogs.
```

```
$ git status
# On branch gerbils
nothing to commit (working directory clean)

$ git stash pop
# On branch gerbils
# Changes not staged for commit:
#
#   modified:   index.html
#
# Untracked files:
#
#   gerbil.html
```

"--include-untracked" option causes untracked files to be stashed, too

When stash is restored, untracked files will be, too

LIST OPTIONS

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

```
$ git stash list --stat
stash@{0}: WIP on gerbils: b2bdead Add dogs.
 gerbil.html | 7 +++++++
 index.html  | 1 +
 wolf.html   | 7 -------
 3 files changed, 8 insertions(+), 7 deletions(-)

stash@{1}: WIP on master: 686b55d Add wolves.
 gerbil.html | 7 -------
 ...
```

When you have more than one stash, it's hard to tell them apart

"git stash list" can take any option "git log" can
For example, "--stat" summarizes file changes

STASH SHOW

```
$ git stash show stash@{0}
 gerbil.html | 7 +++++++
 index.html  | 1 +
 2 files changed, 8 insertions(+)
```

Shows one particular stash

```
$ git stash show
 gerbil.html | 7 +++++++
 index.html  | 1 +
 2 files changed, 8 insertions(+)
```

Like "apply" and "drop", acts on most recent stash by default

STASH SHOW

```
$ git stash show --patch
diff --git a/gerbil.html b/gerbil.html
+<!DOCTYPE html>
+<html lang="en">
+  <head>
+    <meta charset="UTF-8">
+    <title>Our Gerbils</title>
+  </head>
+</html>
diff --git a/index.html b/index.html
    <body>
      <nav>
        <ul>
+         <li><a href="gerbil
          <li><a href="cat.html">Cats</a></li>
          <li><a href="dog.html">Dogs</a></li>
        </ul>
```

Also takes any option "git log" can

For example, "--patch" shows file diffs

STASH MESSAGES

```
$ git stash save "Add gerbils page, start index."
Saved working directory and index state
HEAD is now at b2bdead Add dogs.
```

You can provide a stash message when saving

```
$ git stash list
stash@{0}: On gerbils: Add gerbils page, start index.
stash@{1}: WIP on gerbils: b2bdead Add dogs.
stash@{2}: WIP on master: 686b55d Add wolves.
stash@{3}: WIP on gerbils: b2bdead Add dogs.
```

Gets listed in place of default message

BRANCHING

Gregg has added a gerbils section and is working on the toys page, when management asks him to deploy the branch ASAP

```
$ git stash save "Start toy section."
Saved working directory and index state
HEAD is now at 5e3bde9 Add gerbils page.
```

He stashes his changes...

Gregg accidentally deletes this branch, out of habit

```
$ git branch -d gerbils
Deleted branch gerbils (was 5e3bde9).
```


BRANCHING

Now, he needs a new branch to restore the stashed toys page

new branch name

stash to pop

```
$ git stash branch gerbil-toys stash@{0}
Switched to a new branch 'gerbil-toys'
# On branch gerbil-toys
# Changes not staged for commit:
#
#   modified:   gerbil.html
#
Dropped stash@{0} (5797b65...)
```

"git stash branch" checks a new branch out automatically...

and drops the stash automatically

```
$ git commit -am "Add gerbil toys."
[gerbil-toys e7083eb] Add gerbil toys.
 1 file changed, 3 insertions(+)
```

New branch is an ordinary branch, ready for commits

CLEAR STASHES

We have a lot of stashes we no longer need

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

```
$ git stash clear
```

Clears all of them at once

```
$ git stash list
```

All gone!


GIT REAL
2

PURGING HISTORY

CHAPTER 3

PURGING HISTORY

Bob has committed a file he shouldn't have

```
$ git log --patch
commit 5a794156e9e1ef9b9f4db07363f299258b1a5a93
Author: Bob <bob@example.com>
Date:   Tue Mar 19 00:03:32 2013 -0700

    Fix security breach.

deleted file mode 100644
--- a/passwords.txt
+++ /dev/null
@@ -1 +0,0 @@
-The server password is: 'foobar'.
...
```

Even if he deletes it now, its contents will still be visible in history

BEFORE WE CONTINUE...

- Reasons *not* to rewrite history:
  - Why bother? Your data is already compromised.
  - Everyone must update their work to reflect your revised commits.
- When you should do it anyway:
  - Committed files violate someone's copyright.
  - Large binary files are making your repo too big.
  - You're rewriting commits that haven't been made public.

BEFORE WE CONTINUE...

```
$ git clone petshop petshop-filter
Cloning into 'petshop-filter'...
done.

$ cd petshop-filter
```

Bob backs up
his entire repo

You **can** lose work when
you're rewriting history!

TREE FILTER

```
$ git filter-branch --tree-filter <command> ...
```

Git will check each commit out into working directory, run your command, and re-commit

Examples:

```
--tree-filter 'rm -f passwords.txt'
```

Remove "passwords.txt" from project root

```
--tree-filter 'find . -name "*.mp4" -exec rm {} \;'
```

Remove video files from any directory

TREE FILTER

```
$ git filter-branch --tree-filter 'rm -f passwords.txt' -- --all
Rewrite f5d1142dd3e830940e9f1361e8c500df8834fd20 (4/4)
Ref 'refs/heads/treats' was rewritten
...
```

```
$ git log --patch
commit 5a794156e9e1ef9b9f4db07363f299258b1a5a93
Author: Bob <bob@example.com>
Date:   Tue Mar 19 00:03:32 2013 -0700

    Fix security breach.
```

"passwords.txt" contents are gone!

Goes through all branches, and removes "passwords.txt" from each commit

`--all` filter all commits in all branches

`HEAD` filter only current branch

COMMAND FAILS, FILTER FAILS

fails if file isn't present

```
$ git filter-branch --tree-filter 'rm passwords.txt' -- --all
Rewrite 539180de10622fd3d6786b2ad22f65bf70a476d9 (1/4)
rm: passwords.txt: No such file or directory
tree filter failed: rm passwords.txt
```

File won't be there in some commits

doesn't fail if file isn't present

```
$ git filter-branch --tree-filter 'rm -f passwords.txt' -- --all
Rewrite f5d1142dd3e830940e9f1361e8c500df8834fd20 (4/4)
Ref 'refs/heads/treats' was rewritten
```

INDEX FILTER

```
$ git filter-branch --index-filter <command> ...
```

command MUST operate on staging area

Git will run your command against each commit, but without checking it out first (so it's faster)

operates on working directory

```
--index-filter 'rm -f passwords.txt'
```

No effect!

operates on staging area

```
--index-filter 'git rm --cached --ignore-unmatch passwords.txt'
```

INDEX FILTER

As with "--tree-filter", if the command fails the filter will stop

fails if file isn't present

```
--index-filter 'git rm --cached passwords.txt'
```

succeeds even if file isn't present

```
--index-filter 'git rm --cached --ignore-unmatch passwords.txt'
```

FORCE

After you run filter-branch, Git leaves a backup of your tree in the ".git" directory

```
$ git filter-branch --tree-filter 'rm -f passwords.txt'
Cannot create a new backup.
A previous backup already exists in refs/original/
Force overwriting the backup with -f
```

By default, you can't run filter-branch again because it won't overwrite the backup

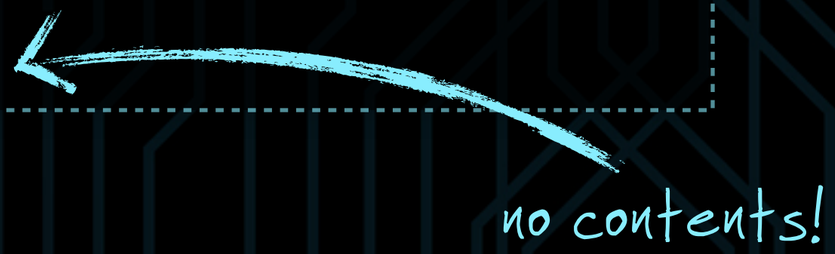
```
$ git filter-branch -f --tree-filter 'rm -f passwords.txt'
Rewrite 68880af108a5eaa19eb543f3455a1d0dcd6ff632 (4/4)
```

You can force it with the -f option

PRUNE EMPTY COMMITS

Our filters are resulting in some empty commits

```
$ git log --patch
commit 5a794156e9e1ef9b9f4db07363f299258b1a5a93
Author: Bob <bob@example.com>
Date:   Tue Mar 19 00:03:32 2013 -0700

    Fix security breach.
```



PRUNE EMPTY COMMITS

```
$ git log --oneline
68880af Add treats page.
5a79415 Fix security breach.
016b982 Update index.
539180d Initial commit.
```

Commit has no contents

```
$ git filter-branch -f --prune-empty -- --all
```

"--prune-empty" option drops commits that don't alter any files

```
$ git log --oneline
6d9a429 Add treats page.
016b982 Update index.
539180d Initial commit.
```

Empty commit is gone

can prune during filtering, too

```
git filter-branch --tree-filter 'rm -f passwords.txt' --prune-empty -- --all
```


GIT REAL
2

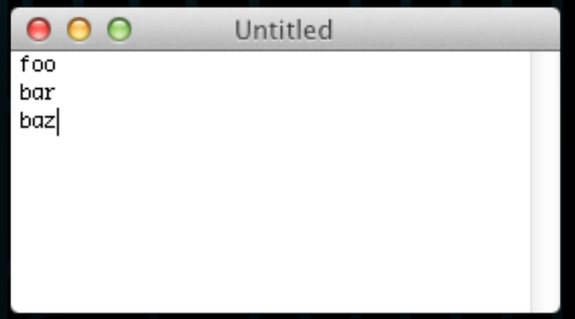
WORKING TOGETHER

CHAPTER 4

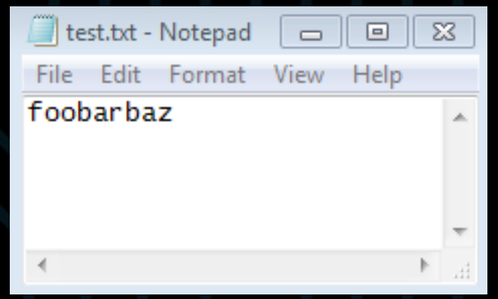
LINE ENDINGS

Different OSs store line separators differently

OS	Expects Character(s)
OSX/Linux	LF (LineFeed)
Windows	CR (Carriage Return) and LF pair



Some team members work on OSX and Linux



Some work on Windows

retrieving the wrong format means trouble!

LINE ENDINGS

Git can auto-correct formats for each OS

On Unix-like systems (Linux, OSX, etc.)

