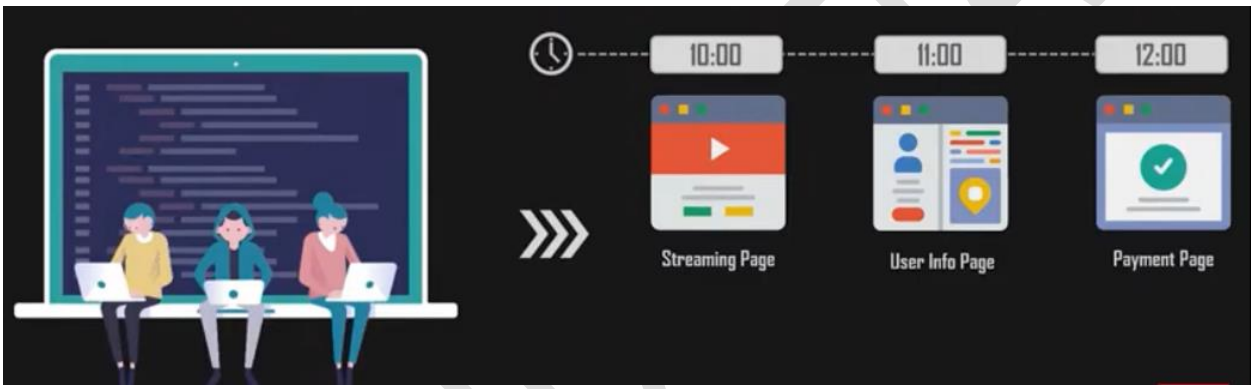
Git Handbook

What is Version Control?

A version control system, or VCS, tracks the history of changes as people and teams collaborate on projects together. As the project evolves, teams can run tests, fix bugs, and contribute new code with the confidence that any version can be recovered at any time. Developers can review project history to find out:

- Which changes were made?
- Who made the changes?
- When were the changes made?
- Why were changes needed?

Suppose there are 3 developers working remotely on a web application. Each of them write code for different pages.



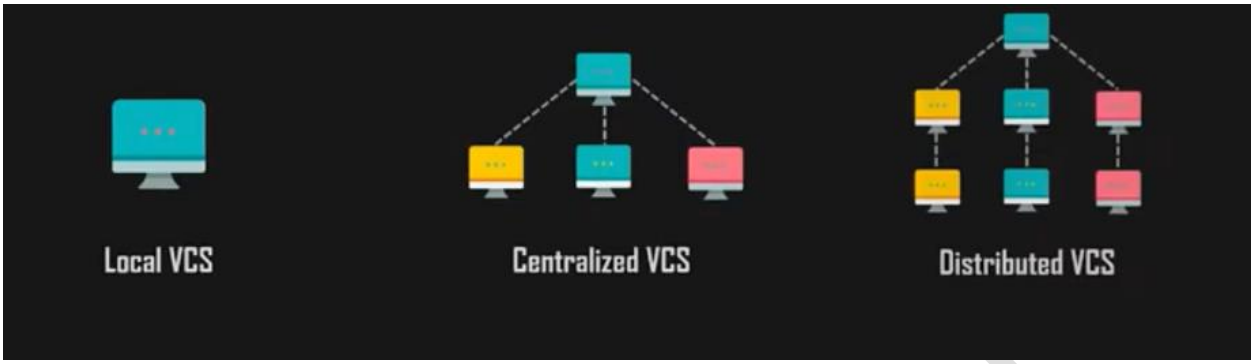
With version control, each modification is recorded and updated to one central directory/folder.

The current state of our project is saved like a snapshot with each modification.

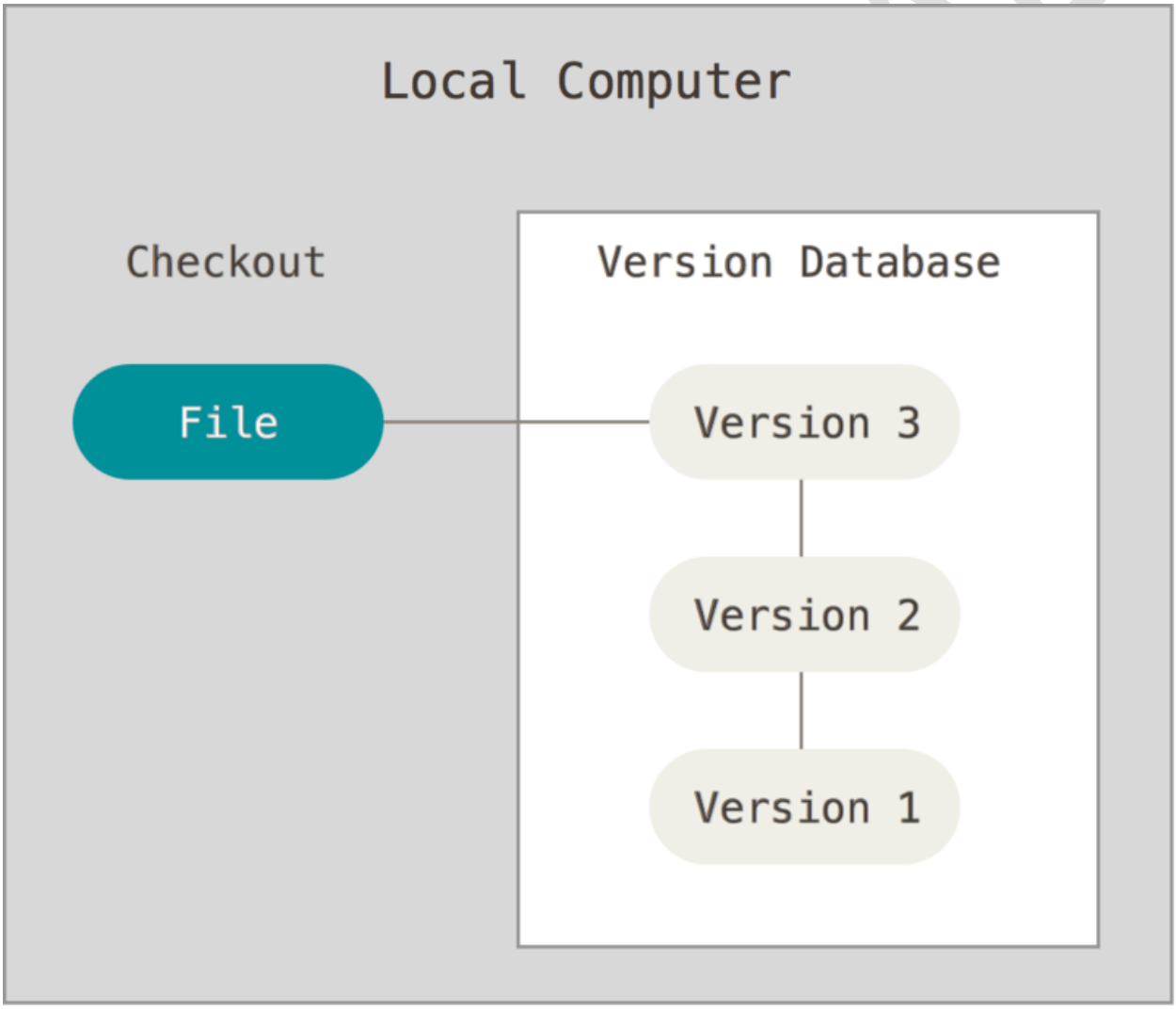
Version control is a system that records changes in a file or set of files over time so that we can recall specific versions later. These versions are recorded in a repository and can be recalled from the same.

Version Control is the management of changes to documents, computer programs, large websites, and other collection of information.

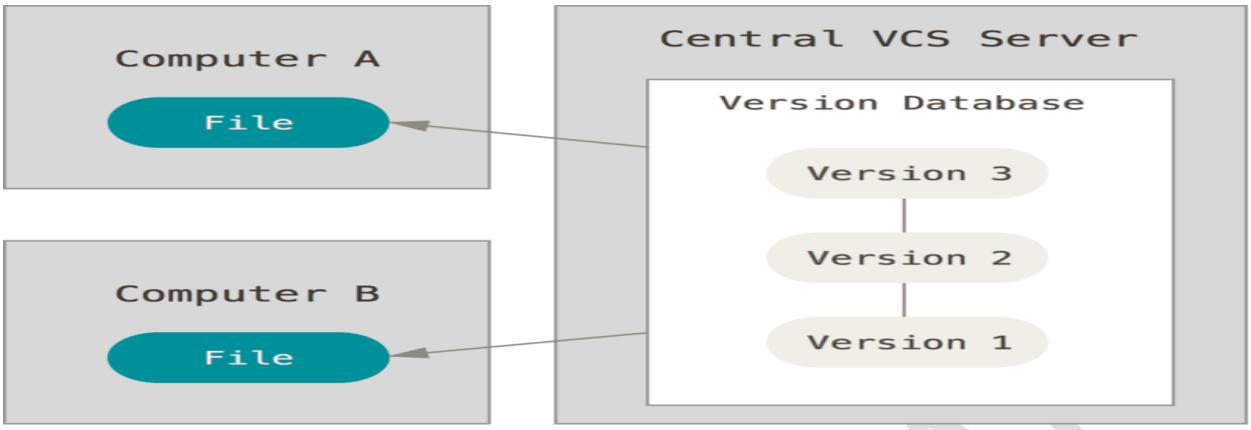
There are local, centralized, and Distributed version control systems to understand before we proceed.



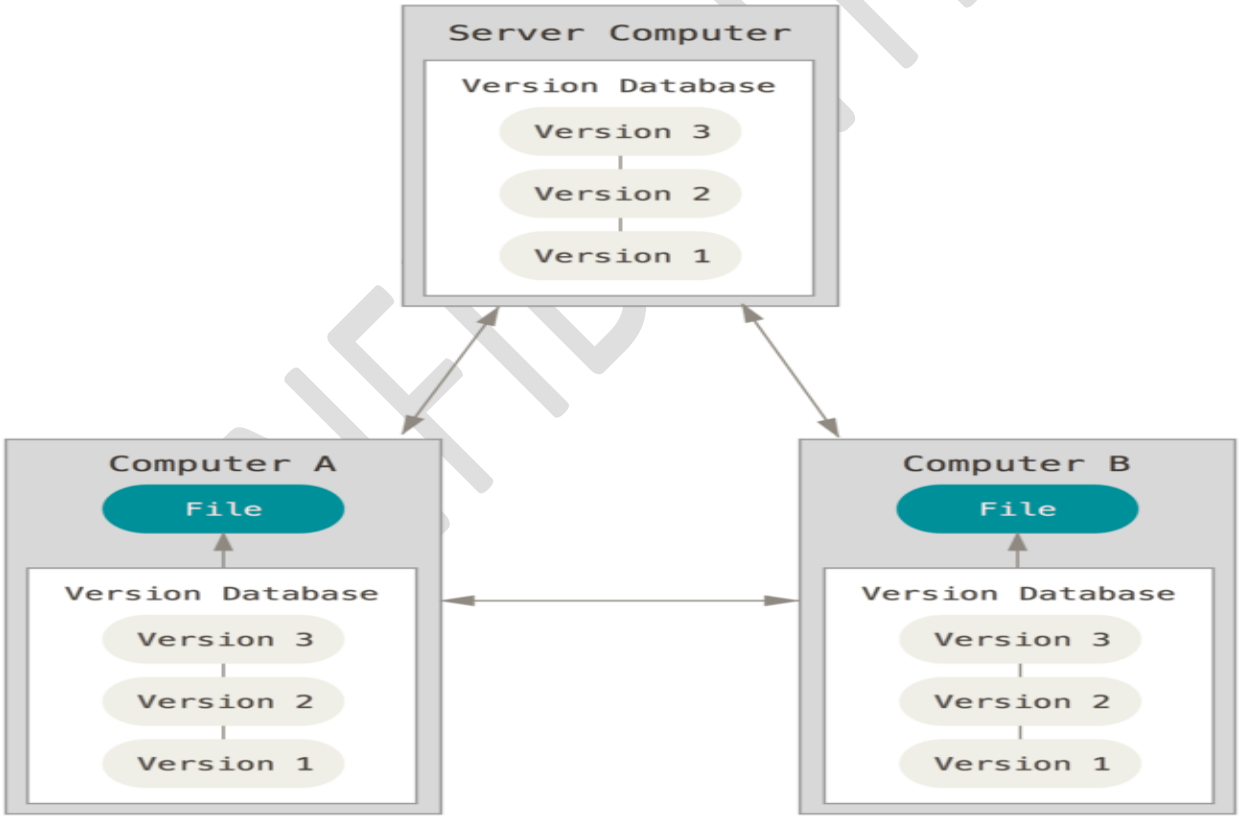
Local VCS --- where all the code, changes are tracked locally at system.



Centralized VCS – where developer can connect to centralized servers, they can update to centralized server and they can collaborate.



Distributed VCS – Each developer has local copy of the repository which they can collaborate with other developers by merging, pushing them into remote repository.



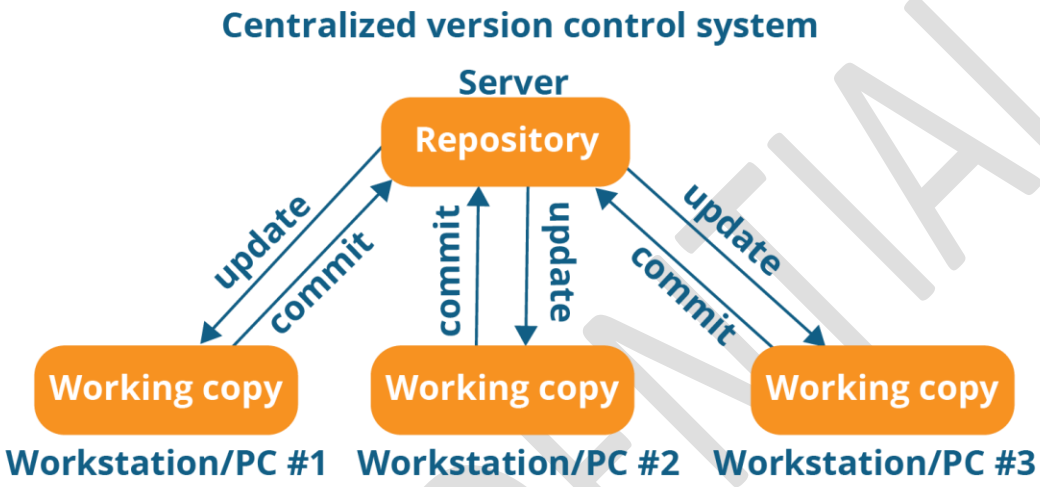
Now we can talk more about main two types of VCS used in IT industry:

- Centralized Version Control System (CVCS)
- Distributed Version Control System (DVCS)

Centralized VCS

Centralized version control system (CVCS) uses a central server to store all files and enables team collaboration. It works on a single repository to which users can directly access a central server.

