Git and GitHub

A Comprehensive Primer

Requirements

1. Internet access and a web browser
2. Permission to install Git
3. Basic understanding of HTML
4. Basic understanding Terminal, Command Prompt, PowerShell, or another command console/line

Table of Contents

Why use Git? **2**
What is it? 2
The Scenario 2
Benefits of Using Git 2
Git Installation **3**
Checking for a Current Installation 3
Installing Git 3
How Git Works **4**
Repositories (Repos) 4
Commits 4
Creating a Repository **5**
Repo Status and Changes **6**
Staging Files **7**
An Extra Layer of Security 9
Making Commits **10**
Undoing Things **12**
Checkout Commit 12
Revert Commit 14
Reset Commit 15
Branches **16**
Creating a Branch 16
Developing a Feature on a Branch 17
Deleting a Branch 17
Developing Two Features Simultaneously 18
Feature One 18
Feature Two 19
Merging Branches and Resolving Conflicts **21**
Fast-Forward and Recursive Merges 21
Merging with Conflicts 21
Syncing Remote Repos **24**
Cloning Remotes 24
Creating Remotes 25
Basic Git Workflow 25
Pushing to Remotes 26
Pulling from Remotes 26
Credits / Information **28**

Why use Git?

What is it?

1. Git is a “distributed version control system”
    a. A system that records changes to files over time
    b. Specific versions of files can be recalled at any given moment
    c. Many can collaborate on the same project, tracking changes from others and submitting their own

The Scenario

A client hires you, provides a design, and you create it. Then, a link is given to the client for proofing, and they want changes. So, the development goes through multiple stages of re-design, until, at some point they decide, “You know what? I think we should just use the original design.”

This might have been a nightmare-of-a-request, but fortunately with Git, it is extremely easy to revert the project to the completed original design.

Benefits of Using Git

- Mark milestones of project development to maintain revision history
- Rewind to any revision instantly
- Work on new features without messing up the code base
- Collaboration with other developers is easier
- Wide ranging implementation and support
    - GUIs[1]: GitHub, Tower, SourceTree, VS Code

[1] Graphical User Interfaces

Git Installation

Checking for a Current Installation

If you're using macOS, it's likely that your machine already includes Git. This can be checked by running the following command from the Terminal. This may also be done from the Command Prompt on Windows.[2]

```
git --version
# git version 2.23.0
```

If this returns "git version" and a number, Git is installed.

Installing Git

You can install Git directly, but since Git by itself is only a command-line tool. It is most helpful to install a GUI wrapper for new users. Fortunately, most GUIs come with Git included in their software. GitHub Desktop and SourceTree are great cross-platform options. Tower is my preferred solution for macOS, however it is not free and requires a license.

To install Git without a GUI, follow the directions at https://git-scm.com/.

Once Git has been installed and the `git --version` command runs successfully, there are few options that need to be configured so Git will know who is requesting and making changes to projects.

The following commands will set the name and email of the user:

```
git config --global user.name "roydukkey"
git config --global user.email contact@changelog.me
```

If you ever want to recall the user name and email, these commands will return that information:

```
git config user.name
# roydukkey
git config user.email
# contact@changelog.me
```

[2] Some versions of macOS will prompt for the installation of command line developer tools. It is fine to accept this installation, however no GUI will be installed during the process.

How Git Works

Repositories (Repos)

- A repo is a container for a project you want to track with Git
- Can have many different repos for many different projects on your computer
- The contents are automatically tracked by Git

Any folder which contains a ".git" sub-folder is almost certainly a Git repository.