```
$ git config --global core.autocrlf input
```

Changes CR/LF to LF on **commit**

(fixes any Windows line endings that get introduced)

On Windows systems

```
> git config --global core.autocrlf true
```

Changes LF to CR/LF on **checkout**

(but converts back to LF on commit)

On Windows-**only** projects

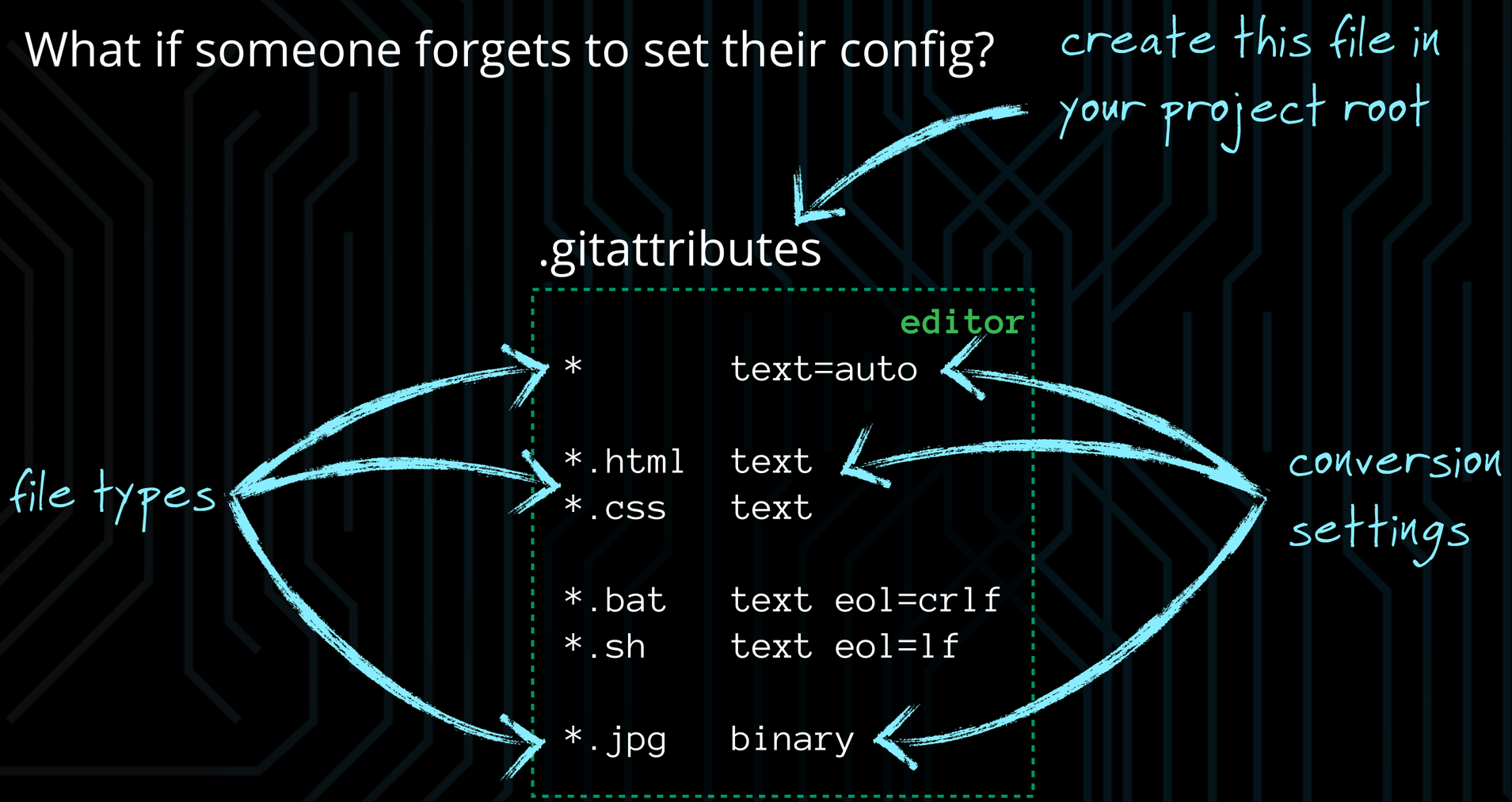
```
> git config core.autocrlf false
```

Does no conversion

(conversion isn't needed if everyone expects CR/LF)

GIT ATTRIBUTES FILE

What if someone forgets to set their config?



GIT ATTRIBUTES FILE

File type - designates which files rule applies to

`*`	Matches any type without an extension-specific rule
`*.html`	Matches all HTML files
`*.jpg`	Matches all JPEG files

GIT ATTRIBUTES FILE

Conversion settings

`text=auto`	Choose conversion automatically
`text`	Treat files as text - convert to OS's line endings on checkout, back to LF on commit
`text eol=crlf` `text eol=lf`	Convert to specified format on checkout, back to LF on commit
`binary`	Treat files as binary - do no conversion

GIT ATTRIBUTES FILE

Some typical rules

```
* text=auto
```

By default, auto-convert line endings

```
*.html text
*.css  text
```

Treat HTML and CSS files as text

(not needed if you have the line above)

```
*.jpg binary
*.png binary
```

Treat image files as binary

```
*.sh  text eol=lf
*.bat text eol=crlf
```

Keep shell scripts in Unix format,
batch files in Windows format

CHERRY-PICK

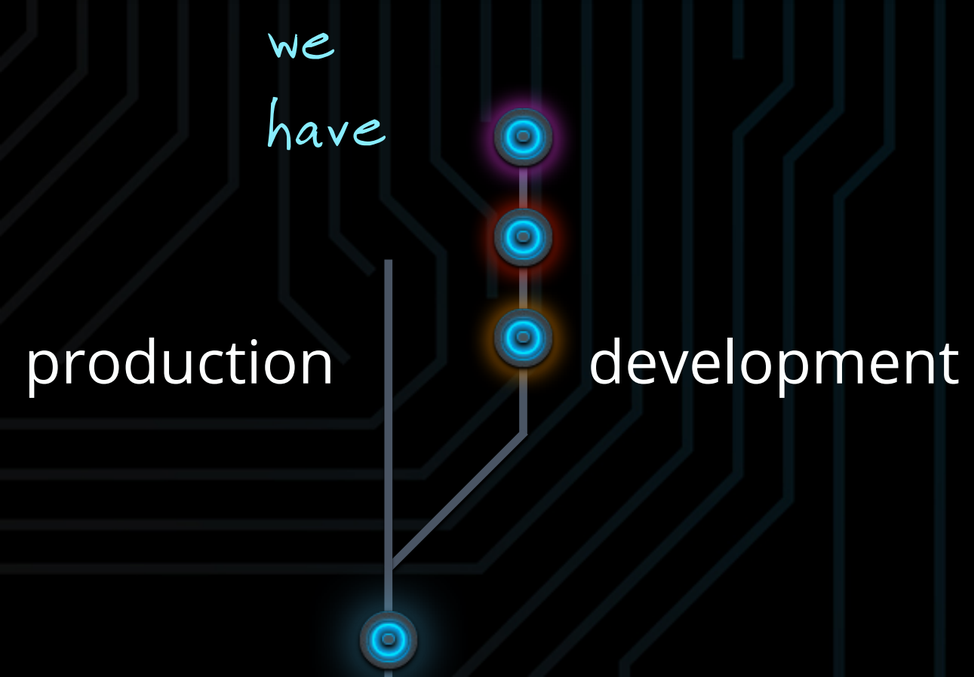
"production" branch

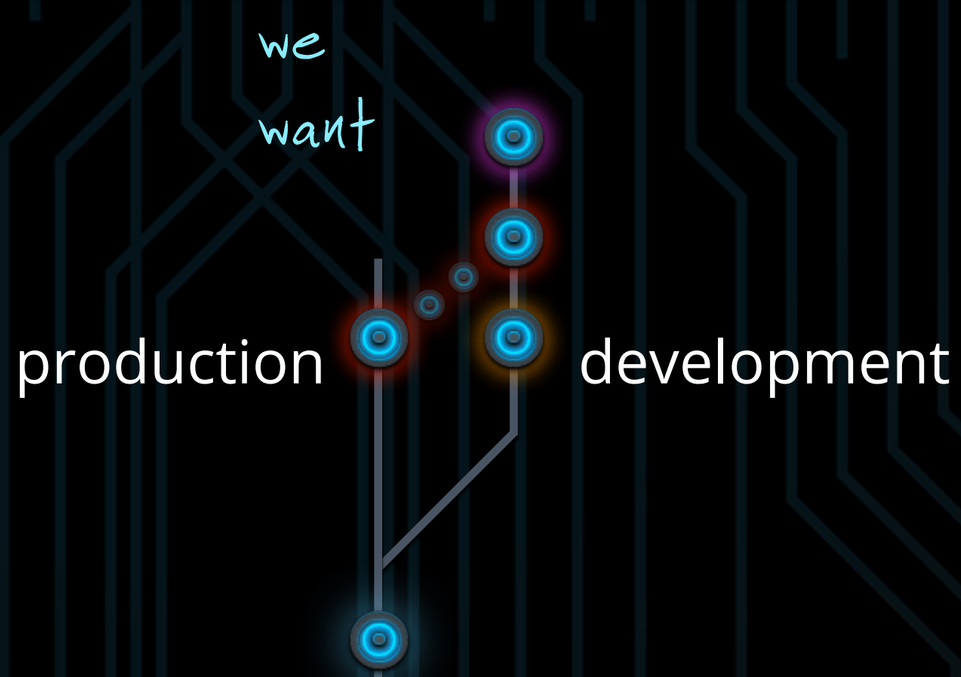
```
$ git log --oneline
3d7225f Add guppies.
```

"development" branch