Please refer to the diagram below to get a better idea of CVCS:



The repository in the above diagram indicates a central server that could be local or remote which is directly connected to each of the programmer's workstation.

Every programmer can extract or **update** their workstations with the data present in the repository or can make changes to the data or **commit** in the repository. Every operation is performed directly on the repository.

Even though it seems convenient to maintain a single repository, it has some major drawbacks. Some of them are:

It is not locally available; meaning you always need to be connected to a network to perform any action.

Since everything is centralized, in any case of the central server getting crashed or corrupted will result in losing the entire data of the project.

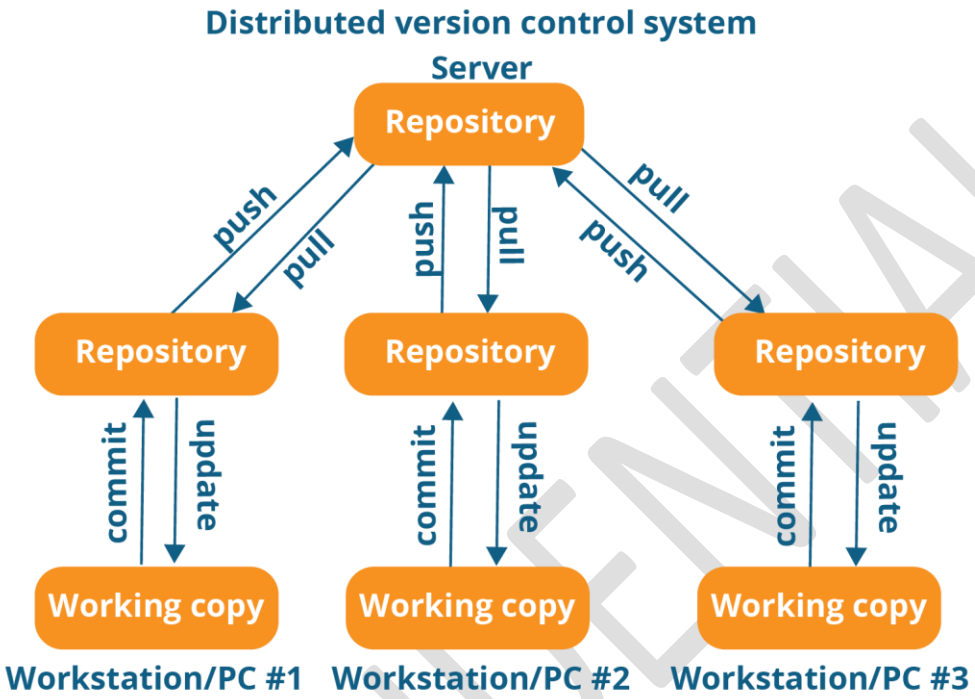
This is when Distributed VCS comes to the rescue.

Distributed VCS

These systems do not necessarily rely on a central server to store all the versions of a project file.

In Distributed VCS, every contributor has a local copy or “clone” of the main repository i.e. everyone maintains a local repository of their own which contains all the files and metadata present in the main repository.

You will understand it better by referring to the diagram below:



As you can see in the above diagram, every programmer maintains a local repository on its own, which is actually the copy or clone of the central repository on their hard drive. They can commit and update their local repository without any interference.

They can update their local repositories with new data from the central server by an operation called “**pull**” and affect changes to the main repository by an operation called “**push**” from their local repository.

The act of cloning an entire repository into your workstation to get a local repository gives you the following advantages:

All operations (except push & pull) are very fast because the tool only needs to access the hard drive, not a remote server. Hence, you do not always need an internet connection.

Committing new change-sets can be done locally without manipulating the data on the main repository. Once you have a group of change-sets ready, you can push them all at once.

Since every contributor has a full copy of the project repository, they can share changes with one another if they want to get some feedback before affecting changes in the main repository.

If the central server gets crashed at any point of time, the lost data can be easily recovered from any one of the contributor's local repositories.

After knowing Distributed VCS, it's time we take a dive into what is Git.

What is Git – Why Git Came into Existence?

We all know "Necessity is the mother of all inventions". And similarly, Git was also invented to fulfill certain necessities that the developers faced before Git.

What is the purpose of Git?

Git is primarily used to manage your project, comprising a set of code/text files that may change.

A Short History of Git

Git is an open-source distributed version control system. It is designed to handle minor to major projects with high speed and efficiency. It is developed to co-ordinate the work among the developers. The version control allows us to track and work together with our team members at the same workspace.

Git is foundation of many services like GitHub and GitLab, but we can use Git without using any other Git services. Git can be used privately and publicly.

Git was created by Linus Torvalds in 2005 to develop Linux Kernel. It is also used as an important distributed version-control tool for the DevOps.

Git is easy to learn and has fast performance. It is superior to other SCM tools like Subversion, CVS, Perforce.

A Git *repository* is a structured history of a project's files through multiple explicitly recorded *snapshots* or *versions*.Each snapshot is called a *commit* (noun) or a *revision*. Each commit is accompanied by metadata describing which earlier commits it follows, who created it and when, plus a description of the important changes since the last commit.

A local Git repository is paired with a *working copy* of the project files in a directory where the user can prepare and commit changes. The git tool lets users inspect the repository history as well as the status of the working copy. Each time the user edits the working copy to create a meaningful new version of the project, they run git commands to record new commits in the repository.

Repositories can be *cloned* to create a new repository with the same history. New commits can be created in separate clones, then shared by *pushing* and *pulling* commit history between related repository clones to bring them up to date with each other.

Some remarkable features of Git are as follows:

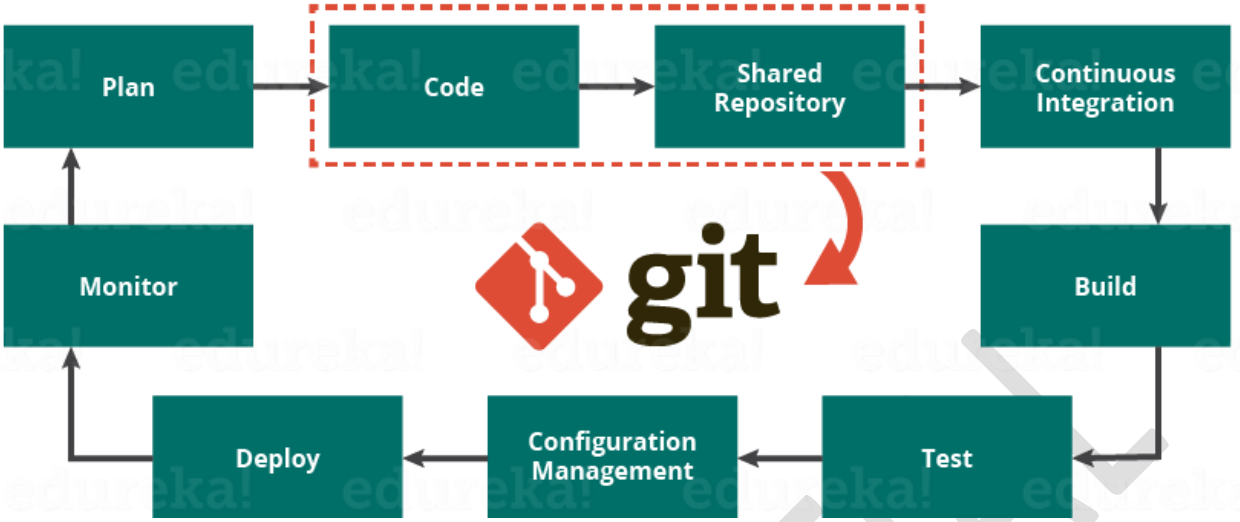
Git is an example of a distributed version control system (DVCS) commonly used for open source and commercial software development. DVCSs allow full access to every file, branch, and iteration of a project, and allows every user access to a full and self-contained history of all changes. Unlike once popular centralized version control systems, DVCSs like Git don't need a constant connection to a central repository. Developers can work anywhere and collaborate asynchronously from any time zone.

What is Git – Role of Git in DevOps?

Now that you know what Git is, you should know Git is an integral part of DevOps.

DevOps is the practice of bringing agility to the process of development and operations. It's an entirely new ideology that has swept IT organizations worldwide, boosting project life cycles and in turn increasing profits. DevOps promotes communication between development engineers and operations, participating together in the entire service life cycle, from design through the development process to production support.

The diagram below depicts the Devops life cycle and displays how Git fits in Devops.

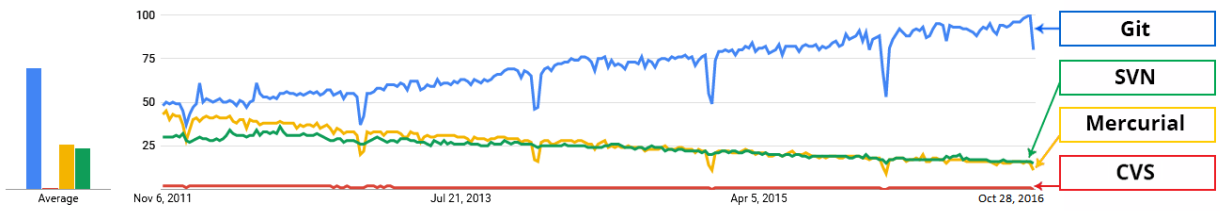


The diagram above shows the entire life cycle of Devops starting from planning the project to its deployment and monitoring. Git plays a vital role when it comes to managing the code that the collaborators contribute to the shared repository. This code is then extracted for performing continuous integration to create a build and test it on the test server and eventually deploy it on the production.

Tools like Git enable communication between the development and the operations team. When you are developing a large project with a huge number of collaborators, it is very important to have communication between the collaborators while making changes in the project. Commit messages in Git play a very important role in communicating among the team. The bits and pieces that we all deploy lies in the Version Control system like Git. To succeed in DevOps, you need to have all the communication in Version Control. Hence, Git plays a vital role in succeeding at DevOps.

Who uses Git? - Popular Companies Using Git

Git has earned way more popularity compared to other version control tools available in the market like Apache Subversion (SVN), Concurrent Version Systems (CVS), Mercurial etc. You can compare the interest of Git by time with other version control tools with the graph collected from *Google Trends* below:



In large companies, products are generally developed by developers located all around the world. To enable communication among them, Git is the solution.

Some companies that use Git for version control are Facebook, Yahoo, Zynga, Quora, Twitter, eBay, Salesforce, Microsoft and many more.

Lately, all of Microsoft's new development work has been in Git features. Microsoft is migrating .NET and many of its open source projects on GitHub which are managed by Git.

Why Version Control?

1. Collaboration -- Shared workspace and real time updates
2. Manage Versions – All versions of code are preserved
3. Rollbacks – easy rollback to current versions
4. Reduce Downtime – Reverse faulty update and save time
5. Analyze Project – Analyze and compare projects
6. Without version control, team members are subject to redundant tasks, slower timelines, and multiple copies of a single project.
7. To eliminate unnecessary work, Git and other VCSs give each contributor a unified and consistent view of a project, surfacing work that's already in progress.
8. Seeing a transparent history of changes, who made them, and how they contribute to the development of a project helps team members stay aligned while working independently.

Why Git?

According to the latest Stack Overflow developer survey, more than 70 percent of developers use Git, making it the most-used VCS in the world. Git is commonly used for both open source and commercial software development, with significant benefits for individuals, teams, and businesses.

Git lets developers see the entire timeline of their changes, decisions, and progression of any project in one place. From the moment they access the history of a project, the developer has all the context they need to understand it and start contributing.

Developers work in every time zone. With a DVCS like Git, collaboration can happen any time while maintaining source code integrity. Using branches, developers can safely propose changes to production code.

Businesses using Git can break down communication barriers between teams and keep them focused on doing their best work. Plus, Git makes it possible to align experts across a business to collaborate on major projects.

GitHub is a highly used software that is typically used for version control. It is helpful when more than just one person is working on a project. Say for example, a software developer team wants to build a website, and everyone has to update their codes simultaneously while working on the project. In this case, GitHub helps them to build a centralized repository where everyone can upload, edit, and manage the code files.

Features of GitHub

Git Handbook

GitHub is a place where programmers and designers work together. They collaborate, contribute, and fix bugs together. It hosts plenty of open source projects and codes of various programming languages.

Some of its significant features are as follows.

- Collaboration
- Integrated issue and bug tracking
- Graphical representation of branches
- Git repositories hosting
- Project management
- Team management
- Code hosting
- Track and assign tasks
- Conversations
- Wikisc

What's a repository?

A *repository*, or Git project, encompasses the entire collection of files and folders associated with a project, along with each file's revision history. The file history appears as snapshots in time called *commits*, and the commits exist as a linked list relationship, and can be organized into multiple lines of development called *branches*. Because Git is a DVCS, repositories are self-contained units and anyone who owns a copy of the repository can access the entire codebase and its history. Using the command line or other ease-of-use interfaces, a git repository also allows for: interaction with the history, cloning, creating branches, committing, merging, comparing changes across versions of code, and more.

