Git Rekt

An Introduction to Using Git

Devon Bautista

Arizona State University

Spring 2018

What is Git?

- A **Version Control System** for tracking changes in a set of files among multiple people.
    - It doesn't just store files, it tracks changes in segments called *commits* ("versions").
    - Mess up? You can revert to *any* previous version.
    - Keeps records of changes (*diffs*) in files. → Version history stored locally in `.git` folder.

What is Git?

- A **Version Control System** for tracking changes in a set of files among multiple people.
  - It doesn't just store files, it tracks changes in segments called *commits* ("versions").
  - Mess up? You can revert to *any* previous version.
  - Keeps records of changes (*diffs*) in files. $\rightarrow$ Version history stored locally in `.git` folder.
- Created by Linus Torvalds in 2005 to manage Linux kernel development.

What is Git?

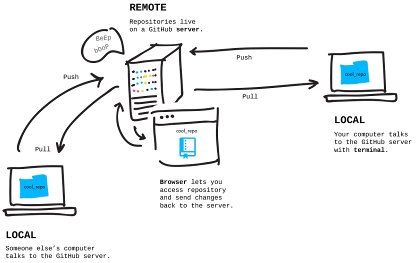
- A **Version Control System** for tracking changes in a set of files among multiple people.
  - It doesn't just store files, it tracks changes in segments called *commits* ("versions").
  - Mess up? You can revert to *any* previous version.
  - Keeps records of changes (*diffs*) in files. → Version history stored locally in `.git` folder.
- Created by Linus Torvalds in 2005 to manage Linux kernel development.
- Primarily a source code management system, but can manage *any* type of file.

What is Git?

- A **Version Control System** for tracking changes in a set of files among multiple people.
  - It doesn't just store files, it tracks changes in segments called *commits* ("versions").
  - Mess up? You can revert to *any* previous version.
  - Keeps records of changes (*diffs*) in files. $\rightarrow$ Version history stored locally in `.git` folder.
- Created by Linus Torvalds in 2005 to manage Linux kernel development.
- Primarily a source code management system, but can manage *any* type of file.
- Allows multiple people to remotely collaborate on a project.
  - See who made what changes.

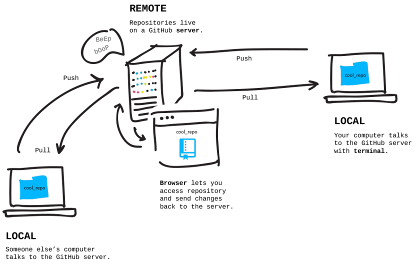
How Does Git Work? (Basics)

- Changes are stored in a repository that resides on a server, or *remote*.
    - e.g. GitHub, GitLab, Bitbucket etc.


REMOTE
Repositories live on a GitHub **server**.
BeEp bOoP
Push
Pull
Push
Pull
cool_repo
cool_repo
cool_repo
LOCAL
Your computer talks to the GitHub server with **terminal**.
Browser lets you access repository and send changes back to the server.
LOCAL
Someone else's computer talks to the GitHub server.

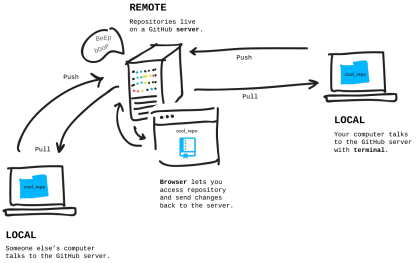
How Does Git Work? (Basics)

- Changes are stored in a repository that resides on a server, or *remote*.
    - e.g. GitHub, GitLab, Bitbucket etc.
- Collaborators `pull` changes from the *remote* to their *local* machine to edit.


REMOTE
Repositories live on a GitHub **server**.
BeEp bOoP
Push
Pull
Push
Pull
cool_repo
cool_repo
cool_repo
LOCAL
Your computer talks to the GitHub server with **terminal**.
Browser lets you access repository and send changes back to the server.
LOCAL
Someone else's computer talks to the GitHub server.

How Does Git Work? (Basics)

- Changes are stored in a repository that resides on a server, or *remote*.
  - e.g. GitHub, GitLab, Bitbucket etc.
- Collaborators `pull` changes from the *remote* to their *local* machine to edit.
- Collaborators then `push` their changes back to the *remote*.