```
$ git log --oneline
c400bf0 Add Cthulhu.
53212e5 Add clownfish.
55ae374 Add sharks.
3d7225f Add guppies.
```

We need **just** the clownfish page in production ASAP





CHERRY-PICK

"production" branch

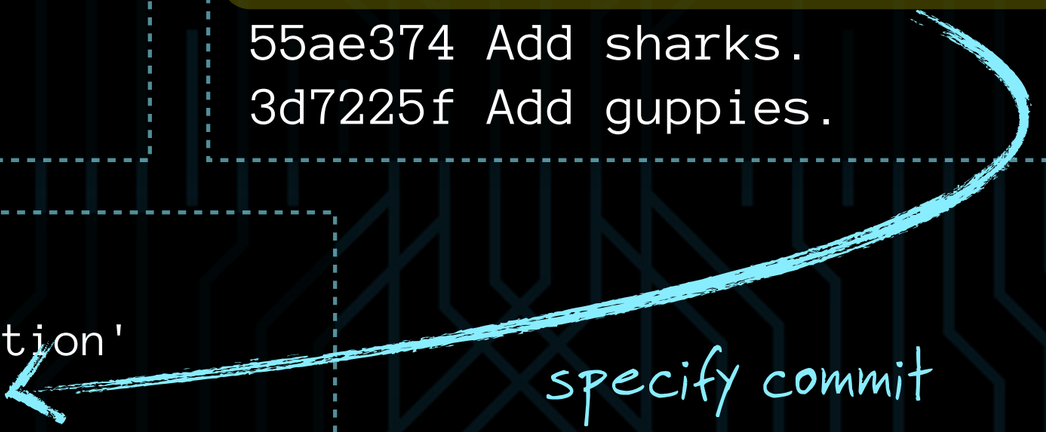
```
$ git log --oneline
3d7225f Add guppies.
```

"development" branch

```
$ git log --oneline
c400bf0 Add Cthulhu.
53212e5 Add clownfish.
55ae374 Add sharks.
3d7225f Add guppies.
```

We need **just** the clownfish page in production ASAP

```
$ git checkout production
Switched to branch 'production'
$ git cherry-pick 53212e5
[production 80a16e2] Add clownfish.
 1 file changed, 1 insertion(+)
 create mode 100644 clownfish.html
```



CHERRY-PICK

"production" branch

```
$ git log --oneline
80a16e2 Add clownfish.
3d7225f Add guppies.
```

"development" branch

```
$ git log --oneline
c400bf0 Add Cthulhu.
53212e5 Add clownfish.
55ae374 Add sharks.
3d7225f Add guppies.
```

production

development

We copied a single commit to the current branch

CHERRY-PICK - EDIT COMMIT

We want a different message on the cherry-picked commit...

```
$ git cherry-pick --edit 5321
```

--edit lets you change messages

editor

```
Add clownfish.
#
# 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...
```

Shows an editor

CHERRY-PICK - EDIT COMMIT

We want a different message on the cherry-picked commit...

```
$ git cherry-pick --edit 5321
```

--edit lets you change messages

editor

```
Pull in clownfish from development.
#
# 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...
```

Enter your message, save, and quit

```
$ git log --oneline
03785d9 Pull in clownfish from development.
3d7225f Add guppies.
```

New commit has new message

CHERRY-PICK - CUSTOMIZE COMMIT

"production" branch

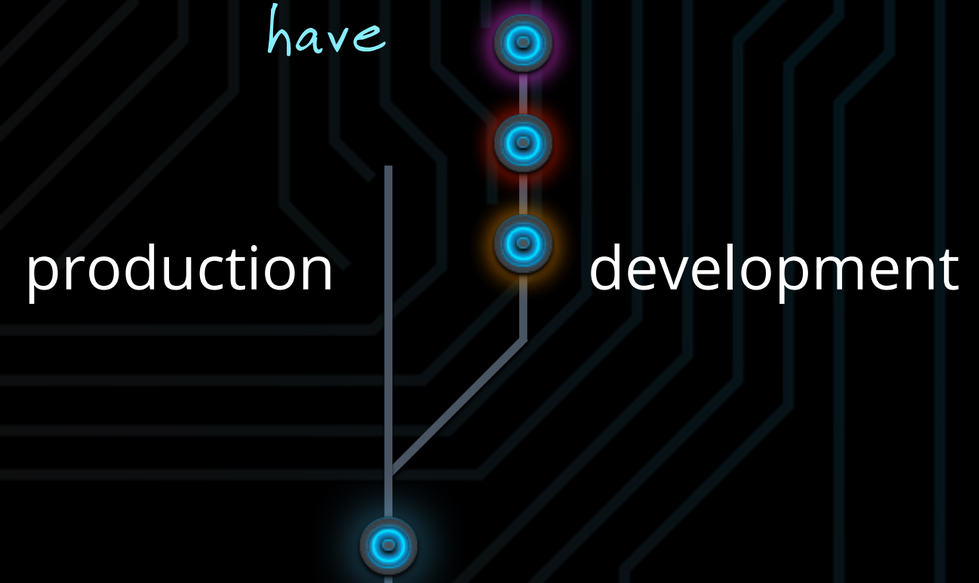
```
$ git log --oneline
3d7225f Add guppies.
```

"development" branch

```
$ git log --oneline
c400bf0 Add Cthulhu.
53212e5 Add clownfish.
55ae374 Add sharks.
3d7225f Add guppies.
```

We need to cherry-pick **and combine** these commits

we have



we want

CHERRY-PICK - CUSTOMIZE COMMIT

"production" branch

```
$ git log --oneline
3d7225f Add guppies.
```

"development" branch

```
$ git log --oneline
c400bf0 Add Cthulhu.
53212e5 Add clownfish.
55ae374 Add sharks.
3d7225f Add guppies.
```

```
$ git cherry-pick --no-commit 53212e5 55ae374
```

```
$ git status
# On branch production
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#  new file:   clownfish.html
#  new file:   shark.html
```

--no-commit pulls in changes and stages them, but doesn't commit

CHERRY-PICK - CUSTOMIZE COMMIT

```
$ git commit -m "Add clownfish and sharks (in separate tanks)."
[production bc75856] Add clownfish and sharks (in separate tanks).
 2 files changed, 2 insertions(+)
 create mode 100644 clownfish.html
 create mode 100644 shark.html
```

Now, manually commit the changes

production development

We cherry-picked the changes from two commits and made a single commit

CHERRY-PICK

We want to track which commit we cherry-picked from

```
$ git cherry-pick -x 5321
[production 6bbeedc] Add clownfish. (cherry picked from commit
53212e591c75100b9b462d3a4e8bdae56d9facae)
 1 file changed, 1 insertion(+)
 create mode 100644 clownfish.html
```

```
$ git log --oneline
6bbeedc Add clownfish. (cherry picked from commit
53212e591c75100b9b462d3a4e8bdae56d9facae)
3d7225f Add guppies.
```

-x adds source SHA to commit message

Only useful with public branches; don't use for local branches

CHERRY-PICK

We want to track who cherry-picked the commit along with the original committer

```
$ git cherry-pick --signoff 5321
[production ef0c80c] Add clownfish.
 Author: Gregg <gregg@example.com>
 1 file changed, 1 insertion(+)
 create mode 100644 clownfish.html
```

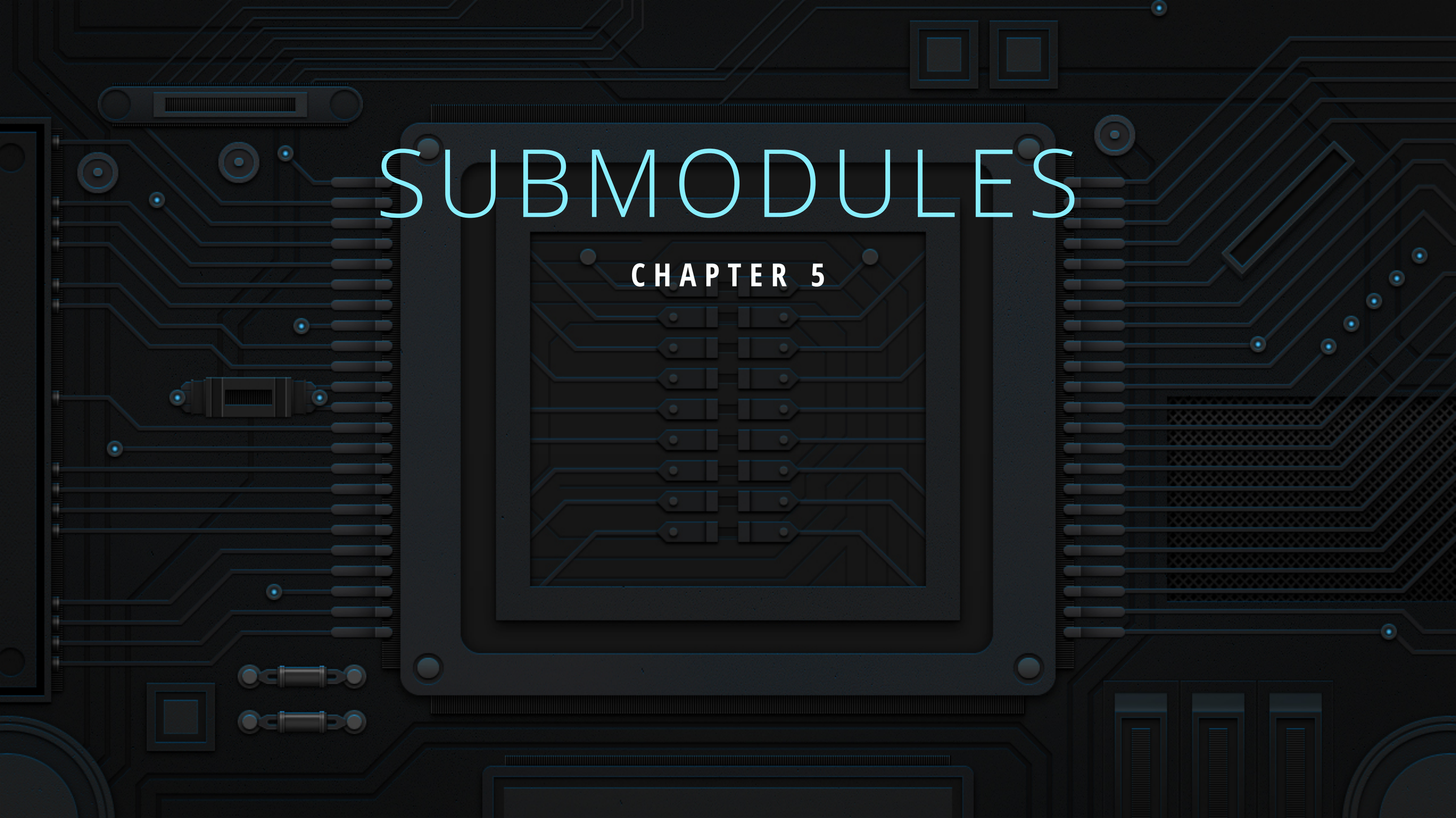
--signoff adds current user's name to commit message

```
$ git log
commit ef0c80cf5bb0357f8f39e0ae76628043ccf0ee91
Author: Gregg <gregg@example.com>
Date:   Wed Mar 20 22:20:07 2013 -0700

    Add clownfish.

