Gitting Good at GitHub

An interactive introduction to working with Git and GitHub

Alex Antonison

linkedin.com/in/alexantonison

github.com/alex-antonison

alex-antonison.com

Agenda

- Module 1 - Introduction to Version Control and Git
- Module 2 - Basic Git Operations
- Module 3 - Branching and Merging
- Module 4 - Collaboration on GitHub
- Module 5 - Simulated Project
- Module 6 - Advanced Git and GitHub Features

Module 1 - Introduction to Version Control and Git

- What is Version Control
  - What are the benefits of Version Control?
- What is Git
  - Key Git Concepts
- What is GitHub
  - Setting up Git with GitHub

What is Version Control?

- Version control is a tool that allows you to track changes to files over time.
- This is most effective when working with text files but also works with other files.

What are the benefits of Version Control?

- **Track Changes.** You can see who changed what to files over time.
- **File History.** Ability to revert back to previous versions of a file.
- **Collaboration.** Supports collaborating in files.

What is Git?

- Git is a distributed version control system used to track changes.
  - There are centralized frameworks such as Team Foundation Server (Microsoft) and Subversion (SVN) (Apache)
- Changes are tracked locally on your computer and then synchronized with a remote repository (like GitHub).
- Each change is saved as a snapshot, not just differences.
- Enables offline work and syncing changes when you're ready.

Key Git Concepts

- **Repository (repo).** A directory where Git stores the history of your project.
- **Commit.** A snapshot of your changes, along with a message describing what was done.
- **Branch.** A branch lets you work on changes separately from the main code.
- **Merge.** Combines commits from one branch into another.
- **Tracks Files.** Git tracks files and their file paths, not directories.

What is GitHub?

- An online platform to store, manage, and track changes to code using Git.
- A **communication** platform.
- **GitHub Actions.** for running and deploying code.
- **GitHub Pages.** for blogs.
- Productivity tools like Issue tracking, project management, and a wiki.

Setting up Git with GitHub

- Install GitHub Desktop at https://github.com/apps/desktop
- Authenticate with GitHub.
- Let’s git started!

Module 2 - Basic Git Operations

- Getting started working with Repositories
- Ignoring Files
- Staging and Committing Changes
- When to Commit
- Undoing Changes
- Removing Files

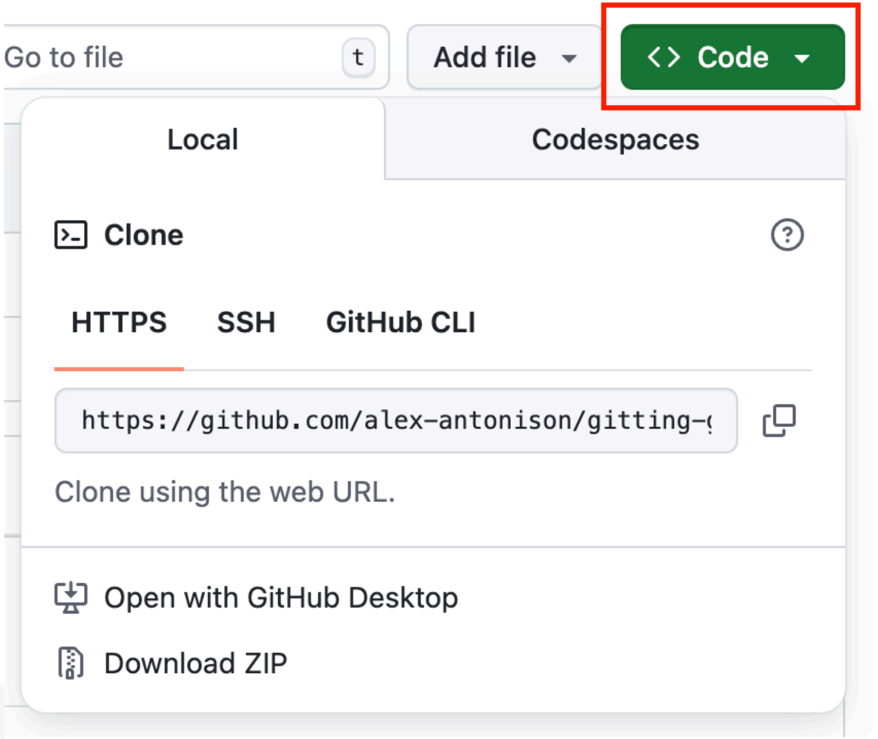
Getting started working with Repositories

