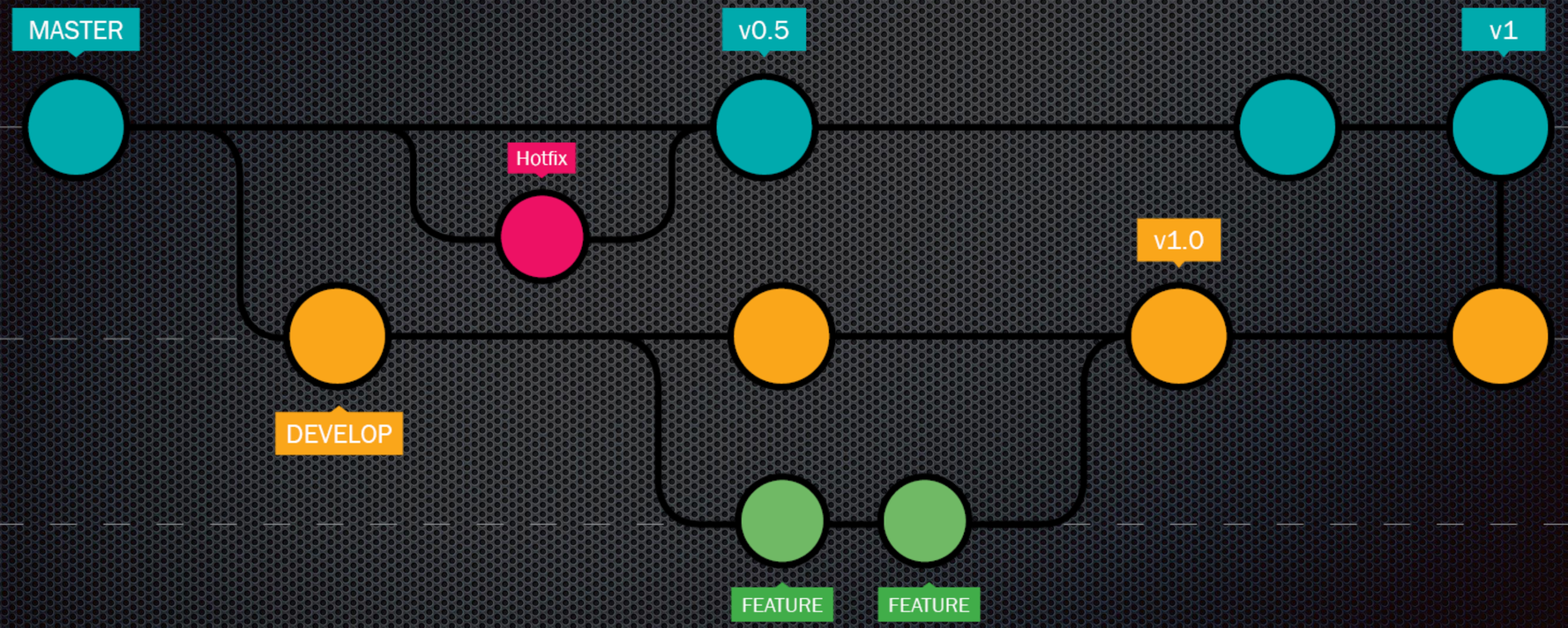
presents

Advanced source control

Branching



Branching Benefits

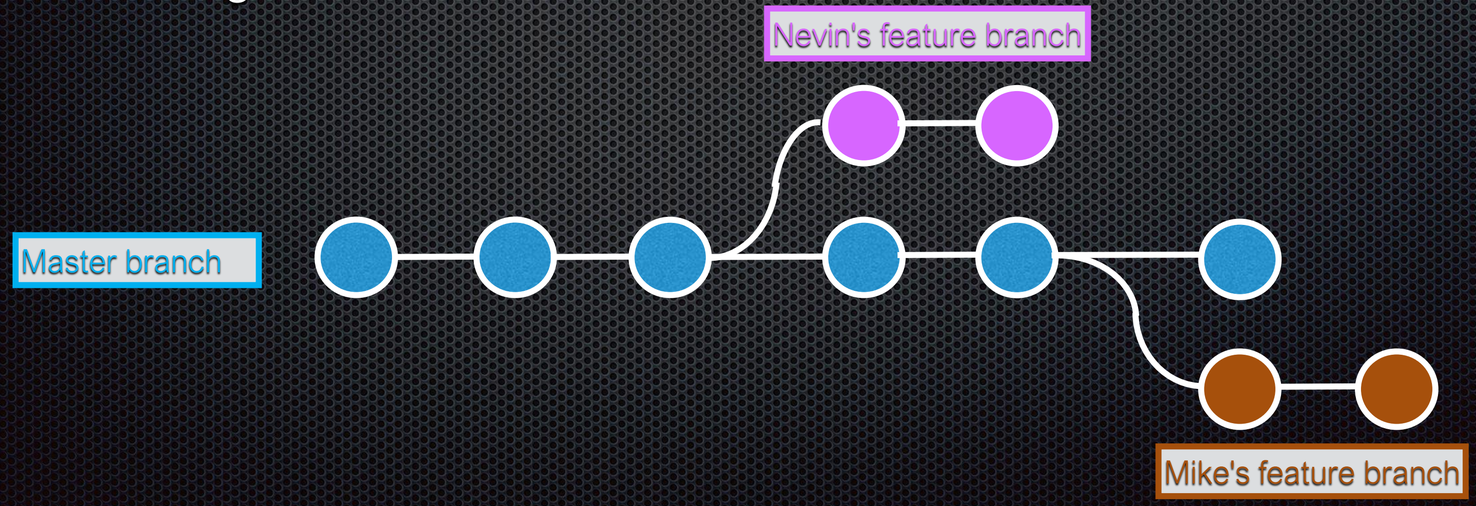
"One does not simply push to master"

Feature Branch Workflow

-One of the biggest advantages of Git is its branching capabilities. Git branches are cheap and easy to merge. This facilitates the feature branch workflow popular with many Git users.

-Feature branches provide an isolated environment for every change to your codebase. When a developer wants to start working on something—no matter how big or small—they create a new branch. This ensures that the master branch always contains production-quality code.