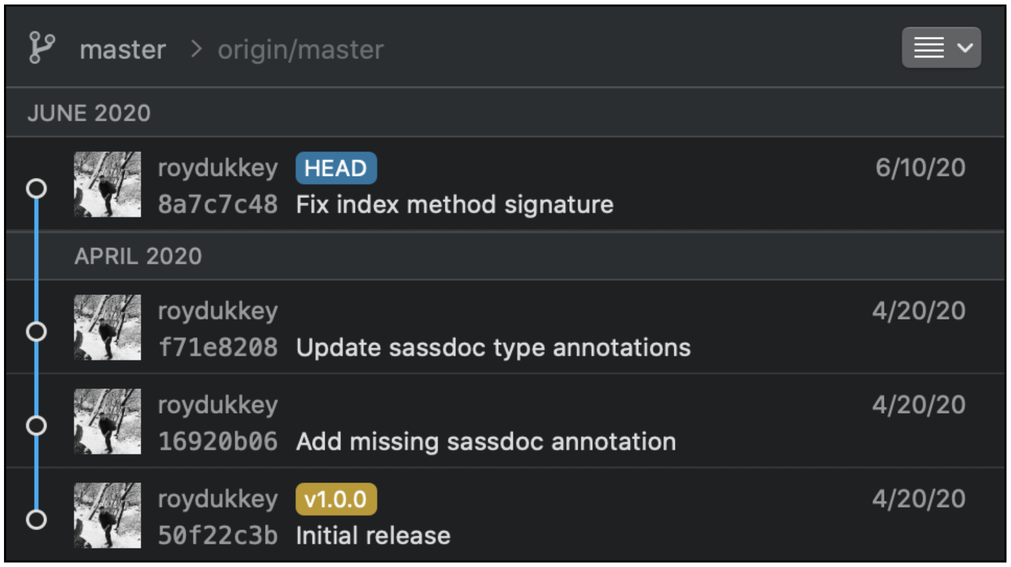
Commits

- Like save points
- Are chronological
- New commits are changes upon a previous commit

Eventually, as changes are finalised, new code will need to be saved to the repository. These are commits. In regard to individual files, they are considered having one of three states during the process of submitting a commit: modified, staged, and committed.

1. A modified file is one that has new changes since the prior commit.
2. A staged file is a modified file which has been verified and marked ready for commit.
3. A committed file is a staged file which has been declared finished for the current sequence of modification.

GUIs will display a commit history, which often looks similar to this:



```
JUNE 2020
roydukkey HEAD                                    6/10/20
8a7c7c48 Fix index method signature
APRIL 2020
roydukkey                                         4/20/20
f71e8208 Update sassdoc type annotations
roydukkey                                         4/20/20
16920b06 Add missing sassdoc annotation
roydukkey v1.0.0                                  4/20/20
50f22c3b Initial release
```

Creating a Repository

A directory is not a functioning Git repository until it has been initialised with one of the commands provided by Git.

1. Create an empty directory
2. Navigation to the new directory from the command line
3. Run the `git init` command

```
git init
# Initialized empty Git repository in ~/Desktop/git-primer/.git/
```

Once this command has executed, you will notice a new folder in the empty directory you created in step (1). This '.git' folder contains all the advanced, behind-the-scenes workers which manage the repository. Nothing in this folder is intended to be modified directly, except for very advanced Git functionality.

Repo Status and Changes

> Continue to use the repo you've created so far, or download the starting point for this section: 1 - Repo Status and Changes.zip.

Git is all about tracking changes, and provides the `git status` command to show them.

```
git status
# On branch master
#
# No commits yet
#
# nothing to commit (create/copy files and use "git add" to track)
```

Create an empty 'index.html' file in the root of the repository, then run the same command again.

```
git status
# On branch master
#
# No commits yet
#
# Untracked files:
#    (use "git add <file>..." to include in what will be committed)
#       index.html
#
# nothing added to commit but untracked files present (use "git add" to track)
```

The command lists the 'index.html' file as "untracked".[3] This means that the file is completely new, having never been previously committed to the repository.

[3] An untracked file is a specific designation for a type of modified file. See How Git Works \ Commits.

Staging Files

> Continue to use the repo you've created so far, or download the starting point for this section: 2 - Staging Files.zip.

In order to save changes to the repo, modified files need to be staged. This can be done with the `git add <file>` command.

```
git add index.html

git status
# On branch master
#
# No commits yet
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#      new file:   index.html
```

After executing the `git status` command again, you'll notice the response states that 'index.html' is changed and ready to be committed. Now this file is considered a file staged with changes.