1. Creating a repository
2. Forking a repository

github.com/alex-antonison/gitting-good-at-github

Once a repository has been created or forked, you can then clone a repository with

```
git clone {repository link}
```



https://github.com/alex-antonison/gitting-(

Clone using the web URL.

Ignoring Files

- A `.gitignore` file is used to ignore files that you do not want to manage in source control. Some examples are:
    - Files containing secrets (passwords, API keys, etc.)
    - Locally installed package files like `.venv/`
    - Large data files (aka 2 GB csv files)

Ignoring Files (Activity)

1. Create a directory `new_folder`
2. Create a file `new_file.txt` in the `new_folder` directory.
3. Add `new_folder/` to the `.gitignore` file in the root project.
4. To the root project, add a file called `new_file.csv`
5. Add `*.csv` to the `.gitignore` file.

Staging and Committing Changes

- You can either add all files with `git add -A` or `git add path/to/filename`
  - Tools like GitHub Desktop and VS Code’s Source Control Panel are helpful to see all files
- Once a file is staged, you can then `commit` the changes with
  - `git commit -m"{insert message here}"`
- To commit all tracked files that have been modified, you can use the `-a` argument:
  - `git commit -am"{insert message here}"`

Committing Changes (Activity)

1. Make changes to the `exercises/existing_file.txt`
2. Create a file in `exercises` directory called `new_file.txt` and add text to it
3. `git commit -am "{insert your descriptive message here}"`
4. `git status`
5. `git add exercises/new_file.txt`
6. `git status`
7. `git commit -m "{insert your descriptive message here}"`
8. Check **Commit History.** You can use `git log` but GitHub Desktop or VS Code is easier

When to Commit?

- When do you commit?
  - Committing too often leads to noisy commits
  - Committing too infrequently makes it hard to find changes
- *The best time to commit is when you have completed a "thought"*
  - This could be when you finish a function, update some business logic, etc.

Undoing Changes

- `git reset HEAD~1`
  - This simply undoes the last commit with files intact
- `git checkout origin/main path/to/filename`
  - This will reset the file to what is in the main branch

Undoing Changes (Activity)

1. To undo your last commit, run `git reset HEAD~1`

You can explore other ways of changing history, but suggest using GitHub Desktop.

- git commit --amend
- git revert

Removing Tracked Files

- In the event a file gets added that you want to stop tracking, you can use the `git rm` command:
- If you want to untrack and delete the file, you can do:
  - `git rm path/to/file`
- If you want to just stop tracking it in git you can do:
  - `git rm --cached path/to/filename`
  - Add the file to your `.gitignore`
- If you want to untrack an entire directory:
  - `git rm -r path/to/directory`
- Last, you need to commit the removal change using `git commit`

Removing Tracked Files (Activity)

1. Remove `new_folder` from `.gitignore`
2. `git add new_folder/new_file.txt` file.
3. `git commit -m "New file commit message"`.
4. Remove the whole folder `git rm -r new_folder/`
5. Commit the removal `git commit -m "Remove new_folder"`

Removing Files from History

- In the event you discover a secret has been committed, it is best practices to cycle the secret (e.g. change the password, regenerate the API key, etc.) so that it is no longer valid. You can also follow this guide for removing from the file with secrets history
- It is complex and challenging to modify git history (this is a feature, not a bug).
- Be very careful to not include secrets in your branch and especially don't merge a branch with secrets into the main branch.

Module 3 - Branching and Merging

- What are Branches
- Working with Branches
- Merging Branches
- Merge Conflicts

What are Branches?