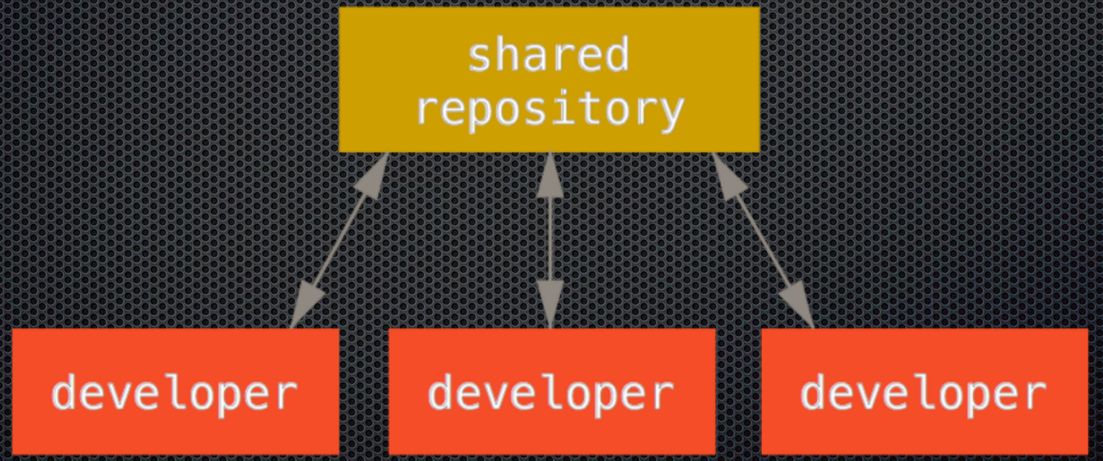
-Using feature branches is not only more reliable than directly editing production code, but it also provides organizational benefits.



Distributed Development

-Each developer gets their own local repository, complete with a full history of commits.

-And, similar to feature branches, distributed development creates a more reliable environment. Even if a developer obliterates their own repository, they can simply clone someone else's and start anew.