Working in repositories keeps development projects organized and protected. Developers are encouraged to fix bugs, or create fresh features, without fear of derailing mainline development efforts. Git facilitates this through the use of topic branches: lightweight pointers to commits in history that can be easily created and deprecated when no longer needed.

Through platforms like GitHub, Git also provides more opportunities for project transparency and collaboration. Public repositories help teams work together to build the best possible final product.

Basic Git commands

To use Git, developers use specific commands to copy, create, change, and combine code. These commands can be executed directly from the command line or by using an application like GitHub Desktop or Git Kraken. Here are some common commands for using Git:

`git init` initializes a brand-new Git repository and begins tracking an existing directory. It adds a hidden subfolder within the existing directory that houses the internal data structure required for version control.

`git clone` creates a local copy of a project that already exists remotely. The clone includes all the project's files, history, and branches.

`git add` stages a change. Git tracks changes to a developer's codebase, but it's necessary to stage and take a snapshot of the changes to include them in the project's history. This command performs staging, the first part of that two-step process. Any changes that are staged will become a part of the next snapshot and a part of the project's history. Staging and committing separately gives developers complete control over the history of their project without changing how they code and work.

`git commit` saves the snapshot to the project history and completes the change-tracking process. In short, a commit functions like taking a photo. Anything that's been staged with `git add` will become a part of the snapshot with `git commit`.

`git status` shows the status of changes as untracked, modified, or staged.

`git branch` shows the branches being worked on locally.

`git merge` merges lines of development together. This command is typically used to combine changes made on two distinct branches. For example, a developer would merge when they want to combine changes from a feature branch into the main branch for deployment.

`git pull` updates the local line of development with updates from its remote counterpart. Developers use this command if a teammate has made commits to a branch on a remote, and they would like to reflect those changes in their local environment.

`git push` updates the remote repository with any commits made locally to a branch.

Learn more from a full reference guide to Git commands.

How GitHub fits in

GitHub is a Git hosting repository that provides developers with tools to ship better code through command line features, issues (threaded discussions), pull requests, code review, or the use of a collection of free and for-purchase apps in the GitHub Marketplace. With collaboration layers like the GitHub flow, a community of 15 million developers, and an ecosystem with hundreds of integrations, GitHub changes the way software is built.

How GitHub works

GitHub builds collaboration directly into the development process. Work is organized into repositories, where developers can outline requirements or direction and set expectations for

team members. Then, using the GitHub flow, developers simply create a branch to work on updates, commit changes to save them, open a pull request to propose and discuss changes, and merge pull requests once everyone is on the same page.

Setting Up Git

You need to setup Git on your local machine, as follows:

Download & Install:

For Windows and Mac, download the installer from http://git-scm.com/downloads and run the downloaded installer.

For Ubuntu, issue command "sudo apt-get install git".

For Windows, use the "Git Bash" command shell bundled with Git Installer to issue commands. For Mac/Ubuntu, use the "Terminal".

Customize Git:
Issue "git config" command (for Windows, run "Git Bash" from the Git installed directory. For Ubuntu/Mac, launch a "Terminal"):

```
// Set up your username and email (to be used in labeling your commits)
$ git config --global user.name "your-name"
$ git config --global user.email "your-email@youremail.com"
```

The settings are kept in "<GIT_HOME>/etc/gitconfig" (of the GIT installed directory) and "<USER_HOME>/. gitconfig" of the user's home directory.

You can issue "git config --list" to list the settings:

```
$ git config --list
user.email=xxxxxx@xxxxxx.com
user.name=xxxxxx
```

Git Basics

Git Commands

Git Handbook

Git provides a set of simple, distinct, standalone commands developed according to the "Unix toolkit" philosophy - build small, interoperable tools.

To issue a command, start a "Terminal" (for Ubuntu/Mac) or "Git Bash" (for Windows):

$ git <command> <arguments>

The commonly used commands are:

- **init**, **clone**, **config**: for starting a Git-managed project.
- **add**, **mv**, **rm**: for staging file changes.
- **commit**, **rebase**, **reset**, **tag**:
- **status**, **log**, **diff**, **grep**, **show**: show status
- **checkout**, **branch**, **merge**, **push**, **fetch**, **pull**

Help and Manual

The best way to get help these days is certainly *googling*.

To get help on Git commands:

$ git help <*command*>

// or

$ git <*command*> --help

Getting Started with Local Repo

There are 2 ways to start a Git-managed project:

- Starting your own project.
- Cloning an existing project from a GIT host.

Setup the Working Directory for a New Project

Let's start a programming project under the *working directory* called "hello-git", with one source file "Hello.java" (or "Hello.cpp", or "Hello.c") as follows:

// Hello.java

```
public class Hello {

  public static void main(String[] args) {

    System.out.println("Hello, world from GIT!");
```

```
  }
}
```

Compile the "Hello.java" into "Hello.class" (or "Hello.cpp" or "Hello.c" into "Hello.exe").

It is also highly recommended to provide a "README.md" file (a text file in a so-called "Markdown" syntax such as "GitHub Flavored Markdown") to describe your project:

```
// README.md

This is the README file for the Hello-world project.
```

Now, we have 3 files in the *working tree*: "Hello.java", "Hello.class" and "README.md". We do not wish to track the ".class" as they can be reproduced from ".java".

Initialize a new Gilt Repo (git init)

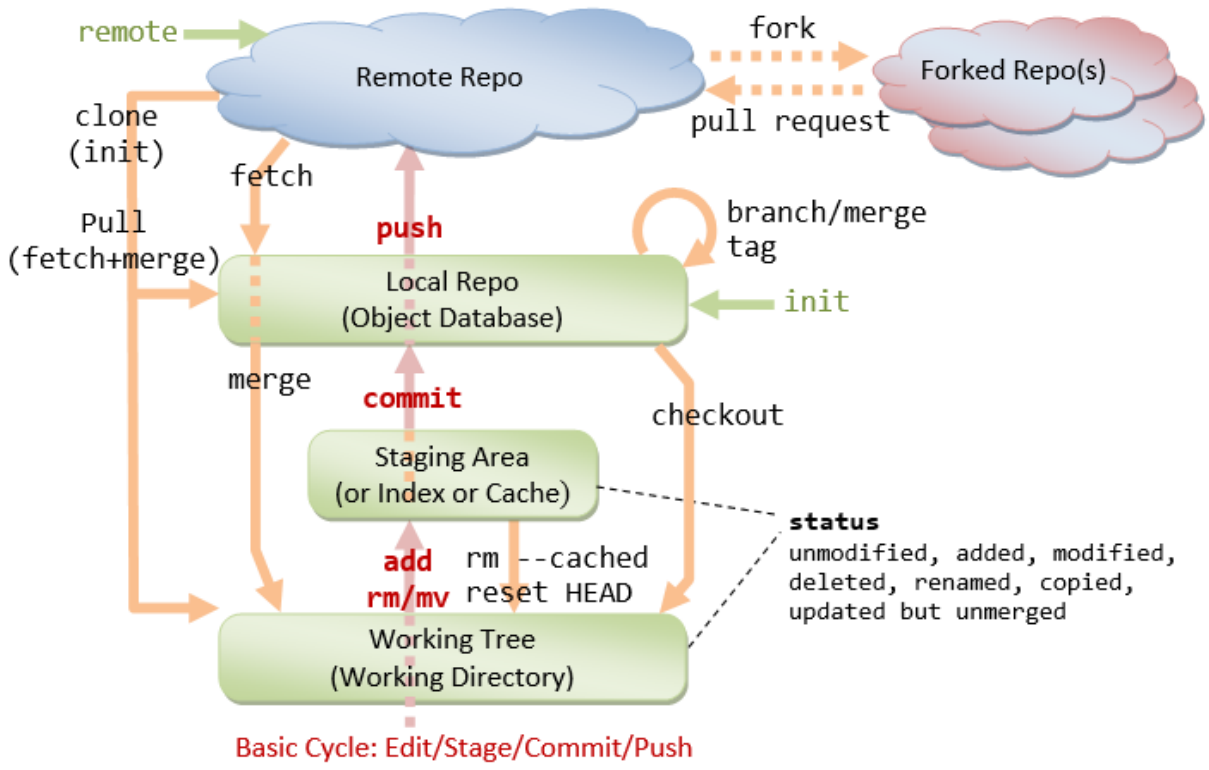
To manage a project under Git, run "git init" at the project *root* directory (i.e., "hello-git") (via "Git Bash" for Windows, or "Terminal" for Ubuntu/Mac):

```
// Change directory to the project directory
$ cd /path-to/hello-git

// Initialize Git repo for this project
$ git init
Initialized empty Git repository in /path-to/hello-git/.git/

$ ls -al
drwxr-xr-x  1 xxxxx  xxxxx  4096 Sep 14 14:58 .git
-rw-r--r--  1 xxxxx  xxxxx   426 Sep 14 14:40 Hello.class
-rw-r--r--  1 xxxxx  xxxxx   142 Sep 14 14:32 Hello.java
-rw-r--r--  1 xxxxx  xxxxx    66 Sep 14 14:33 README.md
```

A hidden sub-directory called ". git" will be created under your project *root* directory (as shown in the above "ls -a" listing), which contains ALL Git related data.

Take note that EACH Git repo is associated with a project directory (and its sub-directories). The Git repo is completely containing within the project directory. Hence, it is safe to copy, move or rename the project directory. If your project uses more than one directory, you may create one Git repo for EACH directory, or use symlinks to link up the directories, or ... (?!).

Git Storage Model



The local repo after "git init" is empty. You need to explicitly deposit files into the repo.

Before we proceed, it is important to stress that Git manages *changes to files between so-called commits*. In other words, it is a *version control system* that allows you to keep track of the *file changes at the commits*.

Staging File Changes for Tracking (git add <file>...)

Issue a "git status" command to show the status of the files:

```
$ git status
On branch master
Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    Hello.class
    Hello.java
    README.md
nothing added to commit but untracked files present (use "git add" to track)
```

By default, we start on a *branch* called "master". We will discuss "branch" later.

In Git, the files in the working tree are either *untracked* or *tracked*. Currently, all 3 files are *untracked*. To stage a new file for tracking, use "git add <file>..." command.

```
// Add README.md file
$ git add README.md

$ git status
On branch master
```

Initial commit

Changes to be committed:

(use "git rm --cached <file>..." to unstage)

new file: README.md

Untracked files:

(use "git add <file>..." to include in what will be committed)

Hello.class

Hello.java

// You can use wildcard * in the filename

// Add all Java source files into Git repo

$ **git add *.java**

// You can also include multiple files in the "git add"

// E.g.,

// git add Hello.java README.md

$ **git status**

On branch master

Initial commit

Changes to be committed:

(use "git rm --cached <file>..." to unstage)

new file: Hello.java

new file: README.md

Untracked files:

(use "git add <file>..." to include in what will be committed)

Hello.class

The command "**git add <file>...**" takes one or more filenames or pathnames with possibly wildcards pattern. You can also use "**git add .**" to add all the files in the current directory (and all sub-directories). But this will include "**Hello.class**", which we do not wish to be tracked.

When a new file is added, it is ***staged*** (or ***index***ed, or ***cached***) in the ***staging area*** (as shown in the GIT storage model), but NOT yet ***committed***.

Git uses two stages to commit file changes:

"**git add <file>**" to stage file changes into the ***staging area***, and

"**git commit**" to commit ALL the file changes in the ***staging area*** to the ***local repo***.

The staging area allows you to group related file changes and commit them together.

Committing File Changes (git commit)

The "**git commit**" command commits ALL the file changes in the ***staging area***. Use a **-m** option to provide a ***message*** for the commit.

$ **git commit -m "First commit"** // -m to specify the commit message

[master (root-commit) 858f3e7] first commit

2 files changed, 8 insertions(+)

create mode 100644 Hello.java

create mode 100644 README.md

// Check the status

$ git status

On branch master

Untracked files:

(use "git add <file>..." to include in what will be committed)

Hello.class

nothing added to commit but untracked files present (use "git add" to track)

Viewing the Commit Data (git log)

Git records several pieces of metadata for every commit, which includes a log message, timestamp, the author's username and email (set during customization).

You can use "git log" to list the commit data; or "git log --stat" to view the file statistics:

$ git log

commit 858f3e71b95271ea320d45b69f44dc55cf1ff794

Author: *username* <*email*>

Date: Thu Nov 29 13:31:32 2012 +0800

First commit

$ git log --stat

commit 858f3e71b95271ea320d45b69f44dc55cf1ff794

Author: *username* <*email*>

Date: Thu Nov 29 13:31:32 2012 +0800

First commit

Hello.java | 6 ++++++

README.md | 2 ++

2 files changed, 8 insertions(+)

Each commit is identified by a 40-hex-digit SHA-1 hash code. But we typcially use the first 7 hex digits to reference a commit, as highlighted.

To view the commit details, use "git log -p", which lists all the *patches* (or *changes*).

$ git log -p

commit 858f3e71b95271ea320d45b69f44dc55cf1ff794

Author: *username* <*email*>

Date: Thu Nov 29 13:31:32 2012 +0800

  First commit

diff --git a/Hello.java b/Hello.java

new file mode 100644

index 0000000..dc8d4cf

--- /dev/null

+++ b/Hello.java

@@ -0,0 +1,6 @@

+// Hello.java

+public class Hello {

+ public static void main(String[] args) {

+   System.out.println("Hello, world from GIT!");

+ }

+}

diff --git a/README.md b/README.md

new file mode 100644

index 0000000..9565113

--- /dev/null

+++ b/README.md

@@ -0,0 +1,2 @@

+// README.md

+This is the README file for the Hello-world project.

Below are more options of using "git log":

$ **git log --oneline**

// Display EACH commit in one line.

$ **git log --author="<author-name-pattern>"**

// Display commits by author

$ **git log <file-pattern>**

// Display commits for particular file(s)

// EXAMPLES

$ **git log --author="Tan Ah Teck" -p Hello.java**

// Display commits for file "Hello.java" by a particular author

File Status (git status)

A file could be *untracked* or *tracked*.