    Signed-off-by: Jane <jane@example.com>
...
```

committed by...

signed off by...


GIT REAL
2


SUBMODULES
CHAPTER 5

BAD WAYS TO SHARE LIBRARIES

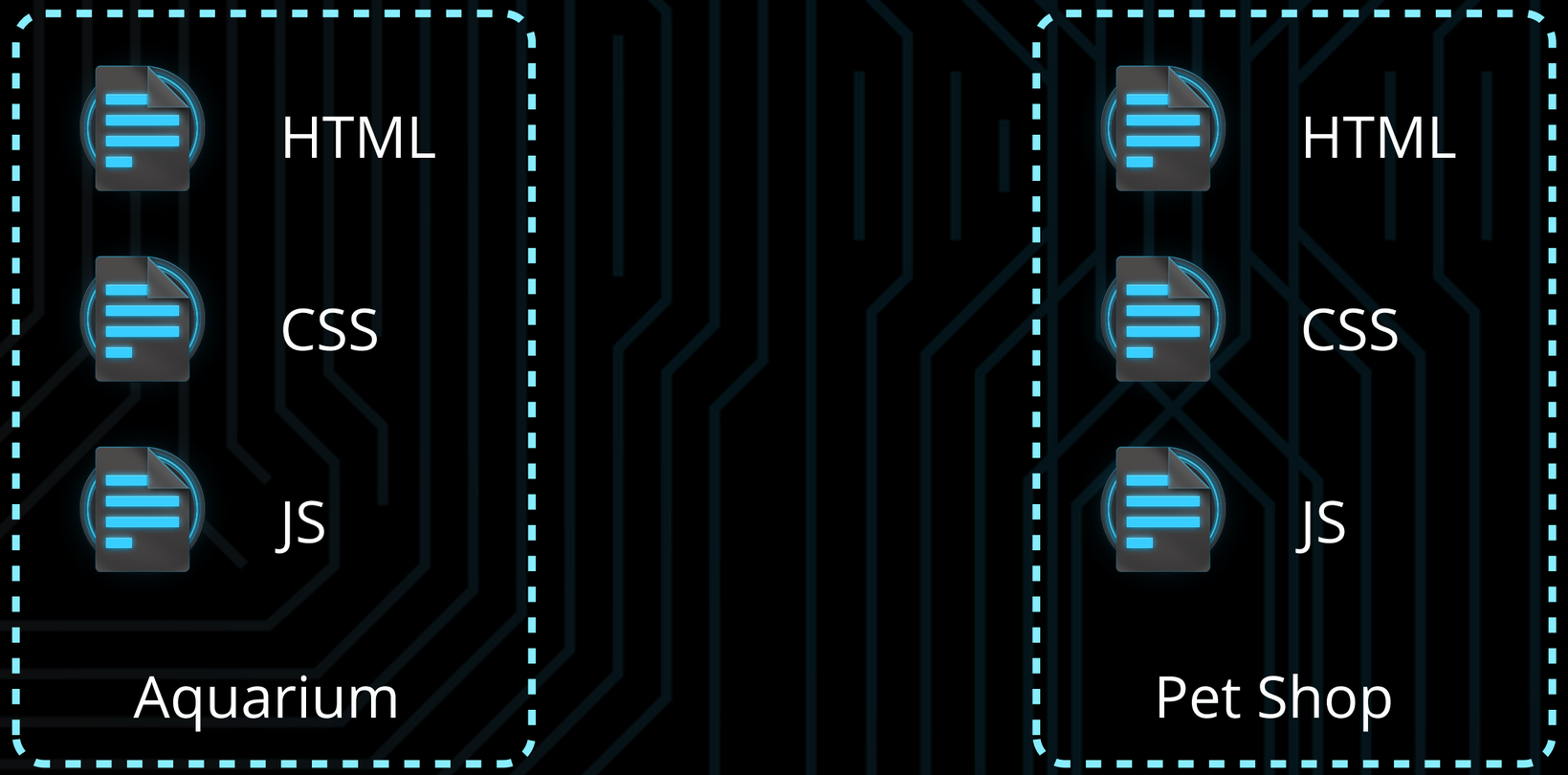
- Copy the libraries into the projects
  - Can't share changes back
  - Library quickly gets outdated
- Post libraries on a central server
  - Can't make library changes without a project to test it in

GIT SUBMODULES

- A Git repo *inside* a Git repo
  - Pull down updates easily
  - Test your changes with an *actual* dependent project
  - Share changes easily
  - History independent of containing repo

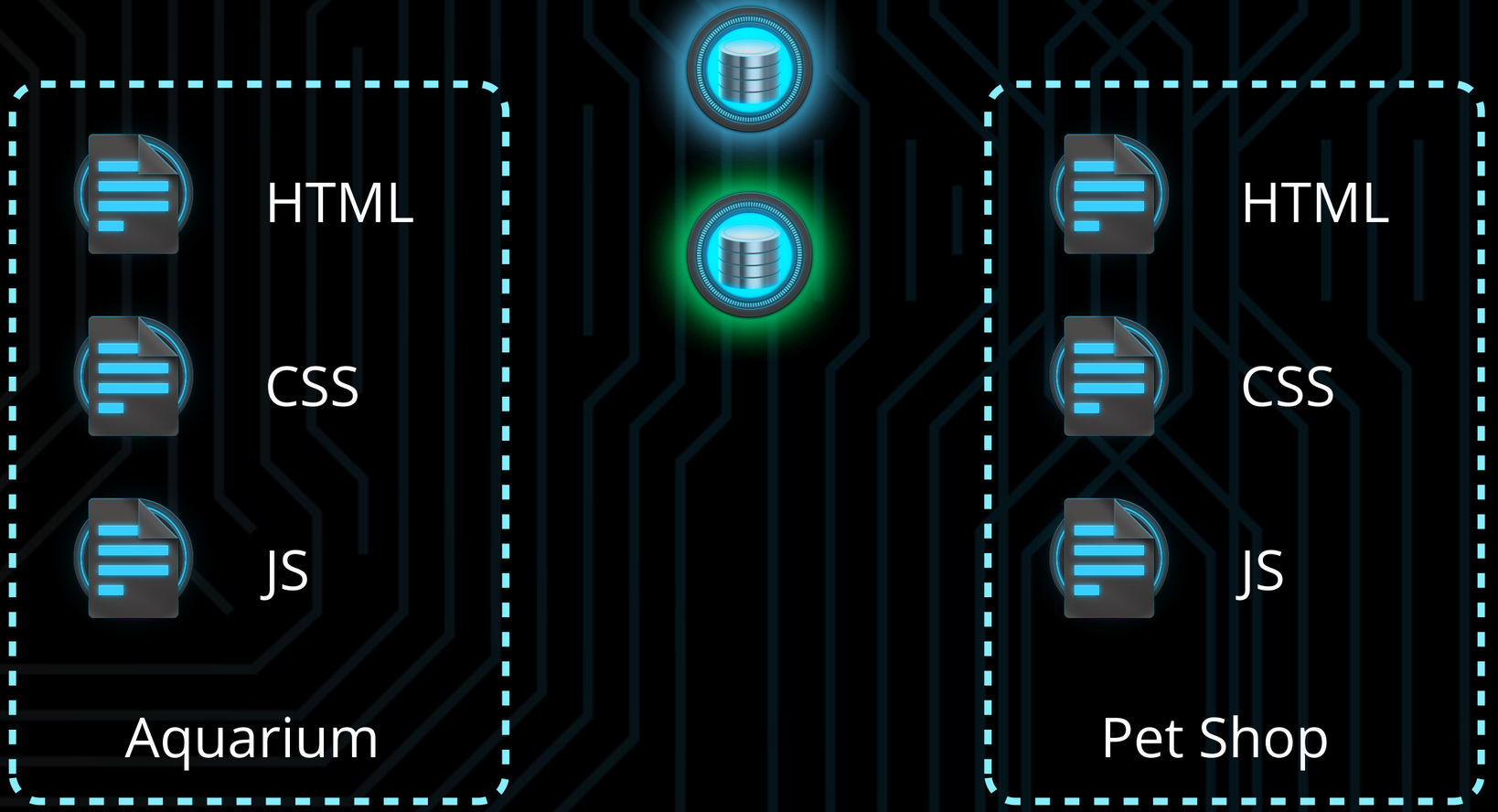
CONVERTING TO SUBMODULES