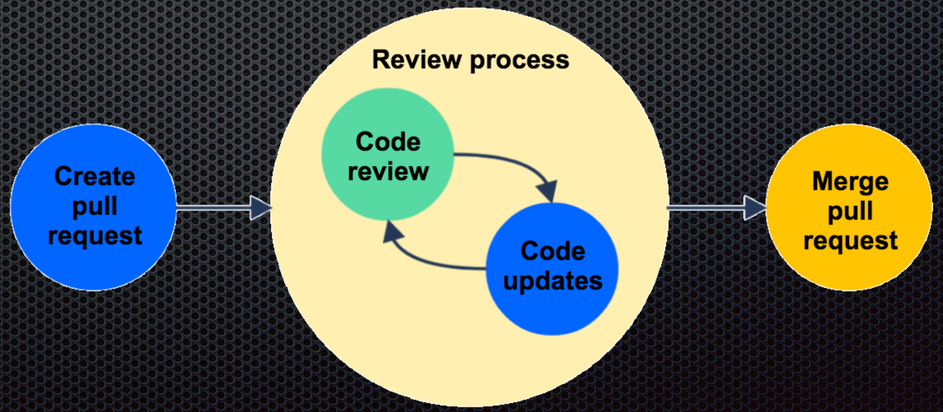
Pull Requests

-A pull request is a way to ask another developer to merge one of your branches into their repository.

-Since they’re essentially a comment thread attached to a feature branch, pull requests are extremely versatile.

-Easy for project leads to keep track of changes, but also lets developers initiate discussions around their work before integrating it with the rest of the codebase.

-When a developer gets stuck with a hard problem, they can open a pull request to ask for help from the rest of the team. Alternatively, junior developers can be confident that they aren’t destroying the entire project by treating pull requests as a formal code review.



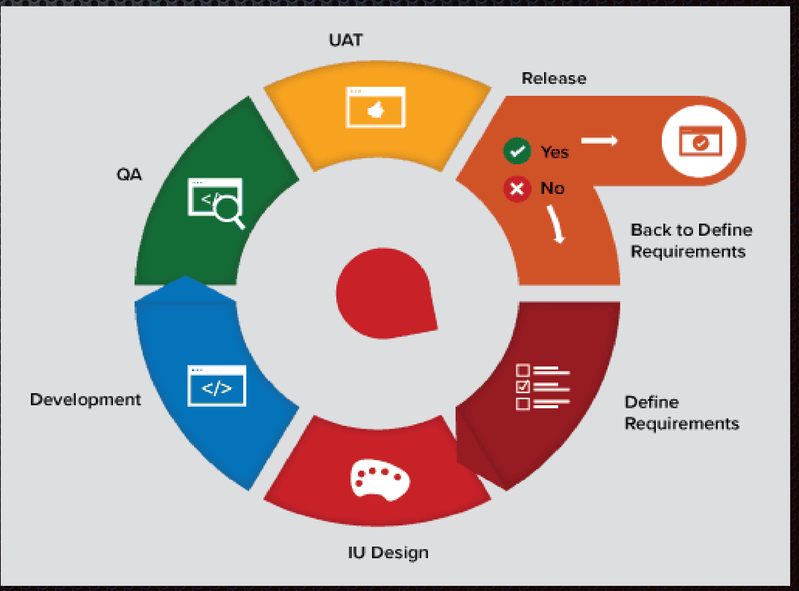
Faster Release Cycle

-The ultimate result of feature branches, distributed development, pull requests, and a stable community is a faster release cycle.

-Using feature branches is not only more reliable than directly editing production code, but it also provides organizational benefits and greatly speed up production time.

-These capabilities facilitate an agile workflow where developers are encouraged to share smaller changes more frequently. In turn, changes can get pushed down the deployment pipeline faster than the monolithic releases common with centralized version control systems.