As mentioned, Git tracks file changes at commits. In Git, changes for a *tracked* file could be:

unstaged (in *Working Tree*) - called *unstaged changes*,

staged (in *Staging Area* or *Index* or *Cache*) - called *staged changes*, or

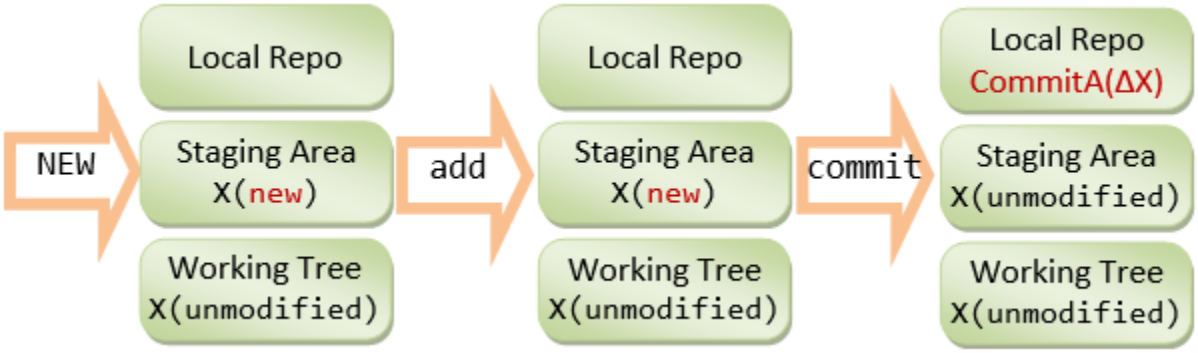
committed (in *local repo object database*).

The files in “working tree” or “staging area”

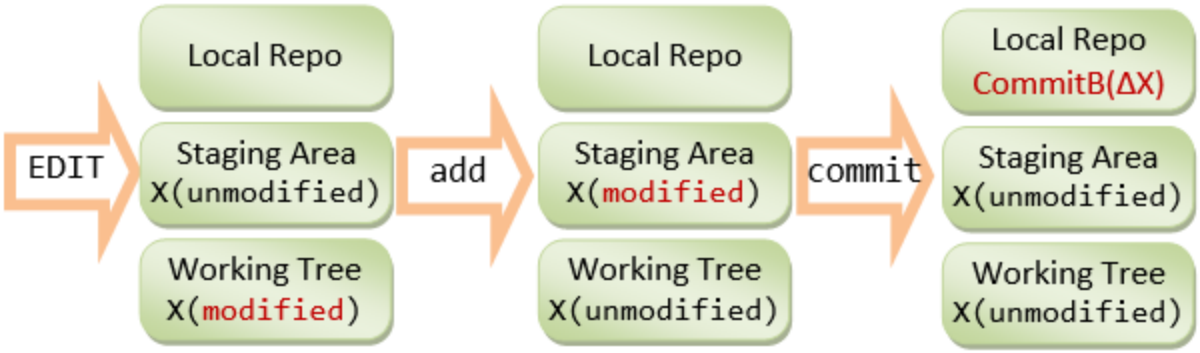
could have status of *unmodified*, *added*, *modified*, *deleted*, *renamed*, *copied*, as reported by "git status".

The "git status" output is divided into 3 sections: "Changes not staged for commit" for the unstaged changes in "working tree", "Changes to be committed" for the staged changes in the "staging area", and "Untracked files". In each section, it lists all the files that have been changed, i, e., files having status other than *unmodified*.

When a new file is created in the working tree, it is marked as *new* in working tree and shown as an untracked file. When the file change is staged, it is marked as *new* (*added*) in the staging area, and *unmodified* in working tree. When the file change is committed, it is marked as *unmodified* in both the working tree and staging area.



When a committed file is modified, it is marked as *modified* in the working tree and *unmodified* in the staging area. When the file change is staged, it is marked as *modified* in the staging area and *unmodified* in the working tree. When the file change is committed, it is marked as *unmodified* in both the working tree and staging area.



For example, made some changes to the file "Hello.java", and check the status again:

```
// Hello.java
public class Hello {
  public static void main(String[] args) {
    System.out.println("Hello, world from GIT!");
    System.out.println("Changes after First commit!");
  }
}
```

```
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
    modified:   Hello.java

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    Hello.class
no changes added to commit (use "git add" and/or "git commit -a")
```

The "Hello.java" is marked as *modified* in the working tree (under "Changes not staged for commit"), but *unmodified* in the staging area (not shown in "Changes to be committed").

You can inspect all the unstaged changes using "git diff" command (or "git diff <file>" for the specified file). It shows the file changes in the working tree since the last commit:

```
$ git diff
```

```
diff --git a/Hello.java b/Hello.java
index dc8d4cf..f4a4393 100644
--- a/Hello.java
+++ b/Hello.java
@@ -2,5 +2,6 @@
 public class Hello {
    public static void main(String[] args) {
       System.out.println("Hello, world from GIT!");
+      System.out.println("Changes after First commit!");
    }
 }
```

The older version (as of last commit) is marked as `---` and new one as `+++`. Each chunk of changes is delimited by "`@@ -<old-line-number>,<number-of-lines> +<new-line-number>,<number-of-lines> @@`". Added lines are marked as `+` and deleted as `-`. In the above output, older version (as of last commit) from line 2 for 5 lines and the modified version from line 2 for 6 lines are compared. One line (marked as +) is added.

Stage the changes of "Hello.java" by issuing the "git add <file>...":

```
$ git add Hello.java
```

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
    modified:   Hello.java
```

Untracked files:

(use "git add <file>..." to include in what will be committed)

Hello.class

Now, it is marked as *modified* in the staging area ("Changes to be committed"), but *unmodified* in the working tree (not shown in "Changes not staged for commit").

Now, the changes have been staged. Issuing an "git diff" to show the unstaged changes results in empty output.

You can inspect the staged change (in the staging area) via "git diff --staged" command:

// List all "unstaged" changes for all files (in the working tree)

$ **git diff**

// empty output - no unstaged change

// List all "staged" changes for all files (in the staging area)

$ **git diff --staged**

```
diff --git a/Hello.java b/Hello.java
index dc8d4cf..f4a4393 100644
--- a/Hello.java
+++ b/Hello.java
@@ -2,5 +2,6 @@
 public class Hello {
    public static void main(String[] args) {
       System.out.println("Hello, world from GIT!");
+      System.out.println("Changes after First commit!");
```

```
   }
 }
  // The "unstaged" changes are now "staged".
```

Commit ALL staged file changes via "git commit":

```
$ git commit -m "Second commit"
[master 96efc96] Second commit
 1 file changed, 1 insertion(+)
```

```
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
     Hello.class
nothing added to commit but untracked files present (use "git add" to track)
```

Once the file changes are committed, it is marked as *unmodified* in the staging area (not shown in "Changes to be committed").

Both "git diff" and "git diff --staged" return empty output, signalling there is no "unstaged" and "staged" changes.

The stage changes are cleared when the changes are committed; while the unstaged changes are cleared when the changes are staged.

Issue "git log" to list all the commits:

```
$ git log
commit 96efc96f0856846bc495aca2e4ea9f06b38317d1
Author: username <email>
Date:   Thu Nov 29 14:09:46 2012 +0800
```

Second commit

commit 858f3e71b95271ea320d45b69f44dc55cf1ff794

Author: *username* <*email*>

Date: Thu Nov 29 13:31:32 2012 +0800

First commit

Check the patches for the latest commit via "git log -p -1", with option *-n* to limit to the last *n* commit:

$ **git log -p -1**

commit 96efc96f0856846bc495aca2e4ea9f06b38317d1

Author: *username* <*email*>

Date: Thu Nov 29 14:09:46 2012 +0800

Second commit

diff --git a/Hello.java b/Hello.java

index dc8d4cf..ede8979 100644

--- a/Hello.java

+++ b/Hello.java

@@ -2,5 +2,6 @@

public class Hello {

public static void main(String[] args) {

System.out.println("Hello, world from GIT!");

+ System.out.println("Changes after First commit!");

}

}

I shall stress again Git tracks the "file changes" at each commit over the previous commit.

The. gitignore File

All the files in the Git directory are either *tracked* or *untracked*. To ignore files (such as .class, .o, .exe which could be reproduced from source) from being tracked and remove them from the *untracked* file list, create a ".gitignore" file in your project directory, which list the files to be ignored, as follows:

```
#. gitignore

# Java class files
*.class

# Executable files
*.exe

# Object and archive files
# Can use regular expression, e.g., [oa] matches either o or a
*. [oa]

# temp sub-directory (ended with a directory separator)
temp/
```

There should NOT be any trailing comments for filename. You can use regexe for matching the filename/pathname patterns, e.g. [oa] denotes either o or a. You can override the rules by

using the inverted pattern (!), e.g., Adding! hello.exe includes the hello.exe although *.exe are excluded.

Now, issue a "git status" command to check the untracked files.

$ **git status**

On branch master

Untracked files:

(use "git add <file>..." to include in what will be committed)

. gitignore

nothing added to commit but untracked files present (use "git add" to track)

Now, "Hello. Class" is not shown in "Untracked files".

Typically, we also track and commit the. gitignore file.

$ **git add. gitignore**

$ **git status**

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file: .gitignore

$ **git commit -m "Added .gitignore"**

[master 711ef4f] Added .gitignore

1 file changed, 14 insertions(+)

create mode 100644 .gitignore

$ **git status**

On branch master

nothing to commit, working directory clean

Setting up Remote Repo

Sign up for a GIT host, such as GitHub https://github.com/signup/free (Unlimited for public projects; fee for private projects); or Bitbucket @ https://bitbucket.org/ (Unlimited users for public projects; 5 free users for private projects; Unlimited for Academic Plan); among others.

Login to the GIT host. Create a new remote repo called "test".

On your local repo (let's continue to work on our "hello-git" project), set up the remote repo's *name* and *URL* via "git remote add <remote-name> <remote-url>" command.
By convention, we shall name our remote repo as "origin". You can find the URL of a remote repo from the Git host. The URL may take the form of HTTPS or SSH. Use HTTPS for simplicity.

// Change directory to your local repo's working directory

$ **cd /*path-to*/hello-git**

// Add a remote repo called "origin" via "git remote add <remote-name> <remote-url>"

// For examples,

$ **git remote add origin https://github.com/*your-username*/test.git**
// for GitHub

$ **git remote add origin https://*username*@bitbucket.org/*your-username*/test.git** // for Bitbucket

You can list all the remote names and their corresponding URLs via "git remote -v", for example,

// List all remote names and their corresponding URLs

$ git remote v

origin https://github.com/your-username/test.git (fetch)

origin https://github.com/your-username/test.git (push)

Now, you can manage the remote connection, using a simple *name* instead of the complex URL.

Push the commits from the local repo to the remote repo via "git push -u <remote-name> <local-branch-name>".
By convention, the main branch of our local repo is called "master" (as seen from the earlier "git status" output). We shall discuss "branch" later.

// Push all commits of the branch "master" to remote repo "origin"

$ git push origin master

Username for 'https://github.com': ******

Password for 'https://*your-username*@github.com': *******

Counting objects: 10, done.

Delta compression using up to 8 threads.

Compressing objects: 100% (10/10), done.

Writing objects: 100% (10/10), 1.13 KiB | 0 bytes/s, done.

Total 10 (delta 1), reused 0 (delta 0)

To https://github.com/*your-username*/test.git

 * [new branch]   master -> master

Branch master set up to track remote branch master from origin.

Login to the GIT host and select the remote repo "test", you shall find all the committed files.

On your local system, make some change (e.g., on "Hello.java"); stage and commit the changes on the local repo; and push it to the remote. This is known as the "Edit/Stage/Commit/Push" cycle.

// Hello.java

```
public class Hello {

   public static void main(String[] args) {
```

```
    System.out.println("Hello, world from GIT!");
    System.out.println("Changes after First commit!");
    System.out.println("Changes after Pushing to remote!");
  }
}
```

$ **git status**

```
On branch master
Your branch is up-to-date with 'origin/master'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working dire
    modified:   Hello.java
no changes added to commit (use "git add" and/or "git commit -a")
```

// Stage file changes

$ **git add *.java**

$ **git status**

```
On branch master
Your branch is up-to-date with 'origin/master'.