Add the following content to the 'index.html' file:

```
<!doctype html>
<html lang="en-US">

<head>
 <meta charset="utf-8">
 <title></title>
</head>

<body>
 <p>Hello world! This is an empty HTML document.</p>
</body>

</html>
```

Run the `git status` command again. Notice ‘index.html’ is listed as staged and modified.

```
git status
# On branch master
#
# No commits yet
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#      new file:   index.html
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git restore <file>..." to discard changes in working directory)
#      modified:   index.html
```

You’ve learned that the `git add <file>` command will stage a file, there is also a command that will unstage changes. The `git rm --cached <file>` command will do just that.

```
git rm --cache index.html
# error: the following file has staged content different from both the
# file and the HEAD:
#     index.html
# (use -f to force removal)
```

Oops, that command failed. That’s because other changes were made to the same file which are not saved. The response describing the error states that you can use the `-f` flag with the command to force the removal, but there is another way to clear things up.

In the previous responses from the command line, you might have noticed another command which will discard changes. The `git restore <file>` command will discard unstaged changes on the specified file.

Run a couple of commands and see what happens. First restore `index.html`, and check the status. Then unstage the changes to `index.html`, and check the status one more time.

```
git restore index.html
git status
# On branch master
#
# No commits yet
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#       new file:   index.html

git rm --cache index.html
# rm 'index.html'

git status
# On branch master
#
# No commits yet
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#       index.html
#
# nothing added to commit but untracked files present (use "git add" to track)
```

The ‘index.html’ file will be unstaged and, if opened in an editor, will be empty just as it was when it was first created.

An Extra Layer of Security

You might be wondering, “Why is staging files even a thing in Git? What’s the point of the extra step?” The staging concept helps avoid committing unintended changes to the repo. Every change must first be verified and added to the staging area, before they are considered acceptable for committal.

Furthermore, it is common to make multiple unrelated changes to a repo simultaneously. For example, you might be tasked with fixing one bug, however as you develop a fix another bug is found. You immediately know the solution, so you fix the new bug in order to avoid forgetting about it.

Now two bug fixes exist on the repo. They should not be committed together. You stage the first bug-fix, comit it, stage the second fix, and finally commit that fix as well.

Making Commits

> Continue to use the repo you've created so far, or download the starting point for this section: 3 - Making Commits.zip.

When making a commit you are telling Git to save the staged changes to the repo. This makes the changes permanent, requiring a future commit to make further changes.

So, previously you have created a file and staged some changes. Go ahead and use what you have learned to add the same content from the previous section to the 'index.html' file and stage the changes.

The `git status` command should look like this:

```
git status
# On branch master
#
# No commits yet
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#      new file:   index.html
```

Now that there are staged changes the `git commit -m` command is used to save those changes. Notice the `-m` flag; this allows a commit message to be specified with the command. This message should be descriptive of the changes, allowing others who (or even yourself months later) review the repo's history to better understand the work that has been done, without reviewing the actual code.

```
git commit -m "Add an index w/basic content"
# [master (root-commit) 7cb1e23] Add an index w/basic content
#  1 file changed, 13 insertions(+)
#  create mode 100644 index.html
```

There are a couple import pieces of information in the response from the command. First, is the short ID of the commit, which in this case is `7cb1e23`. Second, it shows the number of files changed, the number of "insertions" (often accompanied by "deletions"), which is loosely relative to the number of lines changed. Last, the final line indicates the creation of a file.

Next, add the text, “`Git and Github Tutorial`”, as the `<title>` of ‘index.html’. Commit the new change.

```
git commit -m "Add a title to the index"
# [master 808872e] Add a title to the index
#  1 file changed, 1 insertion(+), 1 deletion(-)
```

Humm… there’s a “deletion”, but nothing was deleted by the last commit. Git doesn’t recognise just the text that’s been added. It looks at entire lines as a whole, therefore according to Git, the old line was deleted and a new line inserted.

There are two commits in the commit history. The history is seen by using the `git log` command.

```
git log
# commit 808872eb2ef5f17e3764b9a9f3b3b12b3cbf3e0b (HEAD -> master)
# Author: roydukkey <contact@changelog.me>
# Date:   Tue Aug 18 14:28:53 2020 -0400
#
#     Add a title to the index
#
# commit 7cb1e23a98cdd9febb19ce99246125f998815574
# Author: roydukkey <contact@changelog.me>
# Date:   Tue Aug 18 14:27:36 2020 -0400
#
#     Add an index w/basic content

git log --oneline
# 808872e (HEAD -> master) Add a title to the index
# 7cb1e23 Add an index w/basic content
```

Undoing Things

> Before continuing, download the starting point for this section: 4 - Undoing Things.zip.