- A branch lets you work on changes separately from the main code.
  - For individual projects, working in `main` is "okay"
  - It is a good habit to always work in branches
  - For team based projects, always need to work in branches

Working with Branches

- Create a branch and then checkout that branch:
  - Create a branch `git branch {insert-descriptive-branch-name}`
  - To checkout a `git checkout {insert-descriptive-branch-name}` or `git switch {insert-descriptive-branch-name}`
- Create and checkout a branch:
  - `git checkout -b {insert-descriptive-branch-name}`
- To delete a branch
  - `git checkout main`
  - `git branch -D {insert-descriptive-branch-name}`

Working with Branches (Activity)

1. Create a branch `git checkout -b my-new-branch`
2. Delete the branch:
    1. `git checkout main`
    2. `git branch -D my-new-branch`

Merging Branches

- To merge a branch locally, you do `git merge {insert-branch-name}`
- It is common to need to merge `main` into current branch because:
  - In your Pull Request, you have merge conflicts you need to address
  - Code has been merged into main that you need for your work
- To merge main into your current branch
  - `git checkout main` (swap to main)
  - `git pull` (update main branch)
  - `git checkout {insert-branch-name}` (swap back to your branch)
  - `git merge main` (merge main into your branch)

Merging Branches (Activity)

1. Create a branch `git branch add-feature`
2. Create a file `branch_test_file.txt` and add it
    1. `git add branch_test_file.txt`
    2. `git commit -m "Add branch test file"`
3. `git checkout add-feature` (Swap to add-feature branch)
4. Merge main into `add-feature` branch
    1. `git merge main`

Merge Conflicts

- Occurs when changes in two branches **affect the same part of a file** and Git can’t automatically decide which change to keep, requiring manual resolution.

```
<<<<<<< HEAD
This is the version from the main branch.
=======
This is the version from the test-sample-branch.
>>>>>>> test-sample-branch
```

Merge Conflict (Activity)

1. `git checkout main` (swap back to main)
2. Make a change to the first line in `exercises/existing_file.txt`
3. `git commit -am"Commit message for existing_file.txt"`
4. `git checkout add-feature` (swap back to add-feature)
5. Make a different change to the first line in `exercises/existing_file.txt`
6. `git commit -am"Commit message for existing_file.txt"`
7. Run `git merge main`
8. You can use the built in VS Code or GitHub Desktop to resolve the conflict.

Module 4 - Collaboration on GitHub

- Branching Strategies
- Working with a Ticket
- Creating Pull Requests
- Reviewing Pull Requests
- Merge Types

Branching Strategies

- **Git Flow.** A branching model that uses feature, develop, release, and hotfix branches to manage complex release cycles with multiple environments.
- **Trunk-Based Development.** A streamlined approach where developers work in short-lived branches to enable continuous integration and rapid delivery.

Working with a Ticket

- Working with tickets is a common practice to track tasks, bugs, or features.
- *If a ticket does not exist,* while there are some circumstances where you may not need a ticket, it is generally a good practice to create one even for ad hoc requests or small changes.
  - Creating a ticket is an opportunity to take a step back and think about the work you are about to do, which can help clarify the goals and the scope of changes or an ad hoc data request.
- Tickets help provide context and create a record of the work being done.
- Tickets can be used to show how a team provides value to the organization.

Creating Pull Requests

- To create a successful Pull Request, it should convey the following:
  - **Context.** Why are these changes being made to the code base?
  - **Description.** You should include a summary of the changes being made.
  - **Tests + Documentation.** You should include any required tests or documentation.
  - **Pull Request Template.** Streamlines organizing Pull Request descriptions.
  - **Draft Pull Request.** Create a "draft" pull request indicating a PR is still in progress.