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
      modified:   Hello.java
```

// Commit all staged file changes

$ **git commit -m "Third commit"**

[master 744307e] Third commit

1 file changed, 1 insertion(+)

// Push the commits on local master branch to remote

$ **git push origin master**

Username for 'https://github.com': ******

Password for 'https://*username*@github.com': ******

Counting objects: 5, done.

Delta compression using up to 8 threads.

Compressing objects: 100% (3/3), done.

Writing objects: 100% (3/3), 377 bytes | 0 bytes/s, done.

Total 3 (delta 1), reused 0 (delta 0)

To https://github.com/*your-username*/test.git

711ef4f..744307e master -> master

Again, login to the remote to check the committed files.

Cloning a Project from a Remote Repo (git clone <remote-URL>)

As mentioned earlier, you can start a local GIT repo either running "git init" on your own project, or "git clone <remote-url>" to copy from an existing project.

Anyone having read access to your remote repo can *clone* your project. You can also *clone* any project in any *public* remote repo.

The "git clone <remote-url>" initializes a local repo and copies all files into the working tree. You can find the URL of a remote repo from the Git host.

// SYNTAX

// ======

$ **git clone <remote-url>**

// <url>: can be https (recommended), ssh or file.

// Clone the project UNDER the current directory

// The name of the "working directory" is the same as the remote project name

$ **git clone <remote-url> <working-directory-name>**

// Clone UNDER current directory, use the given "working directory" name

// EXAMPLES

// ========

// Change directory (cd) to the "parent" directory of the project directory

$ **cd *path-to-parent-of-the-working-directory***

// Clone our remote repo "test" into a new working directory called "hello-git-cloned"

$ **git clone https://github.com/*your-username*/test.git hello-git-cloned**

Cloning into 'hello-git-cloned'...

remote: Counting objects: 13, done.

remote: Compressing objects: 100% (11/11), done.

remote: Total 13 (delta 2), reused 13 (delta 2)

Unpacking objects: 100% (13/13), done.

Checking connectivity... done.

// Verify

$ **cd hello-git-cloned**

$ **ls -a**

. .. .git .gitignore Hello.java README.md

$ **git status**

On branch master

Your branch is up-to-date with 'origin/master'.

nothing to commit, working directory clean

The "git clone" automatically creates a remote name called "origin" mapped to the cloned remote-URL. You can check via "git remote -v":

// List all the remote names

$ **git remote -v**

origin https://github.com/*your-username*/test.git (fetch)

origin https://github.com/*your-username*/test.git (push)

Summary of Basic "Edit/Stage/Commit/Push" Cycle

// Edit (Create, Modified, Rename, Delete) files,

// which produces "unstaged" file changes.

// Stage file changes, which produces "Staged" file changes

$ **git add <file>** // for new and modified files

$ **git rm <file>** // for deleted files

$ **git mv <old-file-name> <new-file-name>** // for renamed file

// Commit (ALL staged file changes)

$ **git commit -m "message"**

// Push

$ **git push <remote-name> <local-branch-name>**

OR,

// Stage ALL files with changes

$ **git add -A** // OR, '**git add --all**'

$ **git commit -m "message"**

$ **git push**

OR,

// Add All and Commit in one command

$ **git commit -a -m "message"**

$ **git push**

3.5 More on Staged and Unstaged Changes

If you modify a file, stage the changes and modify the file again, there will be *staged changes* and *unstaged changes* for that file.

For example, let's continue the "hello-git" project. Add one more line to "README.md" and stage the changes:

// README.md

This is the README file for the Hello-world project.

Make some changes and staged.

$ **git status**

On branch master

Your branch is up-to-date with 'origin/master'.

Changes not staged for commit:

modified: README.md

$ **git add README.md**

$ **git status**

On branch master

Your branch is up-to-date with 'origin/master'.

Changes to be committed:

modified: README.md

Before the changes are committed, suppose we modify the file again:

// README.md

This is the README file for the Hello-world project.

Make some changes and staged.

Make more changes before the previous changes are committed.

$ **git status**

On branch master

Your branch is up-to-date with 'origin/master'.

Changes to be committed:

modified: README.md

Changes not staged for commit:

modified: README.md

// Now, "README.md" has both unstaged and staged changes.

// Show the staged changes

$ **git diff --staged**

```
diff --git a/README.md b/README.md
index 9565113..b2e9afb 100644
--- a/README.md
+++ b/README.md
@@ -1,2 +1,3 @@
 // README.md
 This is the README file for the Hello-world project.
+Make some changes and staged.
```

// Show the unstaged changes

$ **git diff**

```
diff --git a/README.md b/README.md
index b2e9afb..ca6622a 100644
--- a/README.md
+++ b/README.md
@@ -1,3 +1,4 @@
 // README.md
 This is the README file for the Hello-world project.
```

Make some changes and staged.

+Make more changes before the previous changes are committed.

// Stage the changes

$ **git add README.md**

$ **git status**

On branch master

Your branch is up-to-date with 'origin/master'.

Changes to be committed:

modified: README.md

// Show staged changes

$ **git diff --staged**

diff --git a/README.md b/README.md

index 9565113..ca6622a 100644

--- a/README.md

+++ b/README.md

@@ -1,2 +1,4 @@

// README.md

This is the README file for the Hello-world project.

+Make some changes and staged.

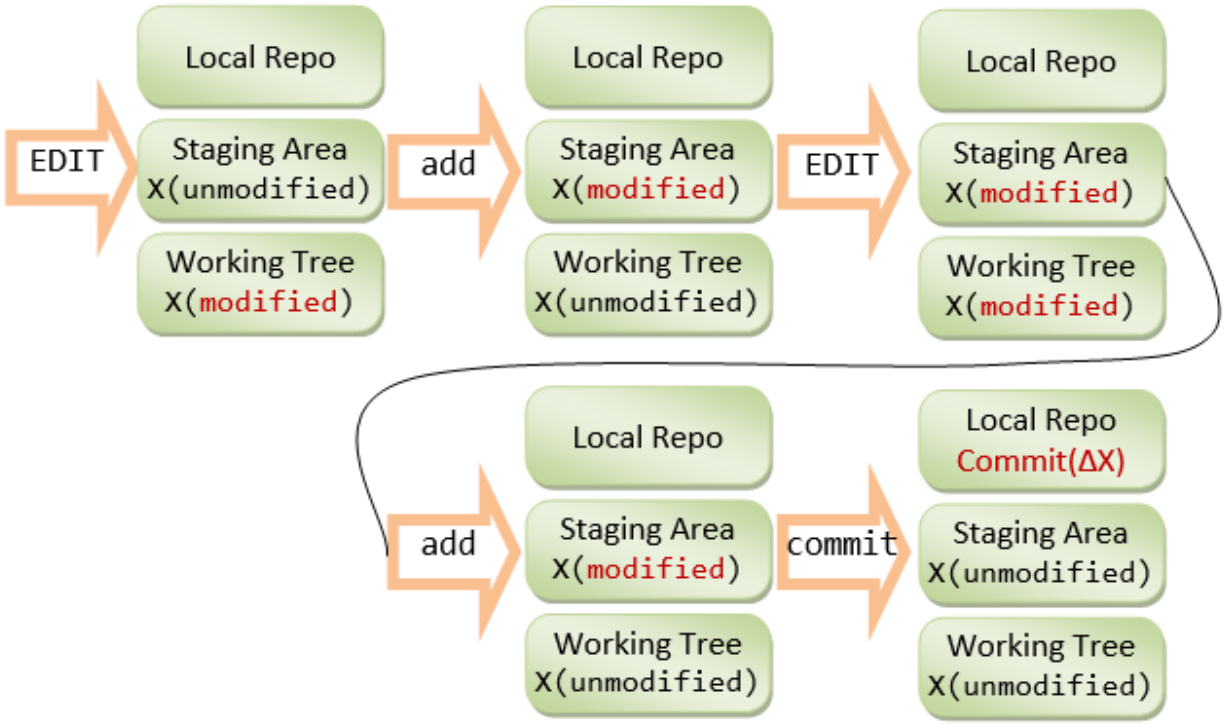
+Make more changes before the previous changes are committed.

// Commit the staged changes

$ git commit -m "Unstaged vs. Staged Changes"

[master a44199b] Unstaged vs. Staged Changes

1 file changed, 2 insertions(+), 0 deletion(-)



Take note that the stage changes are cleared when the changes are committed; while the unstaged changes are cleared when the changes are staged.

For convenience, you can also use the "git-gui" tool to view the unstaged and staged changes.

Git GUI Tools

Git-GUI (Windows)

For convenience, Git provides a GUI tool, called **git-gui**, which can be used to perform all tasks and view the commit log graphically.

Install "Git-Gui".

To run the **git-gui**, you can right-click on the project folder and choose "Git Gui"; or launch the Git-bash shell and run "**git gui**" command.

To view the log, choose "Repository" ⇒ "Visualize master's history", which launches the "gitk". You can view the details of each commit.

You can also view each of the file via "Repository" ⇒ "Browse master's Files" ⇒ Select a file.

Git-gui is bundled with Git. To launch git-gui, right click on the working directory and choose "git gui",or run "git gui" command on the Git-Bash shell.

Tagging

Tag (or label) can be used to tag a specific commit as being important, for example, to mark a particular release. The release is often marked in this format: *version-number.release-no.modificaton-no* (e.g., v1.1.5) or or *version-number.release-no.upgrade-no_modificaton-no* (e.g., v1.7.0_26).

I recommend that you commit your code and push it to the remote repo as often as needed (e.g., daily), to BACKUP your code. When you code reaches a stable point (in turn of functionality), create a tag to mark the commit, which can then be used for CHECKOUT, if you need to show your code to others.

Listing Tags (git tag)

To list the existing tags, use "git tag" command.

Types of Tags - Lightweight Tags and Annotated Tags

There are two kinds of tags: lightweight tag and annotated tag. Lightweight tag is simply a pointer to a commit. Annotated tag contains annotations (meta-data) and can be digitally signed and verified.

Creating an Annotated Tag (git tag -a <tag-name> -m <message>)

To create an annotated tag at the latest commit, use "git tag -a <tag-name> -m <message>", where -a option specifies annotation tag having meta-data. For example,

```
$ git tag -a v1.0.0 -m "First production system"
```

```
// List all tags
```

```
$ git tag
```

v1.0.0

// Show tag details

$ **git show v1.0.0**

// Show the commit point and working tree

To create a tag for an earlier commit, you need to find out the commit's name (first seven character hash code) (via "git log"), and issue "git tag -a <tag-name> -m <message> <commit-name>". For example,

$ **git log**

.....

commit 7e7cb40a9340691e2b16a041f7185cee5f7ba92e

.....

Commit 3

$ **git tag -a "v0.9.0" -m "Last pre-production release" 7e7cb40**

// List all tags

$ **git tag**

v0.9.0

v1.0.0

// Show details of a tag

$ **git show v0.9.0**

.....

[Branching/Merging]

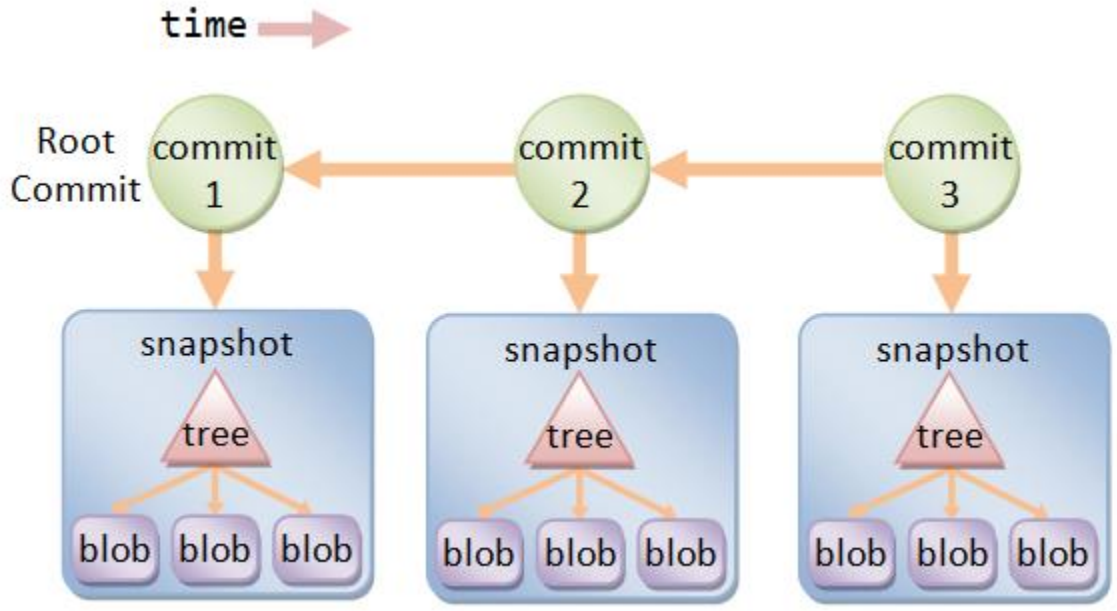
Git's Data Structures

Git has two primary data structures:

an immutable, append-only *object database* (or *local repo*) that stores all the commits and file contents.

a mutable *staging area* (or *index*, or *cache*) that caches the staged information.

The *staging area* serves as the connection between *object database* and *working tree* (as shown in the storage model diagram). It serves to avoid volatility and allows you to stage ALL the file changes before issuing a commit, instead of committing individual file change. Changes to files that have been explicitly added to the index (staging area) via "git add <file>" are called *staged changes*. Changes that have not been added are called *unstaged changes*. Staged and unstaged changes can co-exist. Performing a commit copies the statged changes into object database (local repo) and clears the index. The unstaged changes remain in working tree.



The object database contains these objects:

Each version of a file is represented by a *blob* (binary large object - a file that can contain any data: binaries or characters). A blob holds the file data only, without any metadata - not even the filename.

A *snapshot* of the working tree is represented by a *tree* object, which links the blobs and sub-trees for sub-directories.

A *commit* object points to a tree object, i.e., the snapshot of the working tree at the point the commit was created. It holds metadata such as timestamp, log message, author's and committer's username and email. It also references its parent commit(s), except the root commit which has no parent. A normal commit has one parent; a merge commit could have multiple parents. A commit, where new branch is created, has more than one children. By referencing through the chain of parent commit(s), you can discover the history of the project.

Each object is identified (or named) by a 160-bit (or 40 hex-digit) SHA-1 hash value of its contents (i.e., a content-addressable name). Any tiny change to the contents produces a different hash value, resulted in a different object. Typically, we use the first 7 hex-digit prefix to refer to an object, as long as there is no ambiguity.

There are two ways to refer to a particular commit: via a branch or a tag.

A branch is a mobile reference of commit. It moves forward whenever commit is made on that branch.

A tag (like a label) marks a particular commit. Tag is often used for marking the releases.

Branching

Branching allows you and your team members to work on different aspects of the software *concurrently* (on so-called feature branches), and merge into the master branch as and when they complete. Branching is the most important feature in a *concurrent* version control system.

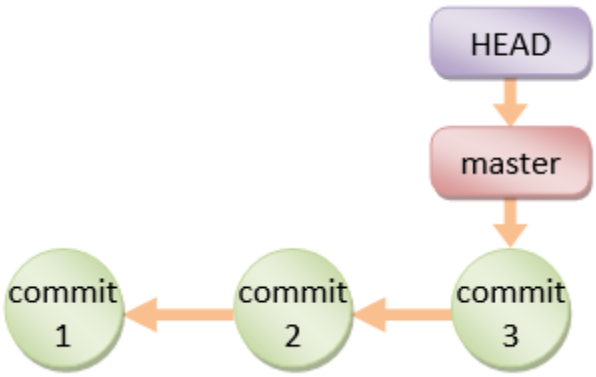
A branch in Git is a lightweight movable pointer to one of the commits. For the initial commit, Git assigns the default branch name called master and sets the master branch pointer at the initial commit. As you make further commits on the master branch, the master branch pointer move forward accordingly. Git also uses a special pointer called HEAD to keep track of the branch that you are currently working on. The HEAD always refers to the latest commit on the current branch. Whenever you switch branch, the HEAD also switches to the latest commit on the branch switched.

Example

For example, let's create a Git-managed project called git_branch_test with only the a single-line README.md file:

This is the README. My email is xxx@somewhere.com

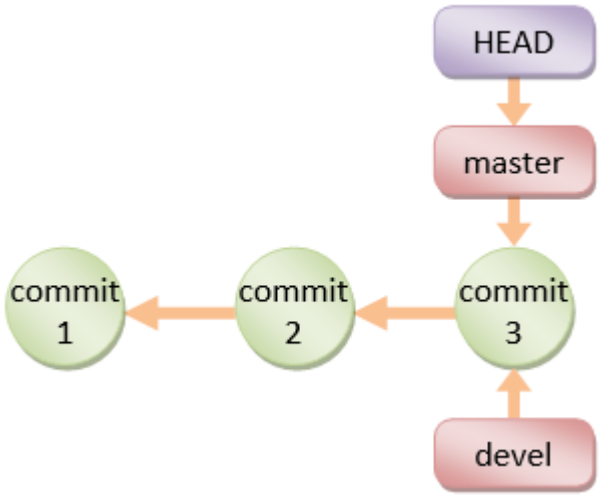
```
$ git init
$ git add README.md
$ git commit -m "Commit 1"