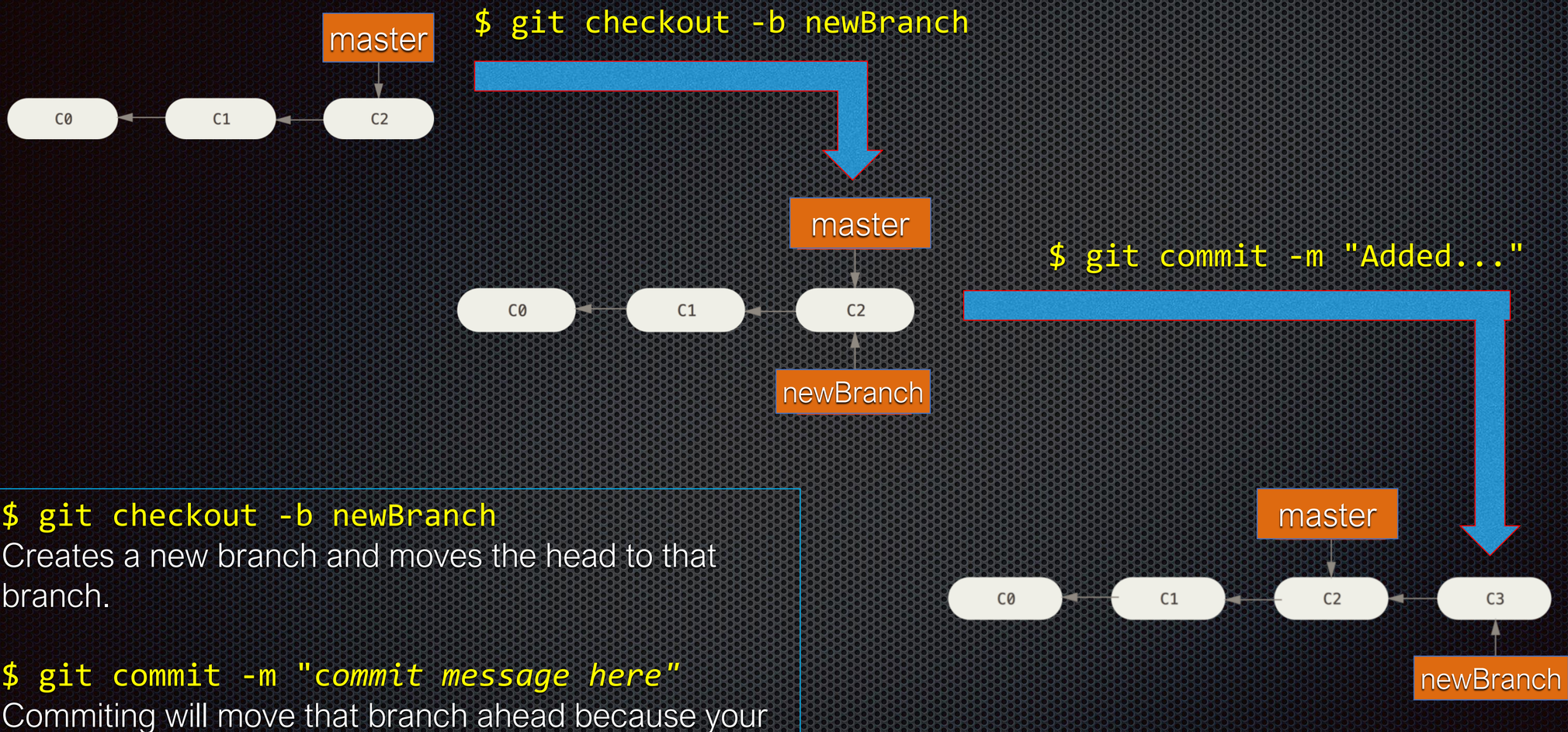
-More frequent releases means more frequent customer feedback and faster updates in reaction to that feedback. Instead of waiting for the next release 8 weeks from now, you can push a solution out to customers as quickly as your developers can write the code.



Basic Commands (Branching)

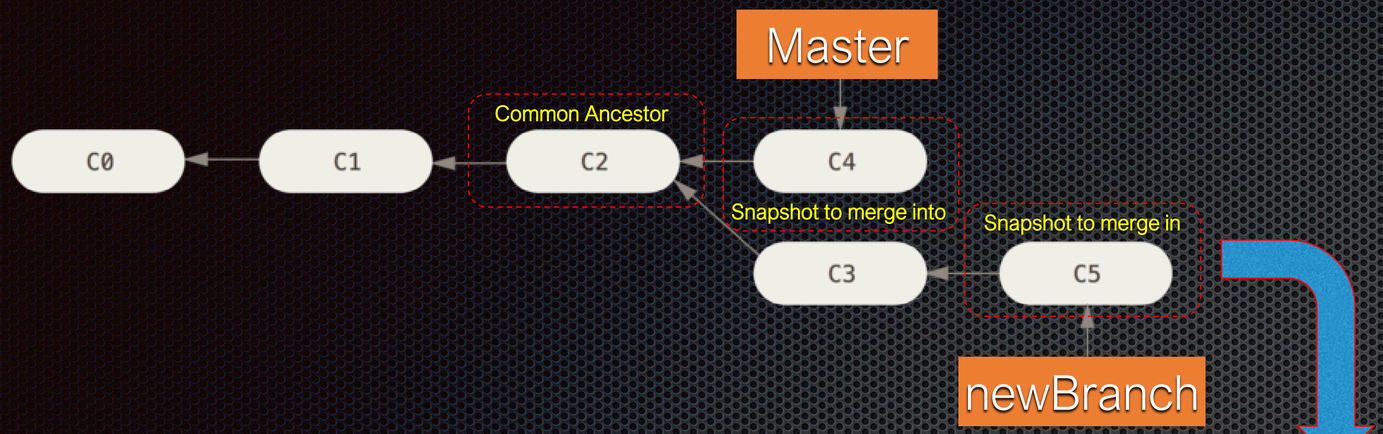
- git branch // lists all local branches in repo
- git log // lists all commits in current branch's history
- git branch [branch-name] // creates new branch
- git checkout [branch-name] // switches to branch
- git checkout -b [branch-name] // creates then switches to branch
- git merge [branch-name] // combine branch into current branch
- git branch -d [branch-name] // delete branch