REMOTE
Repositories live on a GitHub **server**.
BeEp bOoP
Push
Pull
Push
Pull
cool_repo
cool_repo
cool_repo
LOCAL
Your computer talks to the GitHub server with **terminal**.
Browser lets you access repository and send changes back to the server.
LOCAL
Someone else's computer talks to the GitHub server.

Let's "Git" Our Hands Dirty

A Working Example:

Making a Repository for Schoolwork

First, Let's Configure Git

Open the command prompt, terminal, or **Git Bash** and enter the following commands:

- `git config --global user.name 'GitHub Username'`
- `git config --global user.email 'GitHub Email'`

This will prevent you from having to type in your username every time you want to push to GitHub.

These commands set global Git configuration options for Git (you can do per-repository configuration without the `--global` flag).

Creating the Local Repository

Now, let's create a repository:

1. `mkdir` *`repository_name`*
   - Creates a new folder for your repository. You could use the file explorer as well.

Creating the Local Repository

Now, let's create a repository:

1. `mkdir repository_name`
   - Creates a new folder for your repository. You could use the file explorer as well.
2. `cd repository_name`
   - Changes your *working directory* to the newly created folder.

From now on, I will use `school-test` instead of *`repository_name`*.

Connecting the Repository with GitHub

1. Login to `https://github.com/`.
2. In the upper-right corner of any page, click $+$, and then click **New repository**.
3. Fill out the **Repository name** and **Description**.
4. *Un*check **Initialize this repository with a README**.
5. Click **Create repository**.

NOTE: GitHub offers free private repositories to students. Apply for the student pack:
`https://help.github.com/articles/applying-for-a-student-developer-pack/`
You can also use *Bitbucket* as your remote instead. They have free private repositories.

Connecting the Repository with GitHub (*continued*)

Go back to **Git Bash** or the command prompt:

1 `echo '# school-test' > README.md`

- Creates a *README.md* file with the contents `# school-test`.

Connecting the Repository with GitHub (*continued*)

Go back to **Git Bash** or the command prompt:

1. `echo '# school-test' > README.md`
   - Creates a *README.md* file with the contents `# school-test`.
2. `git init`
   - Initializes the current folder to a Git repository.

Connecting the Repository with GitHub (*continued*)

Go back to **Git Bash** or the command prompt:

1. `echo '# school-test' > README.md`
   - Creates a *README.md* file with the contents `# school-test`.
2. `git init`
   - Initializes the current folder to a Git repository.
3. `git add README.md` **OR** `git add .`
   - Adds the changes (creation of *README.md* file) to Git's *staging area*. The *staging area* is a file in `.git` that tells Git which changes will be used in the next commit. `git add` or `git rm` to add or remove changes from this file.

Connecting the Repository with GitHub (*continued*)

4. `git commit -m "Commit Message"`
    - *Commits* the changes in the staging area with the specified *`Commit Message`*. Think of this as creating a "snapshot" of your changes.

Connecting the Repository with GitHub (*continued*)

4. `git commit -m "Commit Message"`
   - *Commits* the changes in the staging area with the specified *Commit Message*. Think of this as creating a "snapshot" of your changes.
5. `git remote -v`
   - This checks which *remotes* we've added. Remember, a *remote* is where we push or changes to (e.g. GitHub). This should be empty. We will add one in the next step.

Connecting the Repository with GitHub (*continued*)

4. `git commit -m "Commit Message"`
   - *Commits* the changes in the staging area with the specified `Commit Message`. Think of this as creating a "snapshot" of your changes.
5. `git remote -v`
   - This checks which *remotes* we've added. Remember, a *remote* is where we push or changes to (e.g. GitHub). This should be empty. We will add one in the next step.
6. `git remote add origin repository_link`
   - To get your `repository_link`, go to your GitHub repository, click **Clone or Download** and use the link provided.
   - Do a `git remote -v` after this to verify the newly added remote.