// Append a line in README.md: This line is added after Commit 1
$ git status
$ git add README.md
$ git commit -m "Commit 2"

// Append a line in README.md: This line is added after Commit 2
$ git status
$ git add README.md
$ git commit -m "Commit 3"

// Show all the commits (oneline each)
$ git log --oneline
44fdf4c Commit 3
51f6827 Commit 2
fbed70e Commit 1
```



Creating a new Branch (git branch <branch-name>)

You can create a new branch via "git branch <branch-name>" command. When you create a new branch (says devel, or development), Git creates a new *branch pointer* for the branch devel, pointing initially at the latest commit on the current branch master.

$ git branch devel



Take note that when you create a new branch, the HEAD pointer is still pointing at the current branch.

Branch Names Convention

master branch: the production branch with tags for the various releases.

development (or next or devel) branch: developmental branch, to be merged into master if and when completes.

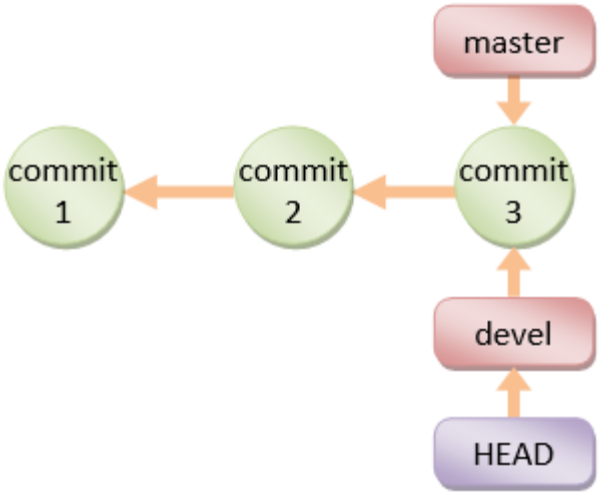
topics branch: a short-live branch for a specific topics, such as introducing a feature (for the devel branch) or fixing a bug (for the master branch).

Switching to a Branch (git checkout <branch-name>)

Git uses a special pointer called HEAD to keep track of the branch that you are working on. The "git branch <branch-name>" command simply create a branch, but does not switch to the new branch. To switch to a branch, use "git checkout <branch-name>" command. The HEAD pointer will be pointing at the switched branch (e.g., devel).

$ git checkout devel

Switched to branch 'devel'



Alternatively, you can use "git checkout -b <branch-name>" to create a new branch and switch into the new branch.

If you switch to a branch and make changes and commit. The HEAD pointer moves forward in that branch.

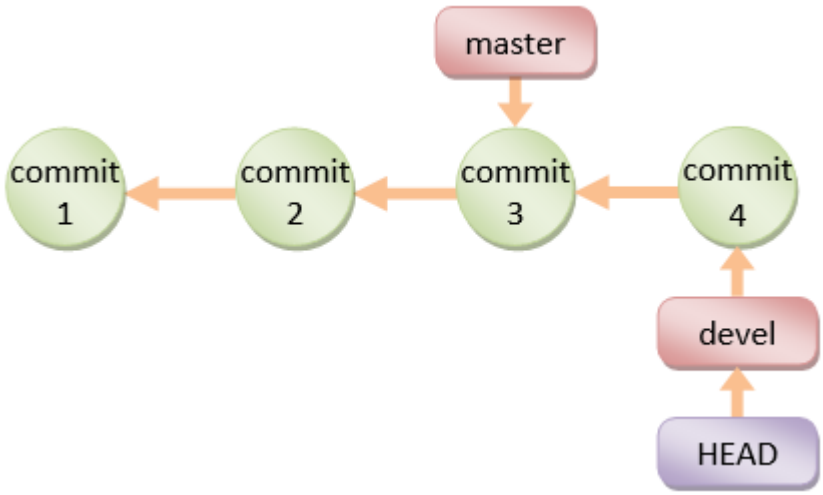
// Append a line in README.md: This line is added on devel branch after Commit 3

$ git status // NOTE "On branch devel"

$ git add README.md

$ git commit -m "Commit 4"

[devel c9b88d9] Commit 4

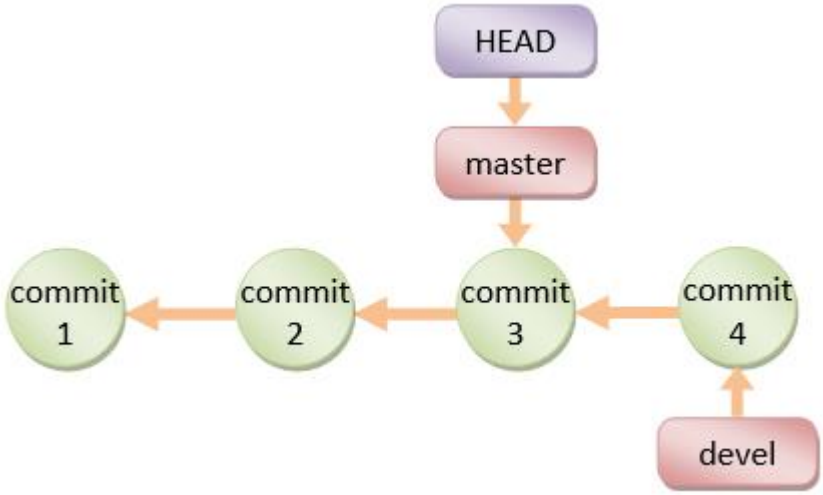


You can switch back to the master branch via "git checkout master". The HEAD pointer moves back to the last commit of the master branch, and the working directory is *rewinded* back to the latest commit on the master branch.

$ **git checkout master**

Switched to branch 'master'

// Check the content of the README.md, which is reminded back to Commit 3



If you continue to work on the master branch and commit, the HEAD pointer moves forward on the master branch. The two branches now *diverge*.

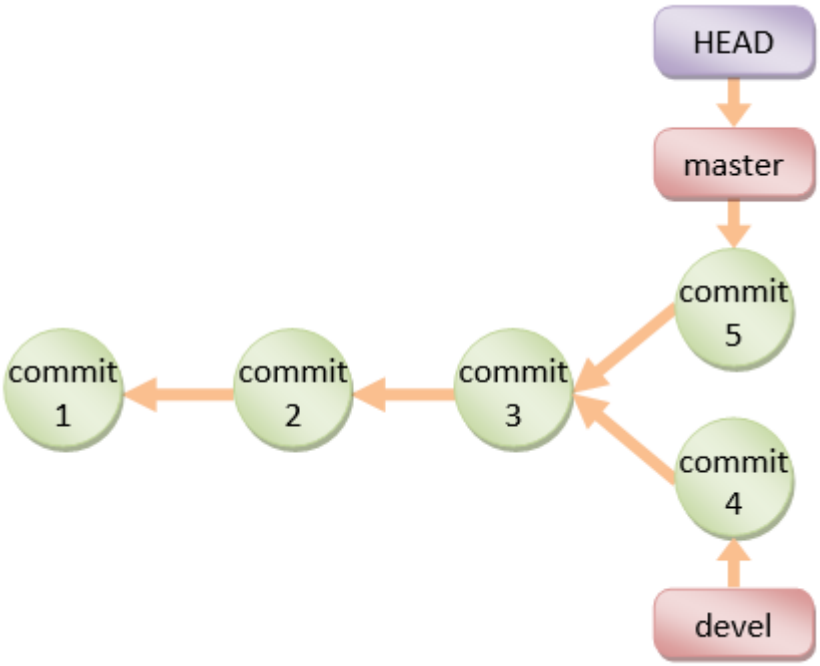
// Append a line in README.md: This line is added on master branch after Commit 4

$ **git status** // NOTE "On branch master"

$ **git add README.md**

$ **git commit -m "Commit 5"**

[master 6464eb8] Commit 5



If you check out the devel branch, the file contents will be rewinded back to Commit-4.

$ **git checkout devel**

// Check file contents

Merging Two Branches (git merge <branch-name>)

To merge two branches, says master and devel, check out the first branch, e,g, master, (via "git checkout <branch-name>") and merge with another branch, e.g., devel, via command "git merge <branch-name>".

Fast-Forward Linear Merge

If the branch to be merged is a direct descendant, Git performs *fast forward* by moving the HEAD pointer forward. For example, suppose that you are currently working on the devel branch at commit-4, and the master branch's latest commit is at commit-3:

$ **git checkout master**

// Let discard the Commit-5 totally and rewind to commit-3 on master branch

// This is solely for illustration!!! Do this with great care!!!

$ **git reset --hard HEAD~1**

HEAD is now at 7e7cb40 Commit 3

// HEAD~1 moves the HEAD pointer back by one commit (-1)

// --hard also resets the working tree

// Check the file contents

$ **git merge devel**

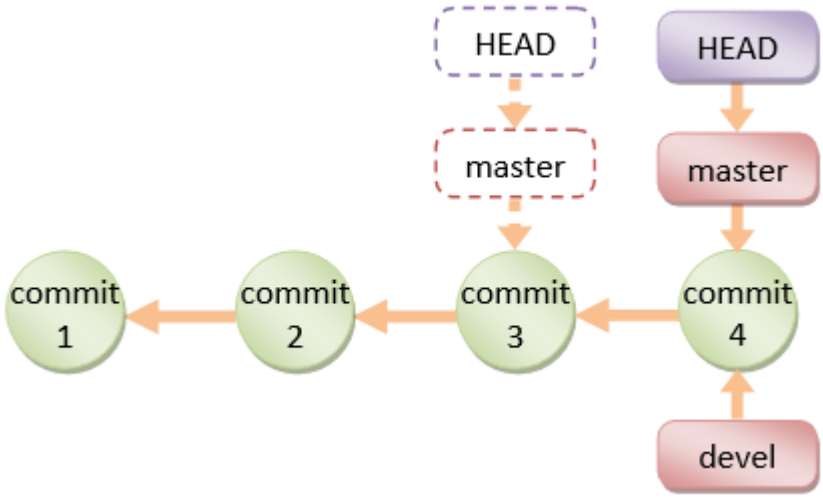
Updating 7e7cb40..4848c7b

Fast-forward

README.md | 1 +

1 file changed, 1 insertion(+)

// Check the file contents



Take note that no *new* commit is created.

3-Way Merge

If the two branches are diverged, git automatically searches for the *common ancestor commit* and performs a 3-way merge. If there is no conflict, a new commit will be created.

If git detects a conflict, it will pause the merge and issue a merge conflict and ask you to resolve the conflict manually. The file is marked as *unmerged*. You can issue "git status" to check the unmerged files, study the details of the conflict, and decide which way to resolve the conflict. Once the conflict is resolve, stage the file (via "git add <file>"). Finally, run a "git commit" to finalize the 3-way merge (the same Edit/Stage/Commit cycle).

$ git checkout master

// undo the Commit-4, back to Commit-3

$ git reset --hard HEAD~1

HEAD is now at 7e7cb40 Commit 3

// Change the email to abc@abc.com

$ git add README.md

$ git commit -m "Commit 5"

$ **git checkout devel**

// undo the Commit-4, back to Commit-3

$ **git reset --hard HEAD~1**

// Change the email to xyz@xyz.com to trigger conflict

$ **git add README.md**

$ **git commit -m "Commit 4"**

// Let's do a 3-way merge with conflict

$ **git checkout master**

$ **git merge devel**

Auto-merging README.md

CONFLICT (content): Merge conflict in README.md

Automatic merge failed; fix conflicts and then commit the result.

$ **git status**

On branch master

You have unmerged paths.

(fix conflicts and run "git commit")

#

Unmerged paths:

(use "git add <file>..." to mark resolution)

both modified: README.md

no changes added to commit (use "git add" and/or "git commit -a")

The conflict file is marked as follows (in "git status"):

<<<<<<< HEAD

This is the README. My email is abc@abc.com

=======

This is the README. My email is xyz@xyz.com

>>>>>>> devel

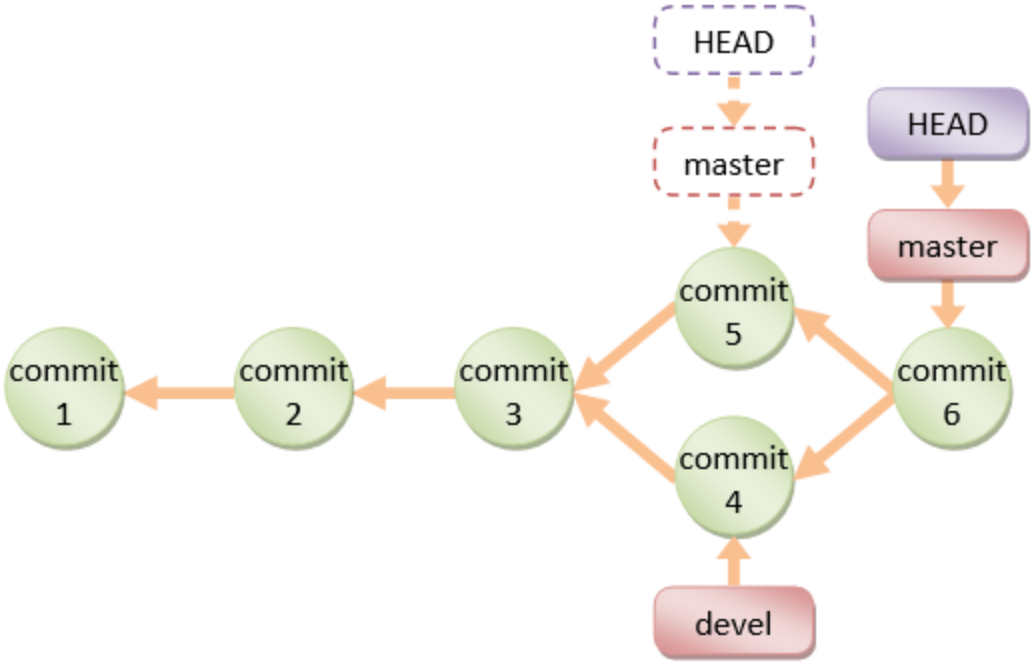
This line is added after Commit 1

This line is added after Commit 2

You need to manually decide which way to take, or you could discard both by setting the email to zzz@nowhere.com.

$ **git add README.md**

$ **git commit -m "Commit 6"**



Take note that In a 3-way merge, a new commit will be created in the process (unlike fast-forward merge).

Deleting a Merged Branch (git branch -d <branch-name>)

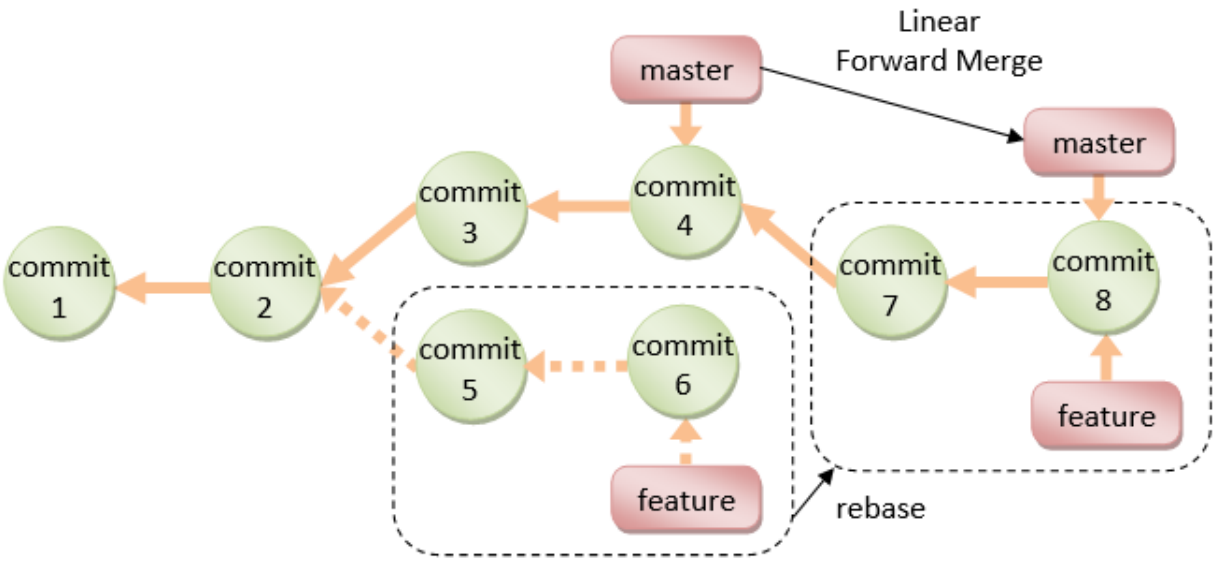
The merged branch (e.g., devel) is no longer needed. You can delete it via "git branch -d <branch-name>".