When it comes to Git, there are three ways of undoing previously committed changes.

1. Checkout Commit – not dangerous
2. Revert Commit – possibly dangerous
3. Reset Commit – most likely dangerous

The checkout commit will turn back the repository to the state of a previous commit. However, for as long as the checkout commit is enabled, the repository will essentially remain in a readonly state. Git will not allow any new changes to be committed on top of the checkout commit.

The revert commit will undo a single commit to make it seem as if the changes had never happened. This is kind of like deleting the commit from history.

The reset commit is the most dangerous, because it will permanently take you back to a particular commit, blowing away all the commits which are newer than the commit to which the repo is being reset.

Checkout Commit

Use the `git checkout <commit-id>` command to view the state of the repo at a previous commit.

```
git log --oneline
# 327a27e (HEAD -> master) Fix the name of the stylesheet
# f9c6476 Add a stylesheet to the index
# 8302d10 Add a script to the index
# 808872e Add a title to the index
# 7cb1e23 Add an index w/basic content
```

```
git checkout 8302d10
# Note: switching to '8302d10'.
#
# You are in 'detached HEAD' state. You can look around, make experimental
# changes and commit them, and you can discard any commits you make in this
# state without impacting any branches by switching back to a branch.

# If you want to create a new branch to retain commits you create, you may
# do so (now or later) by using -c with the switch command. Example:
#
#   git switch -c <new-branch-name>
#
# Or undo this operation with:
#
#   git switch -
#
# Turn off this advice by setting config variable advice.detachedHead to false
#
# HEAD is now at 8302d10 Add a script to the index

git status
# HEAD detached at 8302d10
# nothing to commit, working tree clean
```

All the changes that were made after the `8302d10` commit will have been removed. As indicated in the response from the command, it is possible to branch[4] from this commit in order to save changes; otherwise, the repo is now in a readonly state.

Use the **`git checkout master`** command to get back to the initial state of the repo before the checkout commit.

```
git checkout master
# Previous HEAD position was 8302d10 Add a script to the index
# Switched to branch 'master'

git status
# On branch master
# nothing to commit, working tree clean

git log --oneline
# 327a27e (HEAD -> master) Fix the name of the stylesheet
# f9c6476 Add a stylesheet to the index
# 8302d10 Add a script to the index
# 808872e Add a title to the index
# 7cb1e23 Add an index w/basic content
```

[4] See Branches for more information.

Revert Commit

Okay. So it's been decided that the scripting is no longer necessary to meet the project requirements. A revert commit to be done to undo the `8302d10` commit as if it had never happened. The command for this is `git revert <commit-id>`.

This command will transform the command line into text-editor mode. Do worry too much about this, just notice some of the information that is being displayed. Then once you've reviewed the work that will be done to complete the revert commit, type `:wq` and press enter to exit the view.

```
git revert 8302d10
# Revert "Add a script to the index"
#
# This reverts commit 8302d10a8c810ee57da407ce887ce68037e7ed0a.
#
# # Please enter the commit message for your changes. Lines starting
# # with '#' will be ignored, and an empty message aborts the commit.
# #
# # On branch master
# # Changes to be committed:
# #       modified:   index.html
# #       deleted:    scripts/main.js
# #
# ~
# ~
# "~/Desktop/git-primer/.git/COMMIT_EDITMSG" 12L, 339C
# :wq

# [master 81ea6b2] Revert "Add a script to the index"
#  2 files changed, 3 deletions(-)
#  delete mode 100644 scripts/main.js

git log --oneline
# 81ea6b2 (HEAD -> master) Revert "Add a script to the index"
# 327a27e Fix the name of the stylesheet
# f9c6476 Add a stylesheet to the index
# 8302d10 Add a script to the index
# 808872e Add a title to the index
# 7cb1e23 Add an index w/basic content
```

See that the `8302d10` commit has not been deleted, but rather a new commit has been created to revert the changes applied on the commit.