When to Create a Pull Request

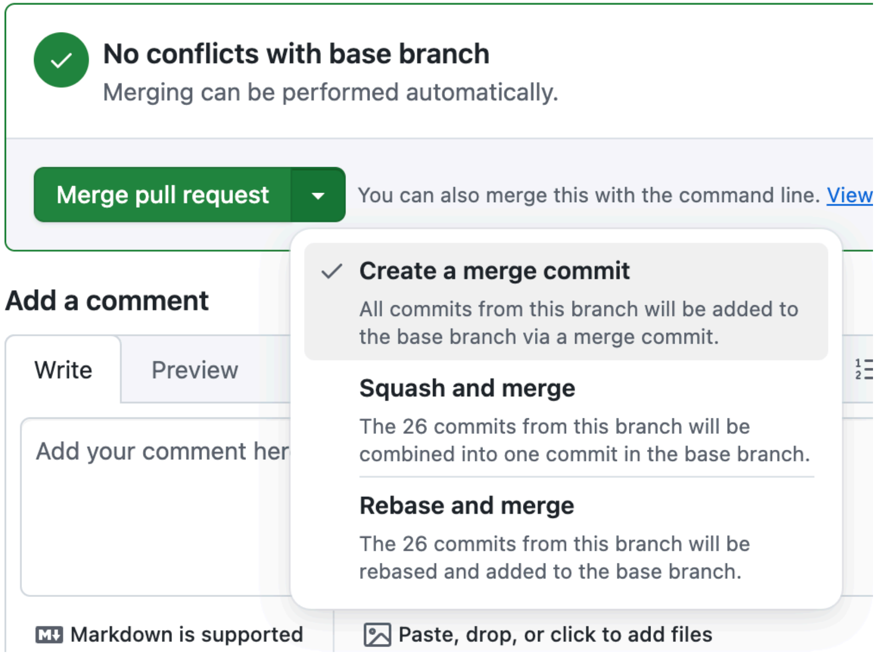
- When you first start working on a feature, you can create a **Draft Pull Request** which allows you to share your work in progress with others.
    - This is optional, but it can be helpful to get early feedback or to signal that you are working on something.
- When you are ready to merge your changes, you can create a **Pull Request** or mark your **Draft Pull Request** ready for review.
    - This is the formal request for others to review and approve your changes before they are merged.

Reviewing Pull Requests

- When reviewing a Pull Request, you can do a couple of things:
    - **In-line Code Comments.** Making in-line comments helps direct someone directly to a bit of code.
    - **Suggested Changes.** Add a suggested change to help improve a Pull Request.
    - **Pull Request Comments.** These can be helpful if there is a high level comment about a Pull Request.

Merge Types

- Merge
- Squash and merge (squerge)
- Rebase and merge


No conflicts with base branch
Merging can be performed automatically.
Merge pull request
You can also merge this with the command line. View
Add a comment
Write
Preview
Add your comment her
✓ Create a merge commit
All commits from this branch will be added to the base branch via a merge commit.
Squash and merge
The 26 commits from this branch will be combined into one commit in the base branch.
Rebase and merge
The 26 commits from this branch will be rebased and added to the base branch.
Markdown is supported
Paste, drop, or click to add files

Module 5 - Simulated Project

This is a sample project for testing working with GitHub

https://github.com/alex-antonison/git-r-done-enterprise-2025-07-14

Module 6 - Advanced Git and GitHub Features

- pre-commit
- GitHub Actions
- Rebasing

pre-commit

- pre-commit is a framework for managing and running automated Git hooks to catch and fix issues (like formatting, linting, or security checks) before code is committed, ensuring consistent code quality across teams.
- Some common pre-commit hooks are the following:
  - SQL - sqlfluff pre-commit
  - Python - ruff pre-commit
  - dbt Osmosis - docs pre-commit
  - git secrets checker

GitHub Actions

- GitHub Actions is a Continuous Integration/Continuous Deployment (CI/CD) automation tool built into GitHub that lets you automatically build, test, and deploy code based on events like pushes, pull requests, or schedule triggers.
- While GitHub Actions was built for CI/CD, it can also be used to run dbt projects as at its core, it is no different than running a serverless server. This can be done on a schedule or it can be done manually (via a workflow dispatch).

Git Rebasing

- Git Rebasing is a Git operation that moves or combines a sequence of commits to a new base commit, creating a cleaner, linear project history by integrating changes without merge commits.
- While git rebasing can create a cleaner history, it is also a destructive in that you are merging multiple commits together.
- You should avoid doing `git rebase` unless you are in a team environment where that is their standard approach.