Connecting the Repository with GitHub (*continued*)

NOTE: The above step uses the `https` protocol to push your changes, so you will have to type in your GitHub credentials every time you push. To avoid this, think about setting up and using an SSH key: https://help.github.com/articles/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent/#platform-windows

Connecting the Repository with GitHub (*continued*)

7. `git push -u origin master`
    - This pushes the *commits* ("snapshots") you've made to the *remote* you added `origin`), in this case, GitHub.

Connecting the Repository with GitHub (*continued*)

7. `git push -u origin master`
   - This pushes the *commits* ("snapshots") you've made to the *remote* you added `origin`), in this case, GitHub.
   - `origin` is the name of the remote repository where you want to publish you commits. We set this up previously using the `git remote add` command.

Connecting the Repository with GitHub (*continued*)

7. `git push -u origin master`
   - This pushes the *commits* ("snapshots") you've made to the *remote* you added `origin`), in this case, GitHub.
   - `origin` is the name of the remote repository where you want to publish you commits. We set this up previously using the `git remote add` command.
   - `master` is the *branch* the commit is pushed to. `master` is the default branch. We will talk later about branches when we discuss collaboration.

Connecting the Repository with GitHub (*continued*)

7. `git push -u origin master`
   - This pushes the *commits* ("snapshots") you've made to the *remote* you added `origin`), in this case, GitHub.
   - `origin` is the name of the remote repository where you want to publish you commits. We set this up previously using the `git remote add` command.
   - `master` is the *branch* the commit is pushed to. `master` is the default branch. We will talk later about branches when we discuss collaboration.

Before we move on, let's briefly look at cloning...

Cloning a Repository

- Another way to create a GitHub repository is to create it on GitHub and clone it to your machine.
- Cloning is also useful if you are not working on a computer with a local copy of the repository.

Cloning a Repository

- Another way to create a GitHub repository is to create it on GitHub and clone it to your machine.
- Cloning is also useful if you are not working on a computer with a local copy of the repository.

Try this:

1. Delete your repository folder on your system.

Cloning a Repository

- Another way to create a GitHub repository is to create it on GitHub and clone it to your machine.
- Cloning is also useful if you are not working on a computer with a local copy of the repository.

Try this:

1. Delete your repository folder on your system.
2. Remember your GitHub repository link? Copy it again...

Cloning a Repository

- Another way to create a GitHub repository is to create it on GitHub and clone it to your machine.
- Cloning is also useful if you are not working on a computer with a local copy of the repository.

Try this:

1. Delete your repository folder on your system.
2. Remember your GitHub repository link? Copy it again...
3. `git clone` *`repository_link`*

This copies your repository from GitHub, including *all* of your commit history! If you `cd` into the repository folder and type `git log`, you can verify this.

Making Changes

Let's actually use this repository now!

1. `cd` into your repo folder and do `git pull`.
   - This gets any changes that might have occurred on the remote and integrates them with your local changes. It's good habit to do this before making local changes so that no *merge conflicts* occur.
   - Since Git tries to automatically integrate changes, you want to prevent *merge conflicts* in which you would have to *manually* edit files to integrate changes. More on this in the collaboration section.
   - Not necessary for single collaborator.

Making Changes

Let's actually use this repository now!

1. `cd` into your repo folder and do `git pull`.
   - This gets any changes that might have occurred on the remote and integrates them with your local changes. It's good habit to do this before making local changes so that no *merge conflicts* occur.
   - Since Git tries to automatically integrate changes, you want to prevent *merge conflicts* in which you would have to *manually* edit files to integrate changes. More on this in the collaboration section.
   - Not necessary for single collaborator.
2. Add any school files and folders (under 100MB) you have to your repo folder.

Staging Changes

3. `git add .`
   - This adds changes in *all* files to the staging area.

TIP: Before each step, do a `git status` to see the state of your repository in each step.

Staging Changes

3. `git add .`
   - This adds changes in *all* files to the staging area.
4. `git diff HEAD`
   - See your changes before you commit.
   - You can view differences between any commits within the same branch or different branches, even between files!

TIP: Before each step, do a `git status` to see the state of your repository in each step.

Committing and Pushing Changes

Now, let's commit our changes and update our remote:

5. `git commit -m "Added school files/folders."`
   - Commits (creates "snapshot" of) changes in staging area with the specified commit message. If you do `git log`, you can see the new commit.