Reset Commit

Lastly, this commit is like the checkout commit, but instead will permanently delete the changes from all newer commits, which came after the commit to which the repo is being reset.

```
git reset 808872e
# Unstaged changes after reset:
# M    index.html

git log --oneline
# 808872e (HEAD -> master) Add a title to the index
# 7cb1e23 Add an index w/basic content

git status
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git restore <file>..." to discard changes in working directory)
#      modified:   index.html
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#      styles/
#
# no changes added to commit (use "git add" and/or "git commit -a")
```

Git reset the commit history back to `808872e` but it didn't delete the changes from the working directory. This is the last safeguard to completely losing the work from those deleted commits. At this point you can recommit the changes as one new commit to recover the work, or delete them once you're absolutely sure they are no longer needed.

These files can be deleted manually, or these two command may be used to reset and clean the working directory:

```
git reset --hard && git clean -df
# HEAD is now at 808872e Add a title to the index
# Removing styles/

git status
# On branch master
# nothing to commit, working tree clean
```

Branches

> Continue to use the repo you've created so far, or download the starting point for this section: 5 - Branches.zip.

Branches allow the development of new features without altering the main application. This is really important if an application has been released to production. For instance, some new features may take a considerable amount of time to develop, but during that period a bug might be reported on production.

Branching the code for the new feature will segment it from the rest of the repo. This leaves the main release branch, the 'master' branch, available for developing a bug fix. In fact, with branching, you could have hundreds of developers working on the same code base simultaneously.

Creating a Branch

Use the `git branch <name>` command to create a new branch.

```
git branch feature-1
```

The `git branch -a` command is used to see all the existing branches. The command will show a list of branches and indicate the active branch with a asterisk (*).

```
git branch -a
#   feature-1
# * master
```

Notice the 'feature-1' branch has been created, but the active branch remains the 'master' branch. The 'master' branch is the repositories default branch and what is typically the main branch for stable, production-ready code.

Before beginning development of feature one, the branch must be checked out.

```
git checkout feature-1
# Switched to branch 'feature-1'

git branch -a
# * feature-1
#   master
```

Developing a Feature on a Branch

The development of the feature can begin and the changes committed to the new branch.

```
git add scripts/feature-1.js
git add index.html

git status
# On branch feature-1
# Changes to be committed:
#   (use "git restore --staged <file>..." to unstage)
#      modified:   index.html
#      new file:   scripts/feature-1.js

git commit -m "Add new script feature"
# [feature-1 c3d7c48] Add new script feature
#  2 files changed, 3 insertions(+)
#  create mode 100644 scripts/feature-1.js
```

The new feature has been completed, but it is completely isolated from the ‘master’ branch. This can be seen with a simple `git log`.

```
git log --oneline
# c3d7c48 (HEAD -> feature-1) Add new script feature
# 808872e (master) Add a title to the index
# 7cb1e23 Add an index w/basic content
```

The HEAD, which is the current working directory, is actively tracking the ‘feature-1’ branch. Also, see that the ‘master’ branch is listed but is shown to be a commit behind the ‘feature-1’ branch.

If you switch back to the ‘master’ branch, you’ll notice that the feature one changes are gone from the repository’s directory. This means that while making changes on the ‘feature-1’ branch, the ‘master’ branch has remained stable and unmodified.

```
git checkout master
# Switched to branch 'master'
```

Deleting a Branch

If a branch becomes irrelevant, there’s no issue deleting it since it is completely isolated from the ‘master’ branch. In order to delete a branch, the same command is used for branch creation but with the `-d` flag to indicate deletion.

```
git branch -d feature-1
# error: The branch 'feature-1' is not fully merged.
# If you are sure you want to delete it, run 'git branch -D feature-1'.

git branch -D feature-1
# Deleted branch feature-1 (was c3d7c48).
```

Don’t worry about the error. This is just a safeguard to help avoid permanently deleting noncommitted code. Branch merging will be described in the next section.

Developing Two Features Simultaneously

To finish up this section go ahead and create two new branches for two different features. The first branch will add a new script feature and the second branch a new style feature.