```
$ git branch -d devel
```

Deleted branch devel (was a20f002).

```
// Create the development branch again at the latest commit
$ git branch devel
```

Rebasing Branch (git rebase)



The primary purpose for rebasing is to maintain a linear project history. For example, if you checkout a devel branch and work on commit-5 and commit-6, instead of doing a 3-way merge into the master branch and subsequently remove the devel branch, you can rebase the commit-5 and commit-6, on commit-4, and perform a linear forward merge to maintain all the project history. New commits (7 and 8) will be created for the rebased commit (5 and 6).

The syntax is:

```
// SYNTAX
$ git rebase <base-name>
  // <base-name> could be any kind of commit reference
```

```
  // (such as an commit-name, a branch name, a tag,
  // or a relative reference to HEAD).
```

Examples:

```
// Start a new feature branch from the current master
$ git checkout -b feature master
// Edit/Stage/Commit changes to feature branch

// Need to work on a fix on the master
$ git checkout -b hotfix master
// Edit/Stage/Commit changes to hotfix branch
// Merge hotfix into master
$ git checkout master
$ git merge hotfix
// Delete hotfix branch
$ git branch -d hotfix

// Rebase feature branch on master branch
//  to maintain a linear history
$ git checkout feature
$ git rebase master
// Now, linear merge
$ git checkout master
$ git merge feature
```

Amend the Last Commit (git commit --amend)

If you make a commit but want to change the commit message or adding more changes, you may amend the recent commit (instead of creating new commit) via command "git commit --amend"):

$ **git commit --amend -m "message"**

For example,

// Do a commit

$ **git commit -m "added login menu"**

// Realize that you have not staged some files.

// Amend the commit

$ **git add morefile**

$ **git commit --amend**

// You could modify the commit message here

More on "git checkout" and Detached HEAD

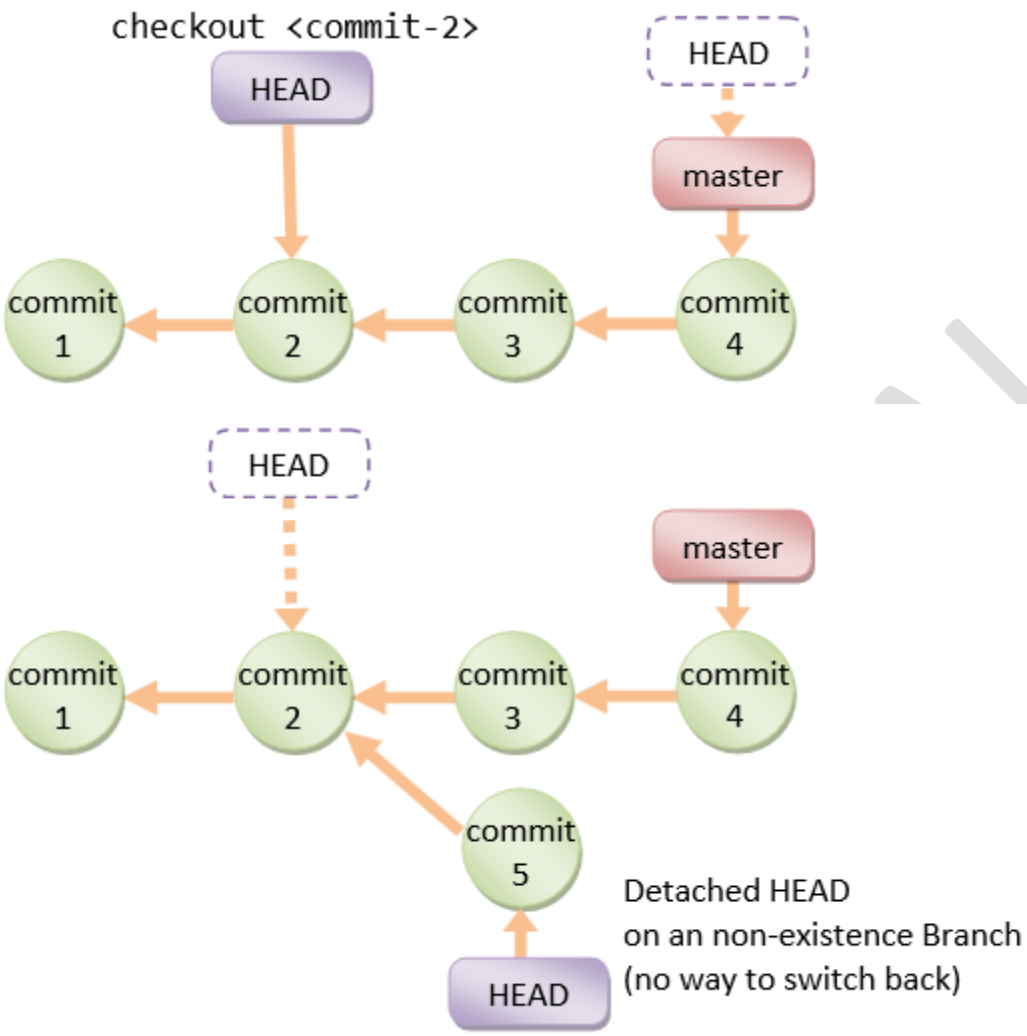
"git checkout" can be used to checkout a branch, a commit, or files. The syntaxes are:

$ **git checkout <branch-name>**

$ **git checkout <commit-name>**

$ **git checkout <commit-name> <filename>**

When you checkout a commit, Git switches into so-called "Detached HEAD" state, i.e., the HEAD detached from the tip of a branch. Suppose that you continue to work on the detached HEAD on commit-5, and wish to merge the commit-5 back to master. You checkout the master branch, but there is no branch name for your to reference the commit-5!!!



In Summary, you can use "git checkout <commit-name>" to inspect a commit. BUT you should always work on a branch, NOT on a detached HEAD.

More on "git reset" and "git reset --hard"

$ **git reset <file>**

```
// Unstage the changes of <file> from staging area,
//   not affecting the working tree.
```

$ **git reset**

```
// Reset the staging area
// Remove all changes (of all files) from staging area,
//   not affecting the working tree.
```

```
$ git reset --hard
// Reset the staging area and working tree to match the
//   recent commit  (i.e., discard all changes since the
//   last commit).
```

```
$ git reset <commit-name>
// Move the HEAD of current branch to the given commit,
//   not affecting the working tree.
```

```
$ git reset --hard <commit-name>
// Reset both staging area and working tree to the given
//   commit, i.e., discard all changes after that commit.
```

git revert <commit-name>

The "git revert" undoes a commit. But, instead of removing the commit from the project history, it undos the changes introduced by the commit and appends a new commit with the resulting content. This prevents Git from losing history. "git revert" is a safer way comparing with "git reset".

```
// SYNTAX
$ git revert <commit-name>
```

// EXAMPLE

[TODO] example and diagram

Summary of Work Flows

Setting up GIT and "Edit/Stage/Commit/Push" Cycle

Step 1: Install GIT.

For Windows and Mac, download the installer from http://git-scm.com/downloads and run the downloaded installer.

For Ubuntu, issue command "sudo apt-get install git".

For Windows, use "git-bash" command shell provided by Windows installer to issue command. For Mac/Ubuntu, use "Terminal".

Step 2: Configuring GIT:

// Setup your username and email to be used in labeling commits

$ **git config --global user.email "*your-email@yourmail.com*"**

$ **git config --global user.name "*your-name*"**

Step 3: Set up GIT repo for a project. For example, we have a project called "olas1.1" located at "/usr/local/olas/olas1.1".

$ **cd /usr/local/olas/olas1.1**

// Initialize the GIT repo

$ **git init**

$ **ls -al**

// Check for ".git" directory

Create a "README.md" (or "README.textile" if you are using Eclipse's WikiText in "textile" markup) under your project directory to describe the project.

Step 4: Start "Edit/Stage/Commit/Push" cycles.

Create/Modify files. Stage files into the staging area via "git add <file>".