TIP: Before each step, do a `git status` to see the state of your repository in each step.

Committing and Pushing Changes

Now, let's commit our changes and update our remote:

5. `git commit -m "Added school files/folders."`
   - Commits (creates "snapshot" of) changes in staging area with the specified commit message. If you do `git log`, you can see the new commit.
6. `git push origin master`
   - This pushes our changes to our GitHub repository (`origin`) on the `master` branch.

TIP: Before each step, do a `git status` to see the state of your repository in each step.

Remember: The Big Three Commands

After `git pull`-ing and making changes, to put your changes on the remote (for **single**-collaborator repositories):

- `git add .`
- `git commit -m "Commit Message."`
- `git push origin master`

How Do I Revert if Something Goes Wrong?

1. Edit the *README.md* file and add some gibberish text.

How Do I Revert if Something Goes Wrong?

1. Edit the *README.md* file and add some gibberish text.
2. `git add README.md`
    - At this point, we can revert to our last commit by doing a `git reset HEAD --hard` Try this and then redo the mistake and re-add the change. (**NOTE:** Omitting the `--hard` will **not** revert *README.md*, just but unstage it.)

How Do I Revert if Something Goes Wrong?

1. Edit the *README.md* file and add some gibberish text.
2. `git add README.md`
   - At this point, we can revert to our last commit by doing a `git reset HEAD --hard` Try this and then redo the mistake and re-add the change. (**NOTE:** Omitting the `--hard` will **not** revert *README.md*, just but unstage it.)
3. `git commit -m "Added gibberish!"`
4. `git push origin master`

How Do I Revert if Something Goes Wrong?

1. Edit the *README.md* file and add some gibberish text.
2. `git add README.md`
   - At this point, we can revert to our last commit by doing a `git reset HEAD --hard` Try this and then redo the mistake and re-add the change. (**NOTE:** Omitting the `--hard` will **not** revert *README.md*, just but unstage it.)
3. `git commit -m "Added gibberish!"`
4. `git push origin master`
5. We've pushed a mistake... We now need to "rewrite history" to fix the change.
   - This can be a complex, but doable process. Just bear with me!

How to Revert a Change

1. `git log` to see commit history.

How to Revert a Change

1. `git log` to see commit history.
2. Look at the commit **before** the mistake commit. You'll see its identifier, which looks like a hash:

   `8628acb3205ce65300bd801ca3b88b6312c6a16d`

   Copy this and exit the log.

How to Revert a Change

1. `git log` to see commit history.
2. Look at the commit **before** the mistake commit. You'll see its identifier, which looks like a hash:

   ```
   8628acb3205ce65300bd801ca3b88b6312c6a16d
   ```

   Copy this and exit the log.
3. `git rebase -i` *`commit_ID`*
   - This "rewinds" `HEAD` to the commit *before* the problem commit so we can change the problem commit. An editor will appear for us to decide what to do with the problem commit. (`-i` for *interactive*.)

How to Revert a Change (*continued*)

4. You'll see a file open up like this:

```
pick 8628acb Added gibberish!

# Rebase ba1db00..8628acb onto ba1db00 (1 command)
#
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
# x, exec = run command (the rest of the line) using shell
# d, drop = remove commit
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

How to Revert a Change (*continued*)

4. You'll see a file open up like this:

```
pick 8628acb Added gibberish!

# Rebase ba1db00..8628acb onto ba1db00 (1 command)
#
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
# x, exec = run command (the rest of the line) using shell
# d, drop = remove commit
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

5. We want to delete this commit, so change the `pick` at the beginning to `drop`. Save the file and exit.

How to Revert a Change (*continued*)

6. `git push --force origin master` to send changes to GitHub.
   - We need `--force` so that Git doesn't think we are a commit behind (since we deleted the last commit).

How to Revert a Change (*continued*)

6. `git push --force origin master` to send changes to GitHub.
   - We need `--force` so that Git doesn't think we are a commit behind (since we deleted the last commit).

- This is a general process for fixing mistakes and works with multiple commits.

How to Revert a Change (*continued*)

6. `git push --force origin master` to send changes to GitHub.
   - We need `--force` so that Git doesn't think we are a commit behind (since we deleted the last commit).