Feature One

1. Create ‘scripts/main.js’
   a. Insert:
      `console.log('Successfully tested the main script.');`
2. Modify ‘index.html’
   a. Before the `</body>` tag, insert:
      `<script src="scripts/main.js"></script>`
3. Commit the changes

```
git checkout -b script-feature[5]
# Switched to a new branch 'script-feature'

git add scripts/main.js
git add index.html

git status
# On branch script-feature
# Changes to be committed:
#   (use "git restore --staged <file>..." to unstage)
#      modified:   index.html
#      new file:   scripts/main.js

git commit -m "Add new script feature"
# [script-feature 06c3885] Add new script feature
#  2 files changed, 3 insertions(+)
#  create mode 100644 scripts/main.js
```

[5] This command combines `git branch script-feature` and `git checkout script-feature`.

```
git log --oneline
# 06c3885 (HEAD -> script-feature) Add new script feature
# 808872e (master) Add a title to the index
# 7cb1e23 Add an index w/basic content

git checkout master
# Switched to branch 'master'
```

Feature Two

1. Create ‘styles/custom.css’
    a. Insert:
    ```
    body {
         background: #c0c0c0;
    }
    ```
2. Modify ‘index.html’
    a. Before the `</head>` tag, insert:
    `<link rel="stylesheet" href="style/main.css">`
3. Commit the changes
4. Rename ‘style/custom.css’ to ‘style/main.css’
5. Commit the changes

```
git checkout -b style-feature
# Switched to a new branch 'style-feature'

git add styles/custom.css
git add index.html

git status
# On branch style-feature
# Changes to be committed:
#   (use "git restore --staged <file>..." to unstage)
#      modified:   index.html
#      new file:   styles/main.css

git commit -m "Add new style feature"
# [style-feature 96ad65f] Add new style feature
#  2 files changed, 5 insertions(+)
#  create mode 100644 styles/custom.css

git add styles/custom.css
git add styles/main.css
```

```
git status
# On branch style-feature
# Changes to be committed:
#   (use "git restore --staged <file>..." to unstage)
#      renamed:    styles/custom.css -> styles/main.css

git commit -m "Fix the name of the stylesheet"
# [style-feature 3a79ea9] Fix the name of the stylesheet
#  1 file changed, 0 insertions(+), 0 deletions(-)
#  rename styles/{custom.css => main.css} (100%)

git log --oneline
# 3a79ea9 (HEAD -> style-feature) Fix the name of the stylesheet
# 96ad65f Add new style feature
# 808872e (master) Add a title to the index
# 7cb1e23 Add an index w/basic content

git checkout master
# Switched to branch 'master'
```

Once more, list all the branches.

```
git branch -a
# * master
#   script-feature
#   style-feature
```

Merging Branches and Resolving Conflicts

> Continue to use the repo you've created so far, or download the starting point for this section: 6 - Merging Branches and Resolving Conflicts.zip.

Now that two new features have been started, imagine that different developers are doing the work. The style feature was a little easier to complete, is finished, tested, and ready for production. It's time for the 'style-feature' branch to be merged into the 'master' branch.

Fast-Forward and Recursive Merges

First, the branch being merged into has to be checked out. Then, the command to merge a branch is used: **`git merge <branch-name>`**.

```
git merge style-feature
# Updating 808872e..3a79ea9
# Fast-forward
#  index.html      | 2 ++
#  styles/main.css | 3 +++
#  2 files changed, 5 insertions(+)
#  create mode 100644 styles/main.css
```

That's it! This was a simple merge, because the 'master' branch hadn't been changed since the 'style-feature' branch was created. This is indicated by the term "Fast-forward".

If changes had been made to the 'master' branch since it was branched for the style feature and the files that were changed on both branches don't overlap in any way, the 'recursive' strategy is used to merge the two branches. The command's response would instead look like this:

```
# Updating 808872e..3a79ea9
# Merge made by the 'recursive' strategy.
#  index.html      | 2 ++
#  styles/main.css | 3 +++
#  2 files changed, 5 insertions(+)
#  create mode 100644 styles/main.css
```

Merging with Conflicts

Sometimes novice developers are given access to a repository. One has decided to add Google Analytics to the application and make the changes directly on the 'master' branch.