Creating a new Branch

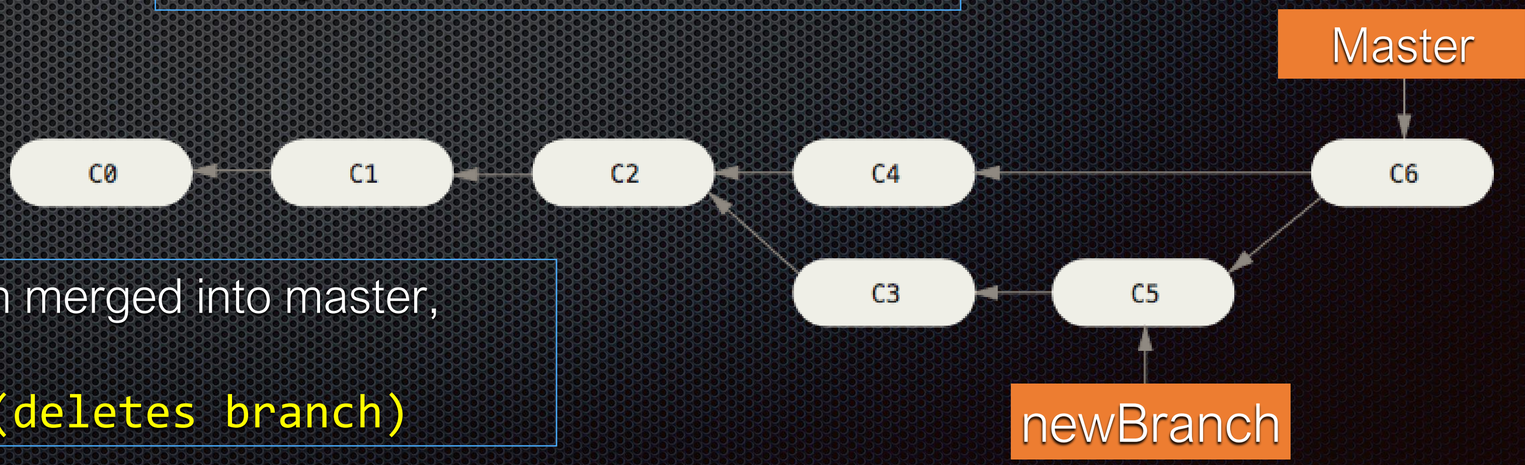


`$ git checkout -b newBranch`
Creates a new branch and moves the head to that branch.

`$ git commit -m "commit message here"`
Commiting will move that branch ahead because your head is pointing to it.




```
$ git checkout master
```
switches to the master branch
```
$ git merge newBranch
```
merges newBranch into master



Now that the newBranch has been merged into master, there is no further need for it.

```
$ git branch -d newBranch (deletes branch)
```

Forking:

When its not your project

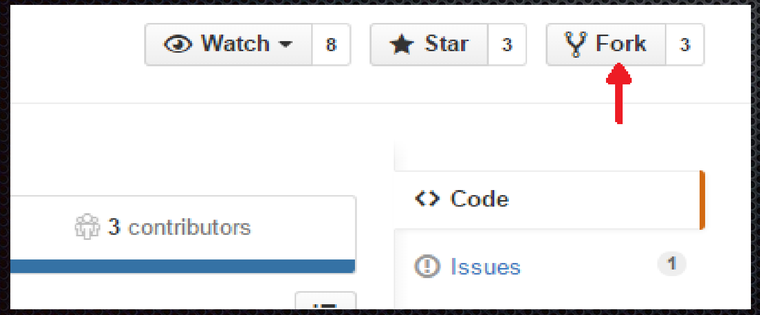
-A *fork* is a copy of a repository. Forking a repository allows you to freely experiment with changes without affecting the original project.