- This is a general process for fixing mistakes and works with multiple commits.
- For a trivial example like this, the editor session is unnecessary. However, it *is* necessary for larger changes that may involve more than one or older commits.

Collaboration in Git

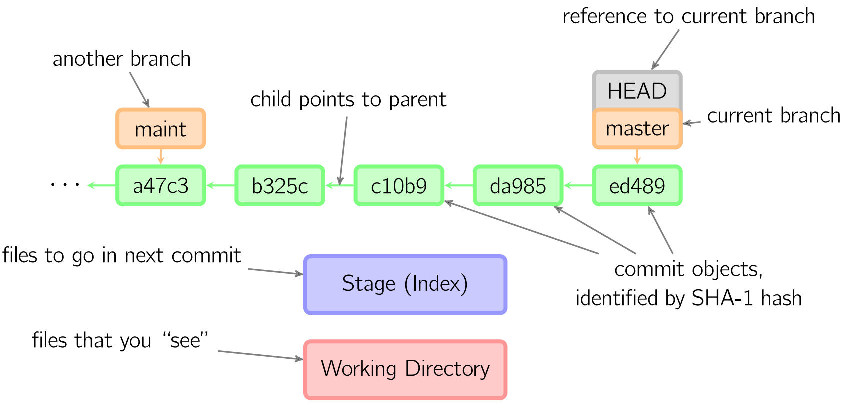
Using Git to Collaborate

Key Concepts

What are Branches?

- A **branch** is simply a series of commits ("snapshots") linked together.
- Each commit references (or "points") to a commit before it.
- The default branch is `master`.
- Current branch referenced by **HEAD**.

A Typical Git Project

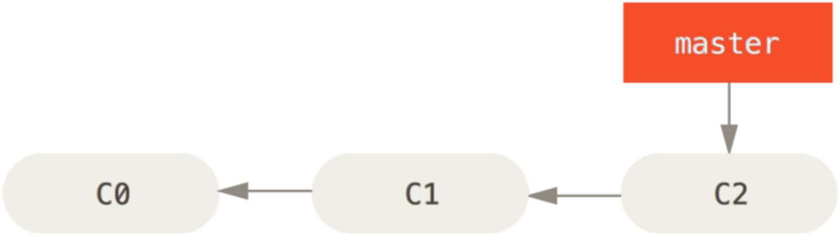


Branching Off of Master

- The start of a branch points to a specific commit.
- To make changes to your project (e.g. implement a feature), make a new branch (e.g. a `features` branch).

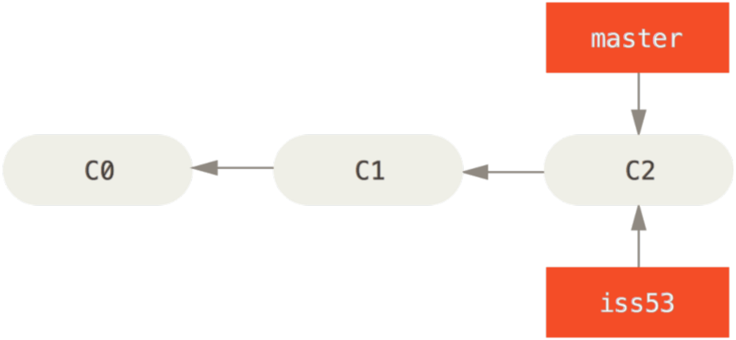
Branching Example

Say we have this setup (**HEAD** is pointing to `master`):



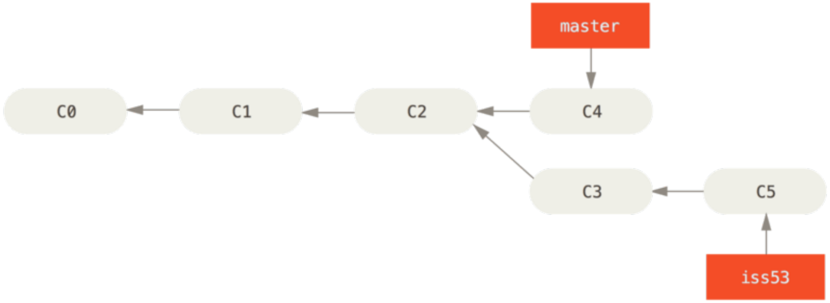
Branching Example

Create (”checkout”) branch `iss53` using the commit pointed to by **HEAD** (`master` at `C2`). **HEAD** is now at `iss53` (still `C2`).