The aquarium and pet shop sites have pretty much the same CSS and JavaScript resources. We want the sites to be able to share them.



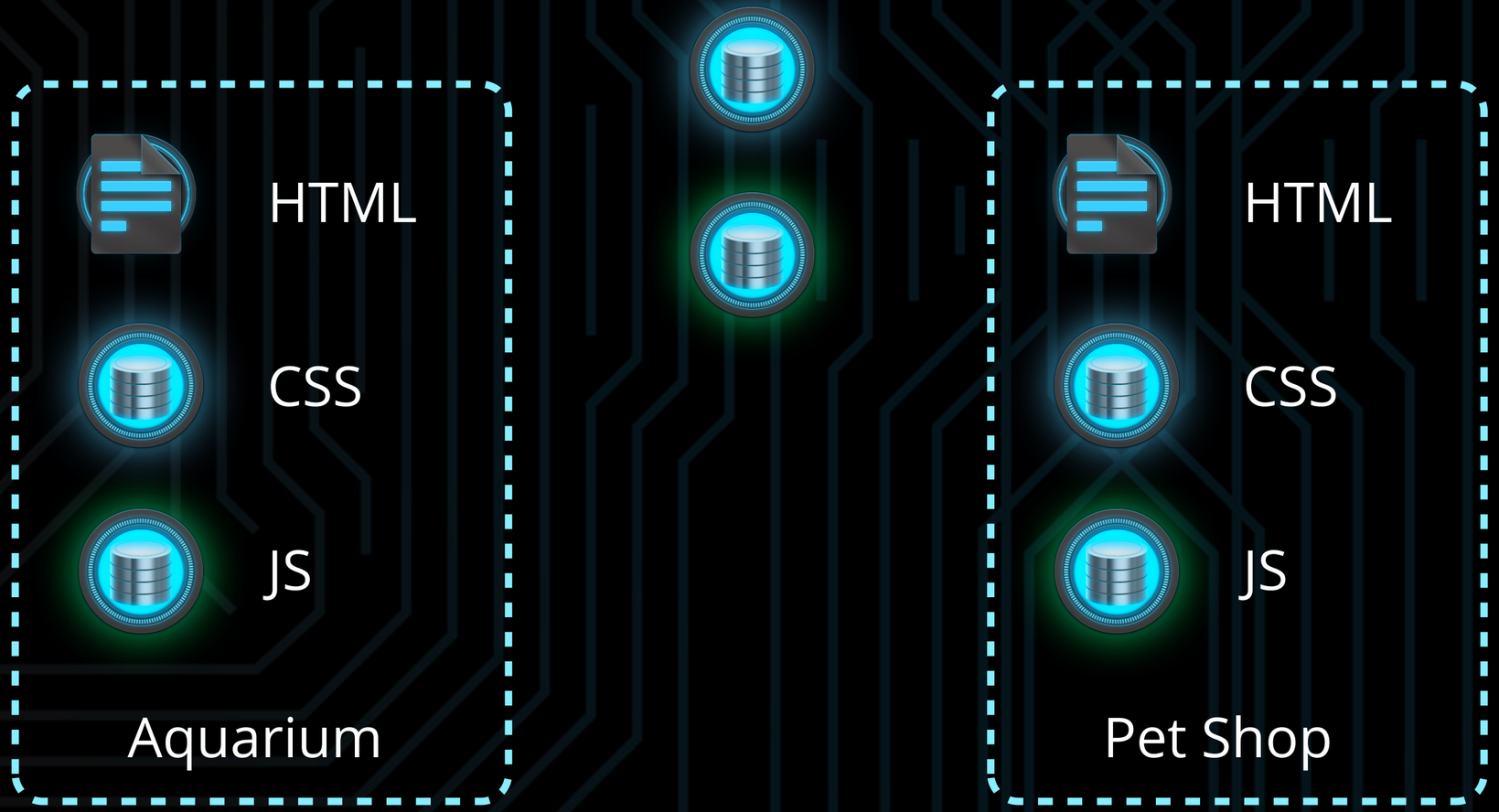
CONVERTING TO SUBMODULES

The plan is to convert the CSS and JavaScript resources to Git repos, hosted on a central server



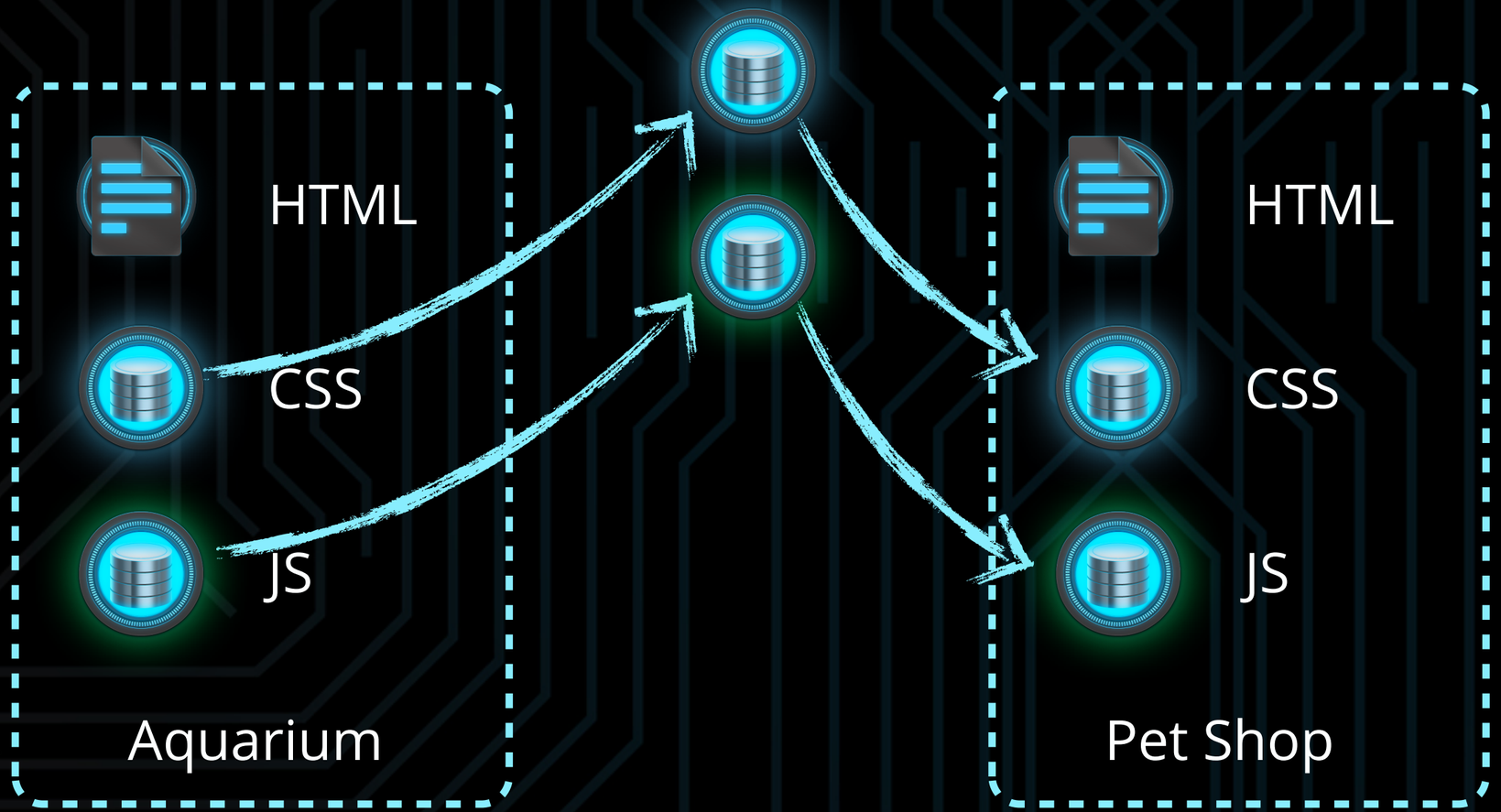
CONVERTING TO SUBMODULES

From there, they'll be pulled into the aquarium and pet shop projects as submodules



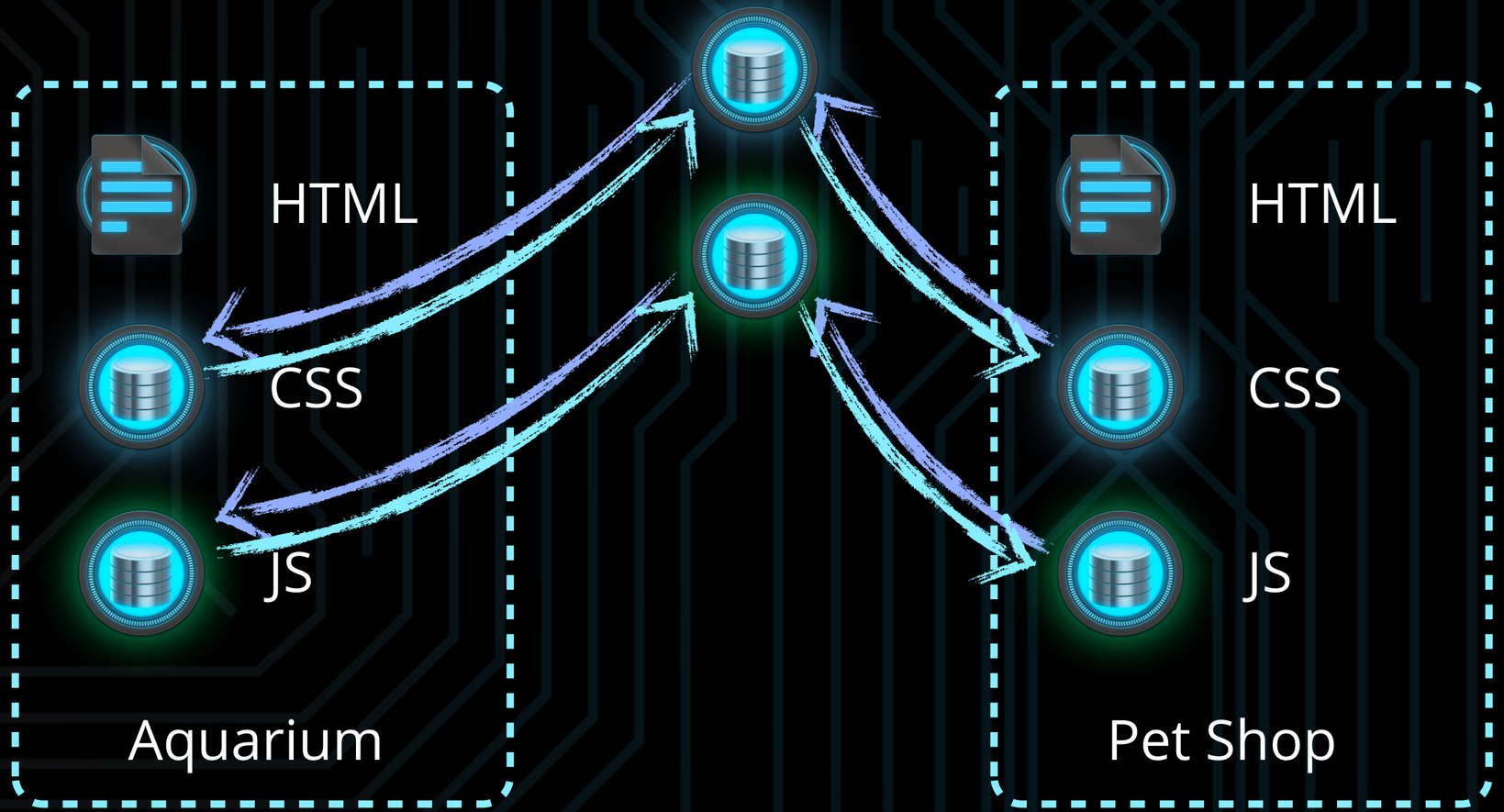
CONVERTING TO SUBMODULES

The aquarium project can push changes that can be pulled in by the pet shop project



CONVERTING TO SUBMODULES

And the pet shop project can contribute back as well



ADDING SUBMODULES

Jane is ready to add the CSS repo into the aquarium project as a submodule