-Most commonly, forks are used to either propose changes to someone else's project or to use someone else's project as a starting point for your own idea.

-At the heart of open source is the idea that by sharing code, we can make better, more reliable software.

-A great example of using forks to propose changes, is for bug fixes. Rather than logging an issue for a bug you've found, you can:

- Fork the repository.
- Make the fix.
- Submit a *pull request* to the project owner.
  (If the project owner likes your work, they might pull your fix into the original repository!)

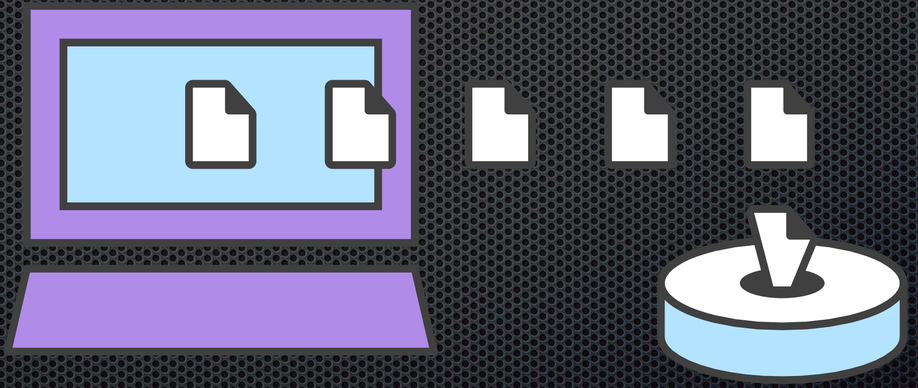


Git Stash

-Temporarily shelves (or *stashes*) changes you've made to your working copy so you can work on something else, and then come back and re-apply them later on.

-Stashing is handy if you need to quickly switch context and work on something else, but you're mid-way through a code change and aren't quite ready to commit.

-Note that the stash is local to your Git repository; stashes are not transferred to the server when you push.



Stashing your work

-The *`git stash`* command takes your uncommitted changes (both staged and unstaged), saves them away for later use, and then reverts them from your working copy. For example:

```
$ git status
On branch master
Changes to be committed:
new file: style.css
Changes not staged for commit:
modified: index.html
$ git stash
Saved working directory and index state WIP on master: 5002d47 our new homepage
HEAD is now at 5002d47 our new homepage
$ git status
On branch master nothing to commit, working tree clean
```

-At this point you're free to make changes, create new commits, switch branches, and perform any other Git operations; then come back and re-apply your stash when you're ready.

Re-applying stashes

-You can reapply previously stashed changes with *`git stash pop`*.

-Popping your stash REMOVES the changes from your stash and reapplies them to your working copy.

```
$ git status
On branch master
nothing to commit, working tree clean
$ git stash pop
On branch master
Changes to be committed:
new file: style.css
Changes not staged for commit:
modified: index.html
Dropped refs/stash@{0} (32b3aa1d185dfe6d57b3c3cc3b32cbf3e380cc6a)
```

-Alternatively, you can reapply the changes to your working copy AND keep them in your stash with *`git stash apply`*.

-This is useful if you want to apply the same stashed changes to multiple branches.

```
$ git stash apply
On branch master
Changes to be committed:
new file: style.css
Changes not staged for commit:
modified: index.html
```

Multiple stashes

-You aren't limited to a single stash. You can run *`git stash`* several times to create multiple stashes, and then use *`git stash list`* to view them.

-By default, stashes are identified simply as a "WIP" – work in progress – on top of the branch and commit that you created the stash from. After a while it can be difficult to remember what each stash contains:

```
$ git stash list
stash@{0}: WIP on master: 5002d47 our new homepage
stash@{1}: WIP on master: 5002d47 our new homepage
stash@{2}: WIP on master: 5002d47 our new homepage
```

-A much better practice would be to annotate your stashes with a descriptive message, using *`git stash save "message"`*

```
$ git stash save "add style to our site"
Saved working directory and index state On master: add style to our site
HEAD is now at 5002d47 our new homepage
$ git stash list
stash@{0}: On master: add style to our site
stash@{1}: WIP on master: 5002d47 our new homepage
stash@{2}: WIP on master: 5002d47 our new homepage
```

-By default, *`git stash pop`* will re-apply the most recent stash. You can choose which stash to apply by passing the identifier as the last argument.

```
$ git stash pop stash@{2}
```

-Finally, to clear your stash, run *`git stash clear`*.