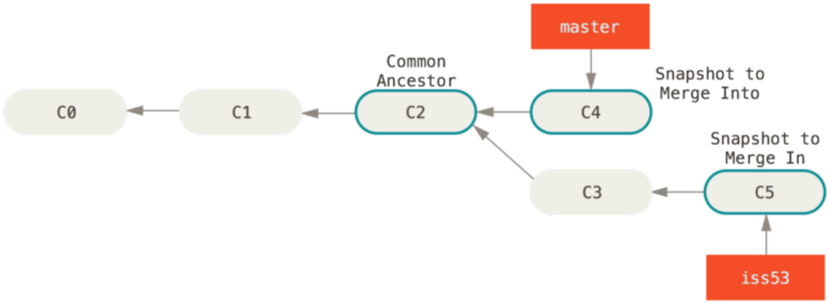
Branching Example

Make some changes on `iss53`, then add, commit, and push them.



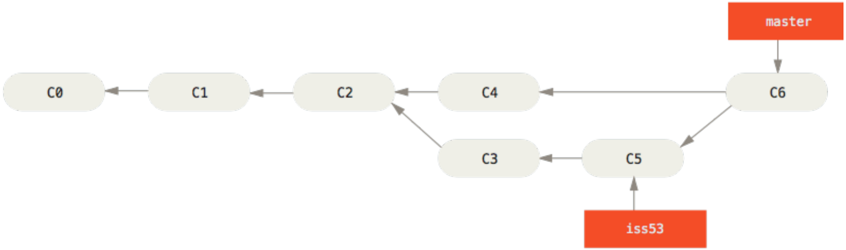
Branching Example

When we finish our feature, we want to merge them back to `master`, the "OK" branch. This occurs through a **3-way merge** between the commits outlined in teal.



Branching Example

After merging (if no merge conflicts arise), Git will automatically merge the changes:



C6 is the new commit resulting from the merge. Yes, Git treats a merge as a separate commit.

Basic Merge Conflicts

If Git is unable to automatically merge, a *merge conflict* will arise. In this case, you will have to go in and manually edit the changes before trying to merge again.

Basic Merge Conflicts

- <<<<< to ===== is one branch's changes (where **HEAD** is).
- ===== to >>>>> is iss53's changes.

```
<<<<<<< HEAD:index.html
<div id="footer">contact : email.
   support@github.com</div>
=======
<div id="footer">
 please contact us at support@github.com
</div>
>>>>>>> iss53:index.html
```

Figure: Example file in a merge conflict.

Basic Merge Conflicts

Once changes are resolved, mark each file as resolved using `git add <file>`, then retry merge.

Collaboration: Multiple People on a Project

Another Working Example:

A Repository for a School Project

Join Sample Repository

1. Login to GitHub.
2. Tell me your username so I can add you to my repo as a collaborator.
3. Accept the invitation from your GitHub account.
4. `git clone https://github.com/Deyyvon/collab-demo.git`
5. `cd collab-demo`

Create Your Own Branch

1. `git fetch`
   - This will fetch any changes that might have occurred on the remote (GitHub) and updates your local copy.

Create Your Own Branch

1. `git fetch`
   - This will fetch any changes that might have occurred on the remote (GitHub) and updates your local copy.
2. `git checkout -b` *`your_branch_name`*
   - This creates a branch called *`your_branch_name`* locally, taken from the latest commit on `master`.

Create Your Own Branch

1. `git fetch`
   - This will fetch any changes that might have occurred on the remote (GitHub) and updates your local copy.
2. `git checkout -b` *`your_branch_name`*
   - This creates a branch called *`your_branch_name`* locally, taken from the latest commit on `master`.
3. Create a plaintext file with your first name and the contents `"This is` *`your_name`*`'s branch!"`.
   - **HINT**: You can use an editor or you can use `echo "This is` *`your_name`*`'s branch!" >` *`your_name`*`.txt`

Make Some Changes On Your Branch

4. `git add .`
   - Stage the new file you created.