```
$ git submodule add git@example.com:css.git
Cloning into 'css'...
done.
```

New files added:

```
$ git status
# On branch master
# Changes to be committed:
#   new file:    .gitmodules
#   new file:    css
```

config for submodules

submodule itself

```
$ git commit -m "Add CSS submodule."
$ git push
```


GITMODULES FILE

```
$ cat .gitmodules
[submodule "css"]
    path = css
    url = git@example.com:css.git
[submodule "js"]
    path = js
    url = git@example.com:js.git
```

local path

origin

additional modules added to file later

Config for submodules

HOW TO MODIFY SUBMODULES

```
$ cd css
$ git checkout master
```

Submodules don’t start out on a branch

Make changes to our files

```
$ git status
# On branch master
# Changes not staged for commit:
#   modified:   example.css
#
no changes added to commit

$ git commit -am "Update menu font."
[master 1e7f964] Update menu font.
...
$ git push
```

Then, we need to update our parent repo!

UPDATE THE PARENT PROJECT

The parent project still references the old commit on the submodule

```
$ cd ..

$ git status
# On branch master
# Changes not staged for commit:
#   modified:   css (new commits)
no changes added to commit
```

There are new commits on the submodule.

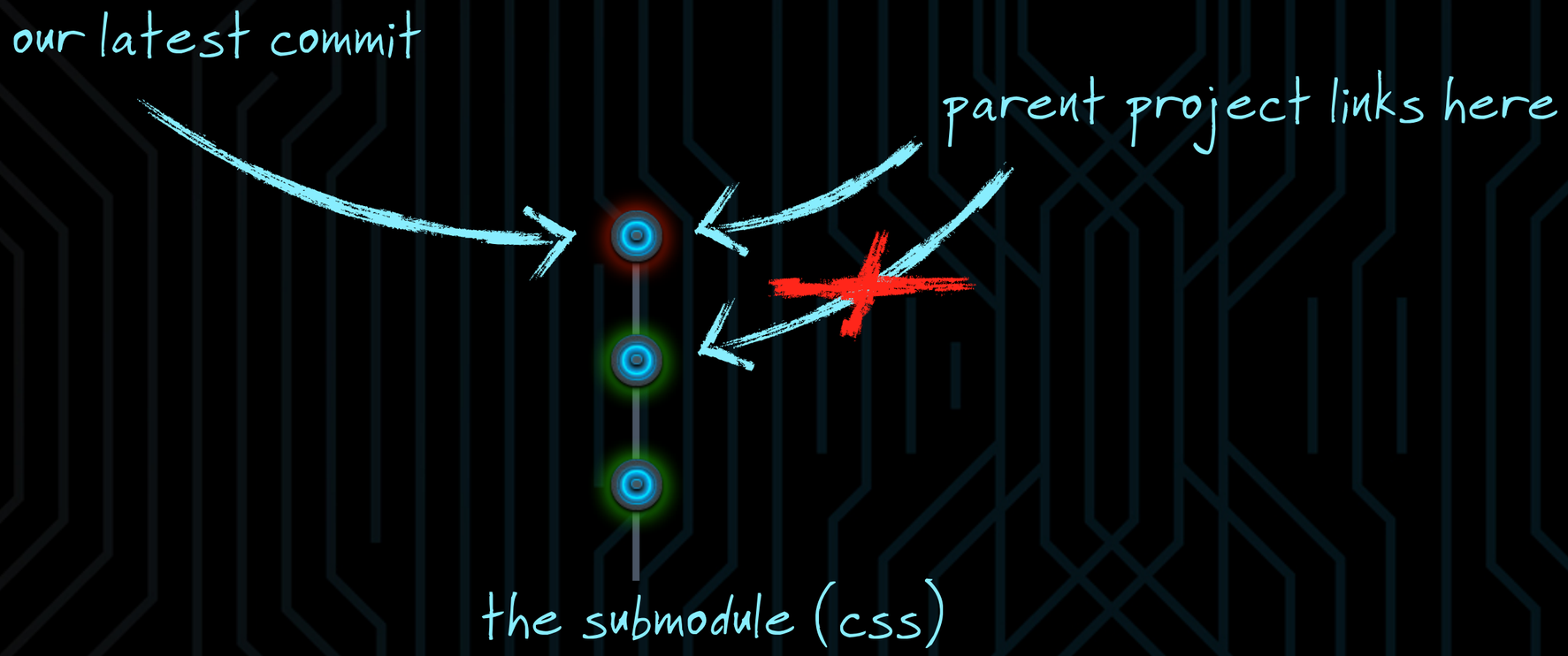
```
$ git add css

$ git commit -m "Update menu font in css."
[master 4e4e316] Update menu font in css.
 1 file changed, 1 insertion(+), 1 deletion(-)

$ git push
```

We add...

commit...

and push

OUR PARENT NEEDS TO BE UPDATED

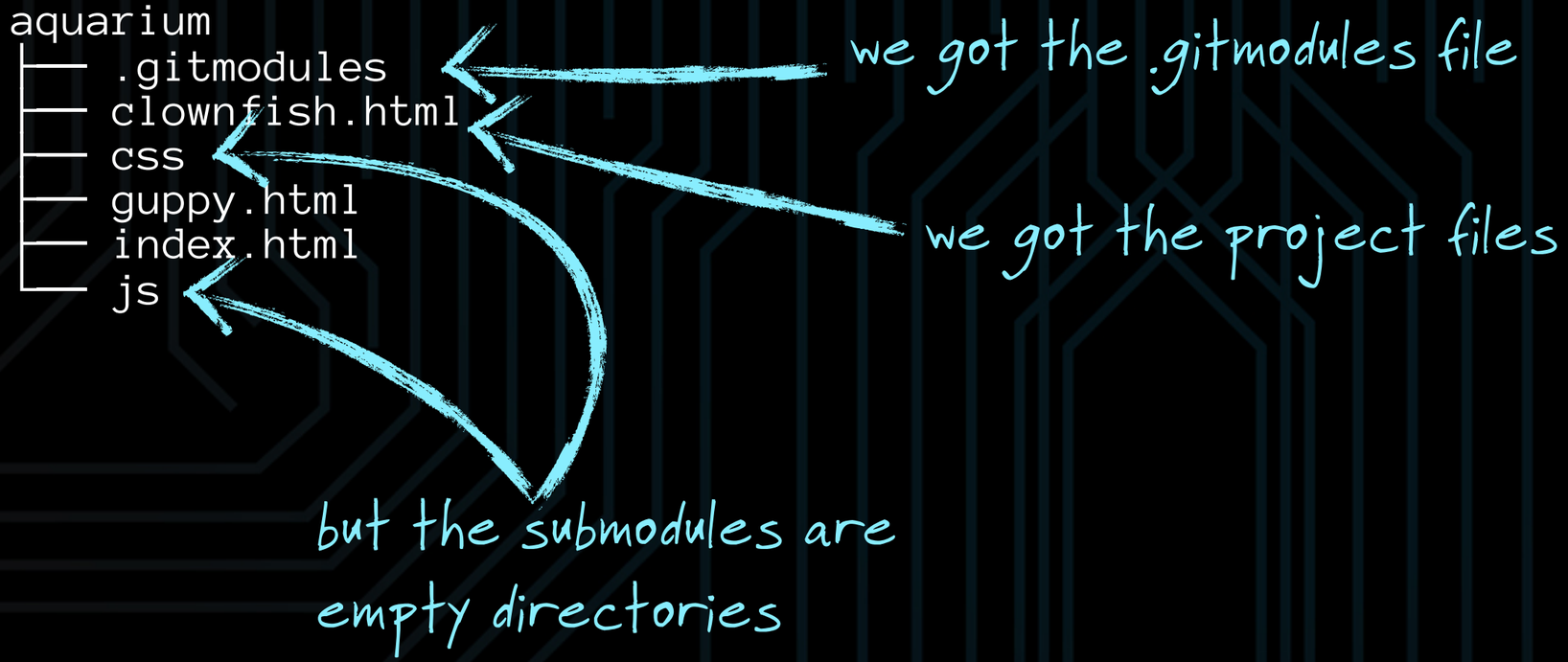


CLONING PROJECTS

Gregg needs to clone the aquarium project to his machine

```
$ git clone git@example.com:aquarium.git
Cloning into 'aquarium'...
done.
```

The process starts the same as always



SUBMODULE INIT

To get the submodules, first Gregg needs to initialize them:

```
$ git submodule init
Submodule 'css' (git@example.com:css.git) registered for path 'css'
Submodule 'js' (git@example.com:js.git) registered for path 'js'
```

Git goes through your .gitmodules file and automatically adds an entry to config for each submodule

.git/config

```
                                    editor
...
[submodule "css"]
    url = git@example.com:css.git
[submodule "js"]
    url = git@example.com:js.git
```

SUBMODULE UPDATE

Now Gregg can run "git submodule update". For each submodule, it will:

```
$ git submodule update
Cloning into 'css'...
done.
Submodule path 'css': checked out '4eac8bd...'
Cloning into 'js'...
done.
Submodule path 'js': checked out '01cb95c...'
```

Clone the repo

Check out the commit specified by the parent project

```
aquarium
├── clownfish.html
├── css
│   └── example.css
├── guppy.html
├── index.html
└── js
    └── example.js
```

Now your submodules have files!

PULL SUBMODULES

Gregg has made some changes to the project, and Jane wants to retrieve it

```
$ git pull
...
From git@example.com:aquarium.git
   2863f5c..4e4e316  master     -> origin/master
Fetching submodule css
...
From git@example.com:css.git
   4eac8bd..9fcd707  master     -> origin/master
Updating 2863f5c..4e4e316
Fast-forward
 css | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

She pulls as usual

She sees that the CSS submodule has been updated

PULL SUBMODULES

```
$ git status
# On branch master
# Changes not staged for commit:
#
#   modified:   css (new commits)

$ git diff
diff --git a/css b/css
index 9fcd707..4eac8bd 160000
@@ -1 +1 @@
-Subproject commit 9fcd707...
+Subproject commit 4eac8bd...
```

The output of "git status" also shows submodule updates

So does the output of "git diff"

The pull only retrieves the new SHA to use, not the submodule contents

PULL SUBMODULES

To get changes in the submodule itself...

```
$ git submodule update
Submodule path 'css': checked out '9fcd707...'
```

from the parent project

Retrieves submodule's file updates

```
$ git status
# On branch master
nothing to commit (working directory clean)
```

Parent project's status is back to normal

MODIFY SUBMODULES

Jane needs to make some changes inside the css submodule.

```
$ cd css
$ git status
# Not currently on any branch.
# Changes not staged for commit:
#
#   modified:   example.css
#
no changes added to commit

$ git add example.css
$ git commit -m "Update menu font."
[detached HEAD b6bb78f] Update menu font.
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Now we can review and commit the actual file changes

b6bb78f

But wait, this could mean trouble...

MODIFY SUBMODULES

```
$ git push
Everything up-to-date
```

Nothing is sent!