Pretend you're the novice. Open the 'index.html' file and add this code, on a new line, before the `</body>` tag.

```
<script>
  window.ga = function () { ga.q.push(arguments) }; ga.q = []; ga.l = +new Date;
  ga('create', 'UA-XXXXX-Y', 'auto'); ga('set', 'anonymizeIp', true);
  ga('set', 'transport', 'beacon'); ga('send', 'pageview');
</script>
<script src="https://www.google-analytics.com/analytics.js" async></script>
```

Commit the novice's work and get out of that yucky headspace.

```
git add index.html

git commit -m "Add Google Analytics"
# [master 57f4958] Add Google Analytics
#  1 file changed, 7 insertions(+)
```

The development of the 'script-feature' branch has just completed, so it's time to merge it into 'master'.

```
git merge script-feature
# Auto-merging index.html
# CONFLICT (content): Merge conflict in index.html
# Automatic merge failed; fix conflicts and then commit the result.
```

Oh no! There's a conflict. If you open 'index.html', the file will contain some unique content. This indicates the conflict and shows the newer change of the HEAD and the incoming change from the 'script-feature' branch.

```
<<<<<<< HEAD
<script>
  window.ga = function () { ga.q.push(arguments) }; ga.q = []; ga.l = +new Date;
  ga('create', 'UA-XXXXX-Y', 'auto'); ga('set', 'anonymizeIp', true);
  ga('set', 'transport', 'beacon'); ga('send', 'pageview');
</script>
<script src="https://www.google-analytics.com/analytics.js" async></script>
=======
<script src="scripts/main.js"></script>
>>>>>>> script-feature
```

This is Git's way of asking for help, because it's reached a situation where there is no clear option for merging changes. In fact, there are three options:

1. Keep only the changes on the HEAD
2. Keep only the change from the incoming branch
3. Keep both and make sure they don't wreck each other

In this case both changes should be kept, so delete the extra markup and save the file.

```
<script>
  window.ga = function () { ga.q.push(arguments) }; ga.q = []; ga.l = +new Date;
  ga('create', 'UA-XXXXX-Y', 'auto'); ga('set', 'anonymizeIp', true);
  ga('set', 'transport', 'beacon'); ga('send', 'pageview');
</script>
<script src="https://www.google-analytics.com/analytics.js" async></script>
<script src="scripts/main.js"></script>
```

Then to finish up the merge, stage the file, and commit the changes. With this commit no message should be provided, because the repo is still trying to finish up the merge and will have automatically created the message for you.

```
git add index.html

git commit[6]
# [master 174e9a8] Merge branch 'script-feature'

git log --oneline
# 174e9a8 (HEAD -> feature-1) Merge branch 'script-feature'
# 57f4958 Add Google Analytics
# 3a79ea9 (style-feature) Add new style feature
# 96ad65f Add new style feature
# 06c3885 (script-feature) Add new script feature
# 808872e Add a title to the index
# 7cb1e23 Add an index w/basic content
```

[6] The command-line will transform to text-editor mode. See Undoing Things \ Revert Commit.

Syncing Remote Repos

Up to this point, this guide has used local repositories, but the whole point of Git is project sharing and collaboration. Unless starting a new project from scratch, most times you will be cloning a remote repository to your local machine from an external host.

GitHub is probably the most popular external repo host. It has thousands of public repos and adds a lot of additional features for sharing and collaborating on Git projects.

Earlier in this guide, you created a repo local on your computer. However, it is often easier to create the repo remotely, then clone it locally. Before stepping through how to create a remote repo, start by cloning the repo which contains all the samples for this primer: https://github.com/roydukkey/git-primer.

When you load up the link, there should be several options for copying the repo. Firstly, you can download the code as a ZIP archive. Secondly, there is an option to open the repo with GitHub Desktop. Lastly, and the option you'll perform in a bit, there is a URL to clone the repo using Git.

Cloning Remotes

Making sure to be in an empty directory, run the `git clone <URL>` command to copy the remote repository to your computer.

```
git clone https://github.com/roydukkey/git-primer.git
# Cloning into 'git-primer'...
# remote: Enumerating objects: 19, done.
# remote: Counting objects: 100% (19/19), done.
# remote: Compressing objects: 100% (13/13), done.
# remote: Total 19 (delta 9), reused 13 (delta 6), pack-reused 0
# Unpacking objects: 100% (19/19), done.
```

The remote repo has been cloned locally. Changes can be made on your machine without changing the remote repo and, if desired, may be pushed to update the remote repo.