Make Some Changes On Your Branch

4. `git add .`
   - Stage the new file you created.
5. `git commit -m "Added your_name.txt"`
   - Commit your changes locally.

Make Some Changes On Your Branch

4. `git add .`
   - Stage the new file you created.
5. `git commit -m "Added *your_name*.txt"`
   - Commit your changes locally.
6. `git push -u origin *your_branch_name*`
   - Pushes your new branch along with its changes to the remote on GitHub.

Next, we will merge changes on your branch with those on `master`.

Merge Branch with Master

1. `git fetch`
2. `git status`
3. `git log`
   - Before any merging occurs, `fetch` any new remote changes, check the `status` of the repo, and check the `log` to see the commit history. Remember, you are integrating your changes with someone else's so you want to be up-to-date and confident!

Merge Branch with Master

1. `git fetch`
2. `git status`
3. `git log`
   - Before any merging occurs, `fetch` any new remote changes, check the `status` of the repo, and check the `log` to see the commit history. Remember, you are integrating your changes with someone else's so you want to be up-to-date and confident!
4. `git checkout master`
5. `git merge` *`your_branch_name`*
   - Merge *`your_branch_name`* into `master`. First, we switch to `master`, then we merge *`your_branch_name`* into it.
   - Choose a commit message for the reason for the merge.
   - If Git can automatically merge, the process will go smoothly.

Merge Branch with Master (*continued*)

6. `git push origin master`
   - Push your merge commit to GitHub!

Merge Branch with Master (*continued*)

6. `git push origin master`
   - Push your merge commit to GitHub!
7. `git push -d origin` *`your_branch_name`*
8. `git branch -d` *`your_branch_name`*
   - **OPTIONAL:** Delete your branch from the remote and locally since you successfully added your feature to `master`.

If we have time, we can deal with a merge conflict...

Merge Branch with Master (*continued*)

6. `git push origin master`
   - Push your merge commit to GitHub!
7. `git push -d origin` *`your_branch_name`*
8. `git branch -d` *`your_branch_name`*
   - **OPTIONAL:** Delete your branch from the remote and locally since you successfully added your feature to `master`.

If we have time, we can deal with a merge conflict...

1. `git pull`

Merge Branch with Master (*continued*)

6. `git push origin master`
   - Push your merge commit to GitHub!
7. `git push -d origin` *`your_branch_name`*
8. `git branch -d` *`your_branch_name`*
   - **OPTIONAL:** Delete your branch from the remote and locally since you successfully added your feature to `master`.

If we have time, we can deal with a merge conflict...

1. `git pull`

(Here, I will edit *names* so that your changes will cause a merge conflict.)

Merge Conflict: Editing the Same File

2. `git checkout -b` *`your_branch_name`*
3. `echo "`*`your_name`*`" >>` *`names`*
4. `git add names`
5. `git commit -m "`*`Commit message`*`"`
6. `git push origin` *`your_branch`*
7. `git checkout master`
8. `git pull`
9. `git merge` *`your_branch_name`*
   - Merge fails because there are conflicting errors to the same part of the file!
10. `git status`
    - We need to edit the red file (*names*) to resolve the merge conflict.

Dealing with a Merge Conflict (*continued*)

11. Edit the *names* file so that both names are added (I can help).
12. `git add names`
    - Adding marks the conflicted file as resolved.
13. `git commit`
14. `git push origin master`

Done!

Things We Didn't Cover (Yet)

- Pull/Merge Requests
- More Git commands (There are a lot...)
- The "nitty gritty" of Git (there's a lot of cool stuff under the hood)

Resources

Stuck or need help?

- `git <command> --help`
- https://git-scm.com/docs
- Google!

Resources

Stuck or need help?

- `git <command> --help`
- https://git-scm.com/docs
- Google!

The command line scares me... :(

GitHub Desktop: Works only with GitHub, but popular.
- https://desktop.github.com/

Git Kraken: Works with any remote; written using Electron (Discord-like interface).
- https://www.gitkraken.com/download