```
$ git branch
* (no branch)
  master
```

active branch

"git submodule update" checks out submodules in a "headless" state

The commit we just made didn't go to a branch!

b6bb78f

(But don't panic...)

MODIFY SUBMODULES

Jane needs to associate the submodule commit to a branch

```
$ git checkout master
Warning: you are leaving 1 commit behind,
not connected to any of your branches:

  b6bb78f Update menu font.

If you want to keep them by creating a new
branch, this may be a good time to do so
with:

 git branch new_branch_name b6bb78f

Switched to branch 'master'
```

Git will suggest a command, but this would add it to a NEW branch

She checks out the branch she wants to move it to

Git warns about the "orphaned" commit

MODIFY SUBMODULES

Jane meant for this commit to go on the "master" branch...

```
$ git merge b6bb78f
Updating 29b657e..b6bb78f
Fast-forward
 example.css | 2 +-
 1 file changed, 1 insertion(+), 1
deletion(-)
```

She merges it in directly

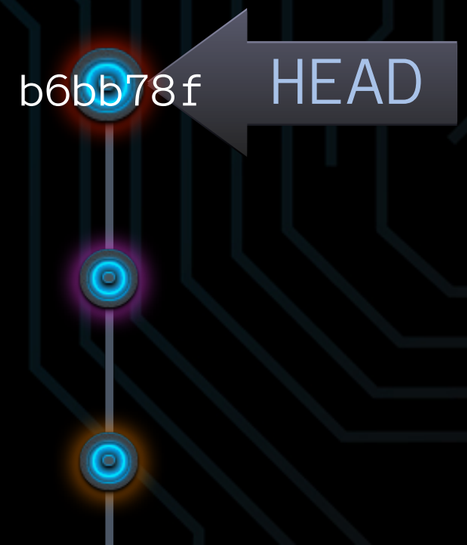
```
$ git log --oneline
b6bb78f Update menu font.
29b657e Revise content padding.
4eac8bd Initial commit.
```

It becomes part of "master" history

```
$ git push
...
To git@example.com:css.git
   29b657e..b6bb78f  master -> master
```

Now she can push

don't forget to!



MODIFY PARENT PROJECT

The parent project still references the old commit

```
$ cd ..

$ git status
# On branch master
# Changes not staged for commit:
#   modified:   css (new commits)
no changes added to commit
```

```
$ git add css

$ git commit -m "Update menu font in css."
[master 4e4e316] Update menu font in css.
 1 file changed, 1 insertion(+), 1 deletion(-)

$ git push
```

She adds...

commits...

and pushes

PUSHING SUBMODULES

So, why is it so important to push submodule updates?

Step 1	/submodule	`$ git push`
Step 2	/	`$ git push`

if Jane had forgotten to push the submodule...

```
$ git submodule update
fatal: reference is not a tree: b6bb78f...
Unable to checkout 'b6bb78f...' in submodule path 'css'
```

Gregg would see this

Collaborators would be stuck until Jane pushed the submodule!

PUSHING SUBMODULES

Worried about forgetting?

```
$ git push --recurse-submodules=check
The following submodule paths contain changes that can
not be found on any remote:
  js
Please try
    git push --recurse-submodules=on-demand
or cd to the path and use
    git push
to push them to a remote.
fatal: Aborting.
```

Will abort a push if you haven't pushed a submodule

Run this when you push from the parent directory

PUSHING SUBMODULES

To push all repositories (even submodules)

```
$ git push --recurse-submodules=on-demand
Pushing submodule 'js'
Counting objects: 5, done.
Total 3 (delta 0), reused 0 (delta 0)
To git@example.com:js.git
   ddfbdcd..136b9d4  master -> master
Counting objects: 3, done.
Delta compression using up to 4 threads.
Total 2 (delta 1), reused 0 (delta 0)
To git@example.com:aquarium.git
   389f86d..c5c9ec9  master -> master
```

PUSHING SUBMODULES

Make it an alias!

```
$ git config alias.pushall "push --recurse-submodules=on-demand"
```

```
$ git pushall
Pushing submodule 'js'
Counting objects: 5, done.
Total 3 (delta 0), reused 0 (delta 0)
To git@example.com:js.git
   ddfbdcd..136b9d4  master -> master
Counting objects: 3, done.
Delta compression using up to 4 threads.
Total 2 (delta 1), reused 0 (delta 0)
To git@example.com:aquarium.git
   389f86d..c5c9ec9  master -> master
```



GIT REAL
2

REFLOG

CHAPTER 6

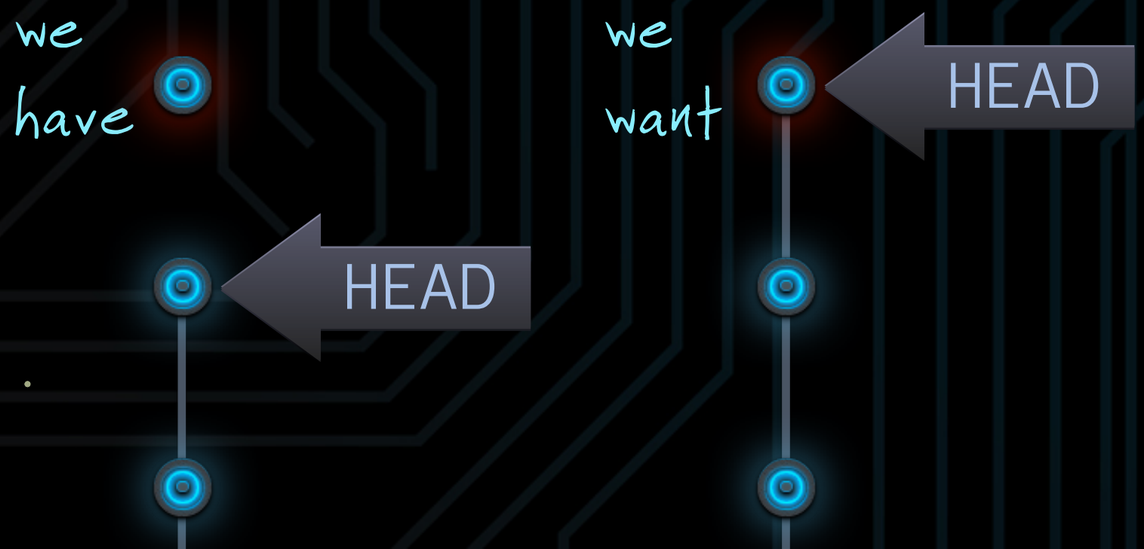
LOST DATA

Gregg decided he didn't need that latest commit, and did a hard reset

```
$ git log --pretty=oneline
1e62107... Add third section.
43c13e7... Add second section.
2bd404a... Add first section.

$ git reset --hard 43c1
HEAD is now at 43c13e7 Add second section.
```

And now he's realized
that was a mistake

Git *never* deletes a commit
(partly because of situations like this)



It's just that no
branch points to
it right now

REFLOG

How can you get those commits back? They're not in the log...

```
$ git log --pretty=oneline
43c13e7... Add second section.
2bd404a... Add first section.
```

But Git keeps a *second* log, only in your local repo, called the *reflog*

```
$ git reflog
43c13e7 HEAD@{0}: reset: moving to 43c1
1e62107 HEAD@{1}: commit: Add third section.
43c13e7 HEAD@{2}: commit: Add second section.
2bd404a HEAD@{3}: commit (initial): Add first section.
```

here's after the reset

here's the commit you want back

REFLOG

Git updates the reflog anytime HEAD moves (due to new commits, checking out branches, or resetting)

```
$ git reflog
43c13e7 HEAD@{0}: reset: moving to 43c1
1e62107 HEAD@{1}: commit: Add third section.
43c13e7 HEAD@{2}: commit: Add second section.
2bd404a HEAD@{3}: commit (initial): Add first section.
```

commit SHA

reflog shortname

operation that caused HEAD to move

HEAD@{0} is always your current commit

RESTORING COMMITS

```
$ git reflog
43c13e7 HEAD@{0}: reset: moving to 43c1
1e62107 HEAD@{1}: commit: Add third section.
43c13e7 HEAD@{2}: commit: Add second section.
2bd404a HEAD@{3}: commit (initial): Add first section.
```

we need this commit back

```
$ git reset --hard 1e62
```

OR

```
$ git reset --hard HEAD@{1}
```

Now that we know where the commit is, we can point the branch back to it

you can use reflog shortnames in place of commit SHAs

RESTORING COMMITS

If we run a "git log", we'll see the commit is back

```
$ git log --oneline
1e62107 Add third section.
43c13e7 Add second section.
2bd404a Add first section.
```

The files will be waiting in the directory

LOCAL-ONLY

Note that your reflog exists only in your local repository

```
$ git log --oneline
1e62107 Add third section.
43c13e7 Add second section.
2bd404a Add first section.
```

If Jane clones Gregg's repo, the log will still be there

```
$ git reflog
1e62107 HEAD@{0}: clone: from git@example.com:aquarium.git
```

But the reflog starts over from scratch

DELETED BRANCHES

Jane decided the aquarium project doesn't need its "aviary" branch

```
$ git branch -d aviary
error: The branch 'aviary' is not fully merged.
If you are sure you want to delete it, run 'git branch -D aviary'.