If you look into the repo, you'll see all the same content that was listed on GitHub, with the addition of the '.git' folder.

```
cd git-primer

ls -a
# .                                     3 - Making Commits.zip
# ..                                    4 - Undoing Things.zip
# .editorconfig                         5 - Branches.zip
# .git                                  6 - Merging Branches and Resolving Conflicts.zip
# .gitignore                            Git and GitHub - A Comprehensive Primer.pdf
# 1 - Repo Status and Changes.zip       README.md
# 2 - Staging Files.zip
```

Creating Remotes

GitHub makes it very simple to create new repositories. Simply go to https://github.com/new, enter a repository name, and click the button to create the repo. There are a few more options, but this is enough to get the most basic repo setup.

Sign up for a GitHub, and create your first repository named "git-test". Then, clone it locally using the same commands from the last section. Don't worry about the empty repo warning; indeed... you have cloned an empty repo.

Basic Git Workflow

It's one thing to make changes locally, but normally those changes need to get somewhere else before they are finally deployed to production. A typical workflow for projects might looks like this:

1. Adam creates a remote repo
2. John clones the repo locally
3. John creates a new 'feature' branch
    a. Develops the feature
    b. Tests the feature
    c. Pushes the branch to remote
4. Adam clones the repo locally
    a. Reviews the 'feature' branch
    b. Merges the 'feature' branch into 'master' branch
    c. Pushes the changes on local 'master' to remote 'master'
5. John checks out the 'master' branch
    a. Pulls changes from remote 'master'
    b. Verifies the merged changes
    c. Deletes the 'feature' branch

This primer has covered most of the commands necessary to complete this workflow, with two exceptions for pushing and pulling changes to and from remote repos.

Pushing to Remotes

In order to begin pushing to your new remote repo, first create an ‘index.html’ with the barebones and commit it.

The `git push origin <branch>` command is used to push these changes from the local repo to the remote repo.

```
git push origin master
# Enumerating objects: 3, done.
# Counting objects: 100% (3/3), done.
# Delta compression using up to 12 threads
# Compressing objects: 100% (2/2), done.
# Writing objects: 100% (3/3), 372 bytes | 372.00 KiB/s, done.
# Total 3 (delta 0), reused 0 (delta 0)
# To https://github.com/roydukkey/git-test.git
#  * [new branch]      master -> master
```

The changes will have been successfully uploaded and viewable on GitHub from the web browser.

Pulling from Remotes

This is essentially the opposite of pushing. Instead of sending changes to the remote, pulling will receive changes from the remote.

From your ‘git-test’ repo on the GitHub website, click the button to add a README. An online editor will display for creating a README Markdown file.[7] Normally, information is added to this file which will assist other developers in understanding how to install, setup, and use your repository. In this case however, just use the content provided by GitHub and commit the new file in the browser.

[7] More information on Markdown can be found in GitHub’s Documentation.

Return to the ‘git-test’ repo from the command-line, so you can check the remote repo for changes with the `git fetch` command.

```
git fetch
# remote: Enumerating objects: 4, done.
# remote: Counting objects: 100% (4/4), done.
# remote: Compressing objects: 100% (2/2), done.
# remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
# Unpacking objects: 100% (3/3), done.
# From https://github.com/roydukkey/git-test
#    5c9ece5..e0bd472  master     -> origin/master

git status
# On branch master
# Your branch is behind 'origin/master' by 1 commit, and can be fast-forwarded.
#   (use "git pull" to update your local branch)
#
# nothing to commit, working tree clean
```

The `git status` command is showing that your local repo has fallen out of sync with the remote repo. The `git fetch` command will not update the active working directory or change the files in any way. It only updates what it knows about the remote repo, so that it can inform you about changes made up-stream.

The `git pull origin <branch>` command is used to pull the change locally.

```
git pull origin master
# From https://github.com/roydukkey/git-test
#   * branch            master     -> FETCH_HEAD
# Updating 5c9ece5..e0bd472
# Fast-forward
#  README.md | 1 +
#  1 file changed, 1 insertion(+)
#  create mode 100644 README.md
```

The `git pull` essentially performs a `git merge` request of the remote ‘master’ branch into the local ‘master’ branch. Some of the same possibilities exist for the pull request as do for merges. Importantly, if there are uncommitted changes locally, there will be conflicts.

Credits / Information

A lot of this guide is derived from Shaun Pelling and his video tutorials on YouTube. Code snippet syntax highlighting provided by the Chimera Theme for Visual Studio Code.