$ git branch -D aviary
Deleted branch aviary (was 280ee63).
```

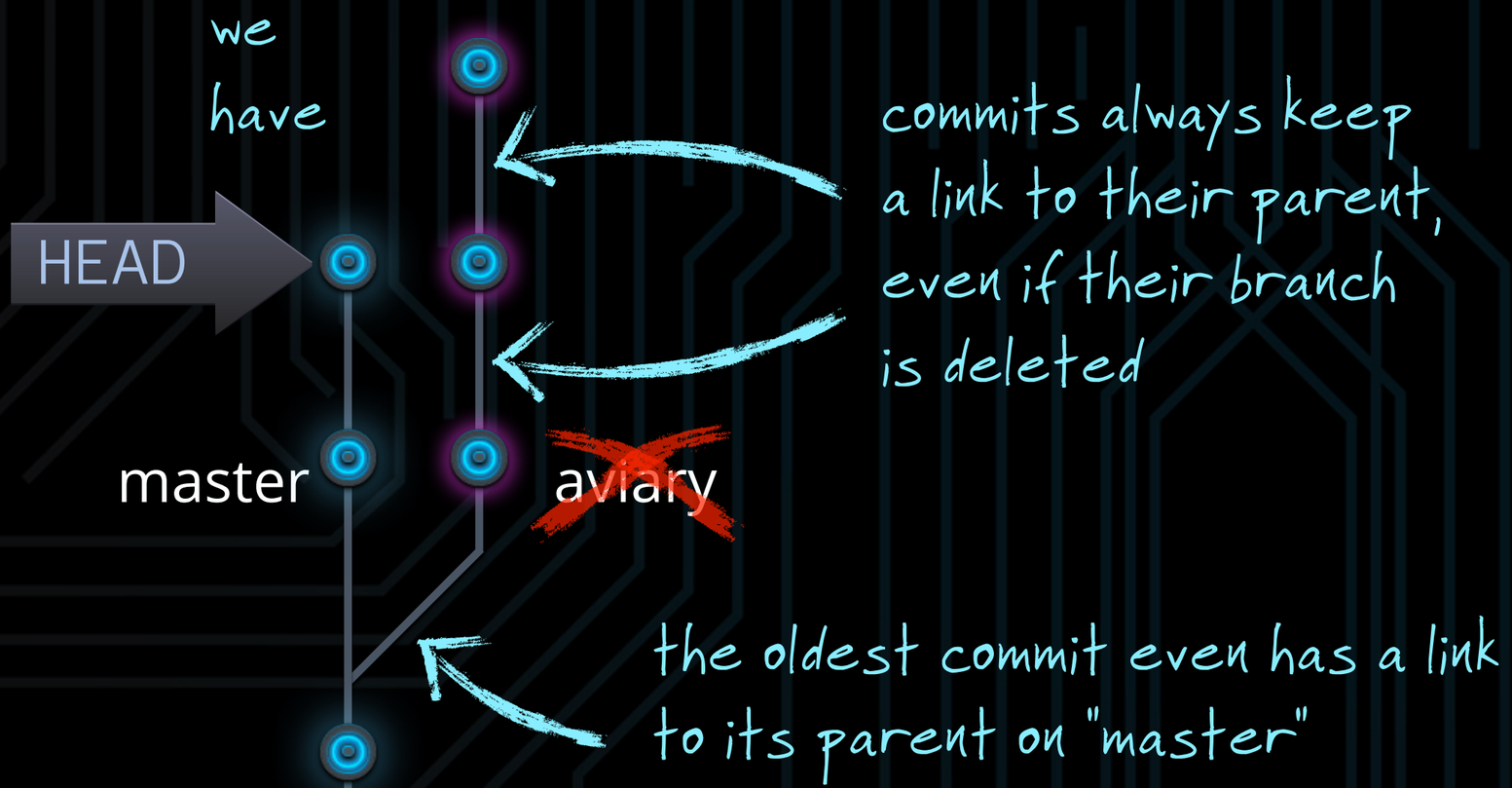
It wasn't merged

So she force-deleted it

She forgot that it had a new menu system she wants for the next feature

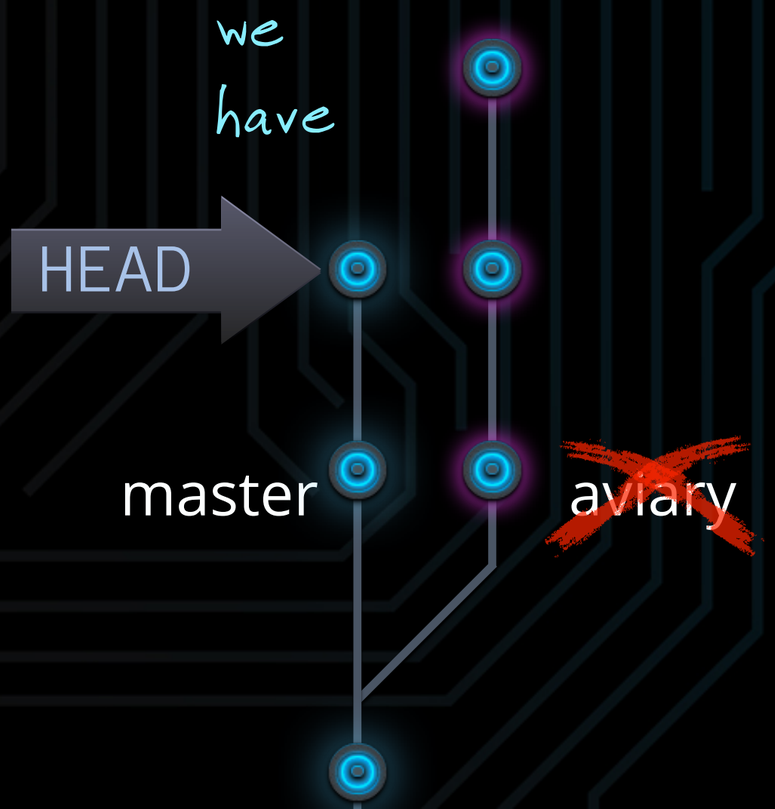
DELETED BRANCHES

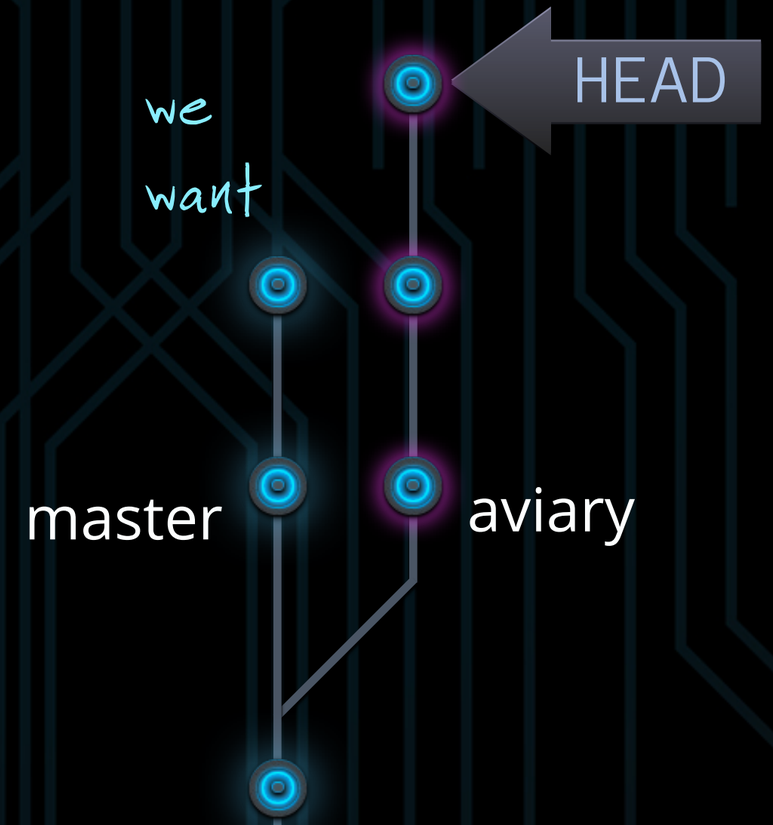
As before, the commits aren't really gone, just the branch



DELETED BRANCHES

If we can find the latest commit and re-create the branch that points to it, it'll be like the branch was never deleted

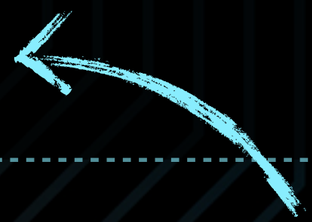




WALK-REFLOGS

The output of "git reflog" isn't as detailed as the full log

```
$ git log --walk-reflogs
commit b7a5df73f03de5bdf534f889d8d2c0a9bf89b1e0
Reflog: HEAD@{0} (Jane <jane@example.com>)
Reflog message: checkout: moving from aviary to master
Author: Jane <jane@example.com>

    Add clownfish.

commit 280ee635634f8c97b6e7d47beef1a110a1679811
Reflog: HEAD@{1} (Jane <jane@example.com>)
Reflog message: commit: Add birds.
Author: Jane <jane@example.com>

    Add birds.
...
```



There's the most recent commit from our deleted branch!

Pass the --walk-reflogs option to "git log" to see the reflog info in full log format

Includes reflog shortnames and messages

RESTORING DELETED BRANCHES

Instead of a reset, Jane creates another branch, and points it at that commit

```
$ git branch aviary 280e
```

OR

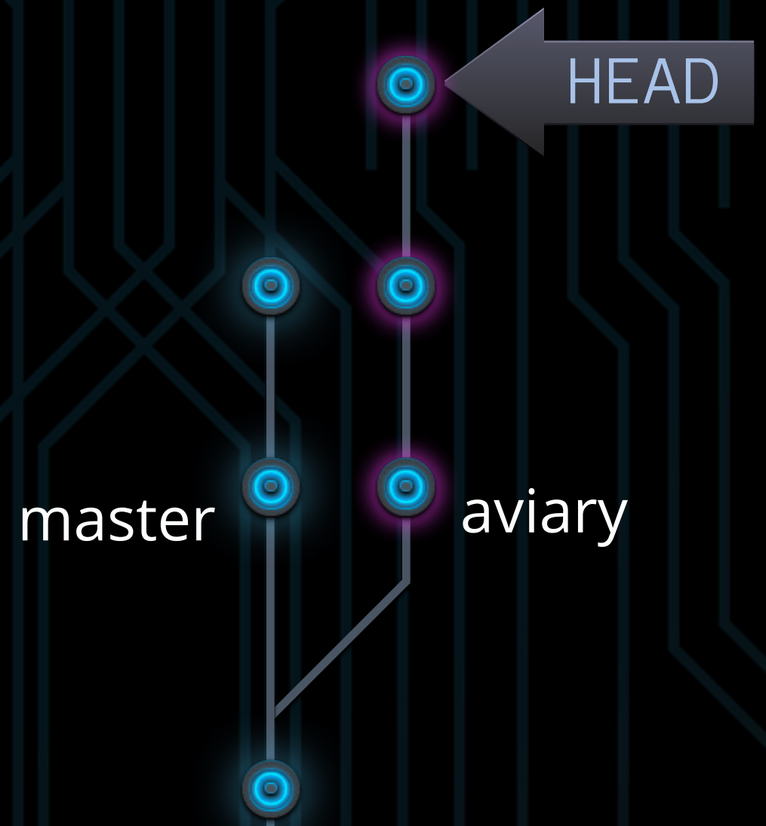
```
$ git branch aviary HEAD@{1}
```

SHA or reflog shortname

RESTORING DELETED BRANCHES

When she checks out the new branch, the aviary work is back

```
$ git checkout aviary

$ git log --oneline
280ee63 Add birds.
b7a5df7 Add license.
```



The deleted branch is resurrected!