```
// Check the status
```

$ **git status**

```
.....
```

```
// Add files into repo
```

$ **git add README.md**

$ **git add www**

```
.....
```

```
// Check the status
```

$ **git status**

```
.....
```

Step 5: Create a ".gitignore" (in the project base directory) to exclude folders/files from being tracked by GIT. Check your "git status" output to decide which folders/files to be ignored.

For example,

```
# ignore files and directories beginning with dot
.*

# ignore directories beginning with dot (a directory ends with a slash)
.*/
```

ignore these files and directories

www/test/

www/.*

www/.*/

The trailing slash indicate directory (and its sub-directories and files).

If you want the "**.gitignore**" to be tracked (which is in the ignore list):

$ **git add -f .gitignore**

// -f to override the .gitignore

Step 6: Commit.

$ **git status**

.....

// Commit with a message

$ **git commit -m "Initial Commit"**

.....

$ **git status**

.....

Step 7: Push to the Remote Repo (for backup, version control, and collaboration).

You need to first create a repo (says olas) in a remote GIT host, such as GitHub or BitBucket. Take note of the remote repo URL, e.g., https://*username*@*hostname*.org/*username*/olas.git.

$ **cd */path-to/local-repo***

// Add a remote repo name called "origin" mapped to the remote URL

$ **git remote add origin https://*hostname*/*username*/olas.git**

// Push the "master" branch to the remote "origin"

// "master" is the default branch name of your local repo after init.

$ **git push origin master**

Check the remote repo for the files committed.

Step 8: Work on the source files, make changes, commit and push to remote repo.

// Check the files modified

$ **git status**

.....

// Stage for commit the modified files

$ **git add ....**

.....

// Commit (with a message)

$ **git commit -m "*commit-message*"**

// Push to remote repo

$ **git push origin master**

Step 9: Create a "tag" (for version number).

// Tag a version number to the current commit

$ **git tag -a v1.1 -m "Version 1.1"**

// -a to create an annotated tag, -m to provide a message

// Display all tags

$ **git tag**

.....

// Push the tags to remote repo

// ("git push -u origin master" does not push the tags)

$ **git push origin --tags**

Branch and Merge Workflow

It is a good practice to freeze the "master" branch for production; and work on a development branch (says "devel") instead. You may often spawn a branch to fix a bug in the production.

// Create a branch called "devel" and checkout.

// The "devel" is initially synchronized with the "master" branch.

$ **git checkout -b devel**

// same as:

// $ **git branch devel**

// $ **git checkout**

// Edit/Stage/Commit

$ **git add <file>**

$ **git commit -m "*commit-message*"**

// To merge the "devel" into the production "master" branch

$ **git checkout master**

```
$ git merge devel

// Push both branches to remote repo
$ git push origin master devel

// Checkout the "devel" branch and continue...
$ git checkout devel
  // Edit/Stage/Commit/Push

// Need to fix a bug in production (in "master" branch)
$ git checkout master
// Spawn a "fix" branch to fix the bug, and merge with the "master" branch

// To remove the "devel" branch (if the branch is out-of-sync)
$ git branch -d devel
// To re-create the "devel" branch
$ git checkout -b devel
```

Viewing the Commit Graph (gitk)

You can use the "git-gui" "gitk" tool to view the commit graph.

To run the git-gui, you can right-click on the project folder and choose "Git Gui"; or launch the Git-bash shell and run "git gui" command.

To view the commit graph, choose "Repository" ⇒ "Visualize master's history", which launches the "gitk". You can view the details of each commit.

Collaboration

Reference: https://www.atlassian.com/git/tutorials/making-a-pull-request/how-it-works.

Synchronizing Remote and Local: Fetch/Merge, Pull and Push

Setup up a remote repo (revision)

As described earlier, you can use "git remote" command to set up a "remote name", mapped to the URL of a remote repo.

// Add a new "remote name" maps to the URL of a remote repo

$ **git remote add <remote-name> <remote-url>**

// For example,

$ **git remote add origin https://*hostname*/*username*/*project-name*.git**

// Define a new remote name "origin" mapping to the given URL

// List all the remote names

$ **git remote -v**

// Delete a remote name

$ **git remote rm <remote-name>**

// Rename a remote name

$ **git remote rename <old-remote-name> <new-remote-name>**

Cloning a Remote Repo (revision)

$ **git clone <remote-url>**

// Init a GIT local repo and copy all objects from the remote repo

$ **git clone <remote-url> <working-directory-name>**

// Use the working-directory-name instead of default to project name

Whenever you *clone* a remote repo using command "git clone <remote-url>", a remote name called "origin" is automatically added and mapped to <remote-url>.

Fetch/Merge Changes from remote (git fetch/merge)

The "git fetch" command imports commits from a remote repo to your local repo, without updating your local working tree. This gives you a chance to review changes before updating (merging into) your working tree. The fetched objects are stored in remote branches, that are differentiated from the local branches.

$ **cd */path-to/working-directory***

$ **git fetch <remote-name>**

// Fetch ALL branches from the remote repo to your local repo

$ **git fetch <remote-name> <branch-name>**

// Fetch the specific branch from the remote repo to your local repo

// List the local branches

$ **git branch**

* master

devel

// * indicates current branch

// List the remote branches

$ **git branch -r**

origin/master

origin/devel

// You can checkout a remote branch to inspect the files/commits.

// But this put you into "Detached HEAD" state, which prevent you

// from updating the remote branch.

// You can merge the fetched changes into local repo

$ **git checkout master**

// Switch to "master" branch of local repo

$ **git merge origin/master**

// Merge the fetched changes from stored remote branch to local

git pull

As a short hand, "git pull" combines "git fetch" and "git merge" into one command, for convenience.

$ **git pull <remote-name>**

// Fetch the remote's copy of the current branch and merge it

// into the local repo immediately, i.e., update the working tree

// Same as

$ **git fetch <remote-name> <current-branch-name>**

```
$ git merge <remote-name> <current-branch-name>
```

```
$ git pull --rebase <remote-name>
  // linearize local changes after the remote branch.
```

The "git pull" is an easy way to *synchronize* your local repo with origin's (or upstream) changes (for a specific branch).

Pushing to Remote Repo (revision)

The "git push <remote-name> <branch-name>" is the counterpart of "git fetch", which exports commits from local repo to remote repo.

```
$ git push <remote-name> <branch-name>
  // Push the specific branch of the local repo
```

```
$ git push <remote-name> --all
  // Push all branches of the local repo
```

```
$ git push <remote-name> --tag
  // Push all tags
  // "git push" does not push tags
```

```
$ git push -u <remote-name> <branch-name>
  // Save the remote-name and branch-name as the
  // reference (or current) remote-name and branch-name.
  // Subsequent "git push" without argument will use these references.
```

"Fork" and "Pull Request"

"Fork" and "Pull Request" are features provided by GIT hosts (such as GitHub and BitBucket):

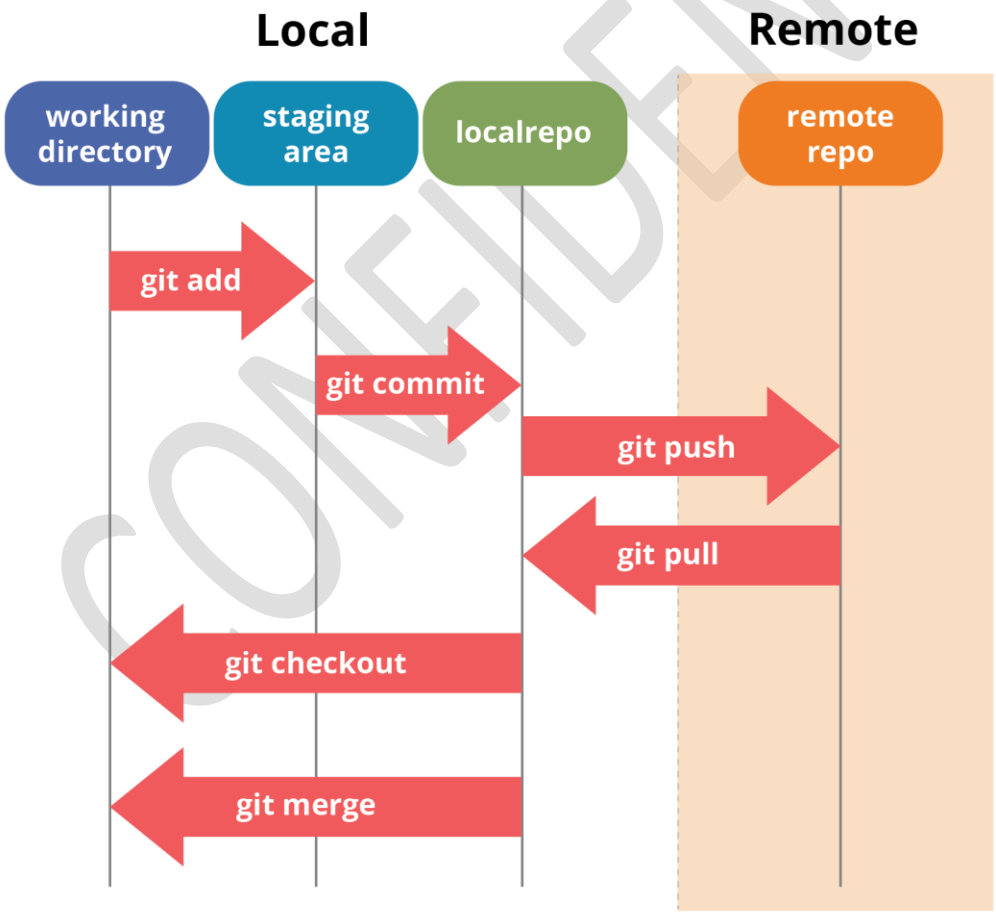
Pushing "Fork" button to *copy* a project from an account (e.g., project maintainer) to your own personal account. [TODO] diagram

Pushing "Pull Request" button to *notify* other developers (e.g., project maintainer or the entire project team) to review your changes. If accepted, the project maintainer can pull and apply the changes. A pull request shall provide the source's repo name, source's branch name, destination's repo name and destination's branch name.

GitHub - Contributing to a Project

https://git-scm.com/book/en/v2/GitHub-Contributing-to-a-Project

Look at the architecture of Git below:

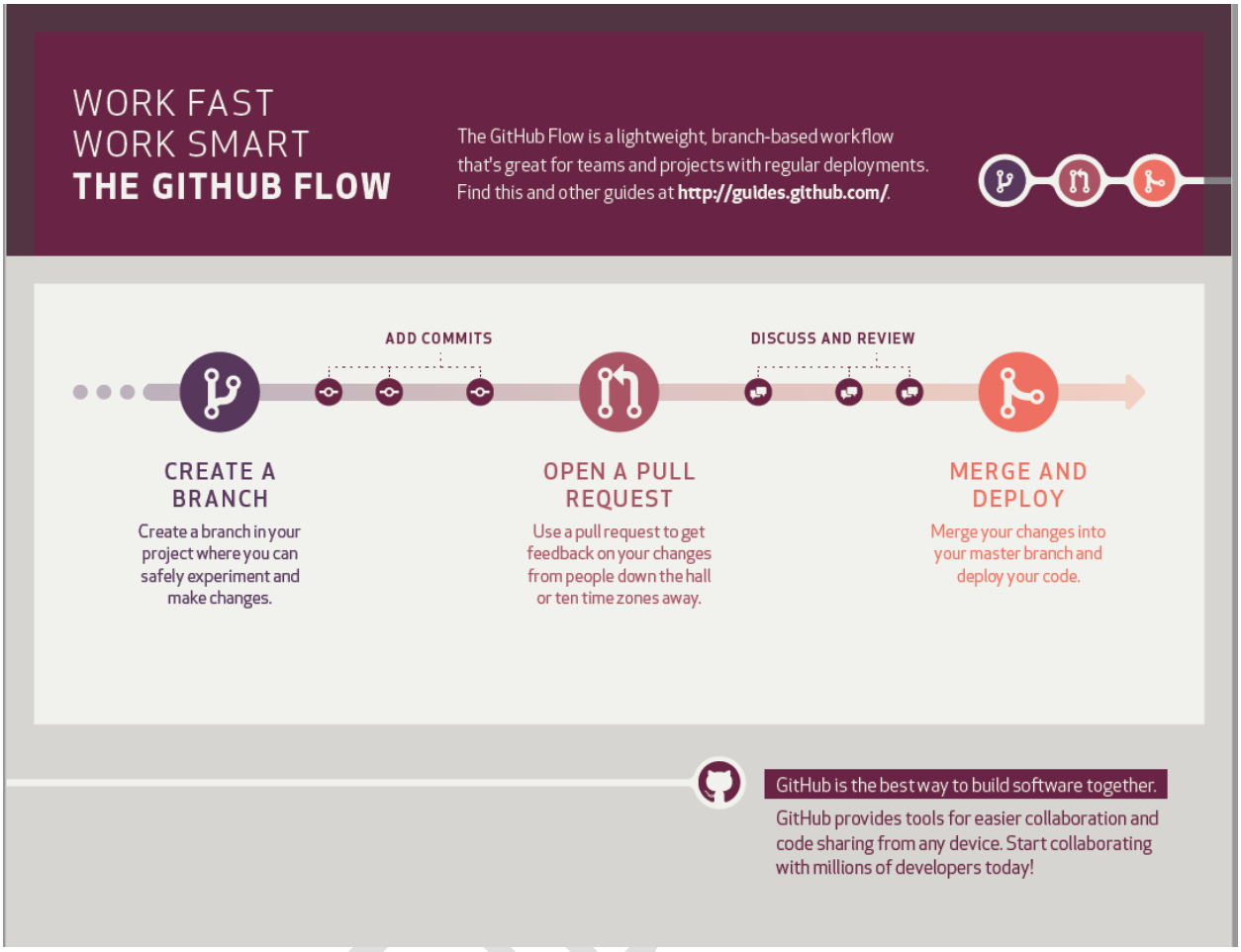


Difference between Git Vs GitHub

Git	GitHub
Git is a distributed version control tool that can manage a programmer's source code history.	GitHub is a cloud-based tool developed around the Git tool.
A developer installs Git tool locally.	GitHub is an online service to store code and push from the computer running the Git tool.
Git focused on version control and code sharing.	GitHub focused on centralized source code hosting.
It is a command-line tool.	It is administered through the web.
It facilitates with a desktop interface called Git Gui.	It also facilitates with a desktop interface called GitHub Gui.
Git does not provide any user management feature.	GitHub has a built-in user management feature.
It has minimal tool configuration feature.	It has a market place for tool configuration.

The GitHub flow

The GitHub flow is a lightweight, branch-based workflow built around core Git commands used by teams around the globe including ours.



The GitHub flow has six steps, each with distinct benefits when implemented:

Create a branch: Topic branches created from the canonical deployment branch (usually `main`) allow teams to contribute to many parallel efforts. Short-lived topic branches keep teams focused and results in quick ships.

Add commits: Snapshots of development efforts within a branch create safe, revertible points in the project's history.

Open a pull request: Pull requests publicize a project's ongoing efforts and set the tone for a transparent development process.

Discuss and review code: Teams participate in code reviews by commenting, testing, and reviewing open pull requests. Code review is at the core of an open and participatory culture.

Merge: Upon clicking merge, GitHub automatically performs the equivalent of a local 'git merge' operation. GitHub also keeps the entire branch development history on the merged pull request.

Deploy: Teams can choose the best release cycles or incorporate continuous integration tools and operate with the assurance that code on the deployment branch has gone through a robust workflow.

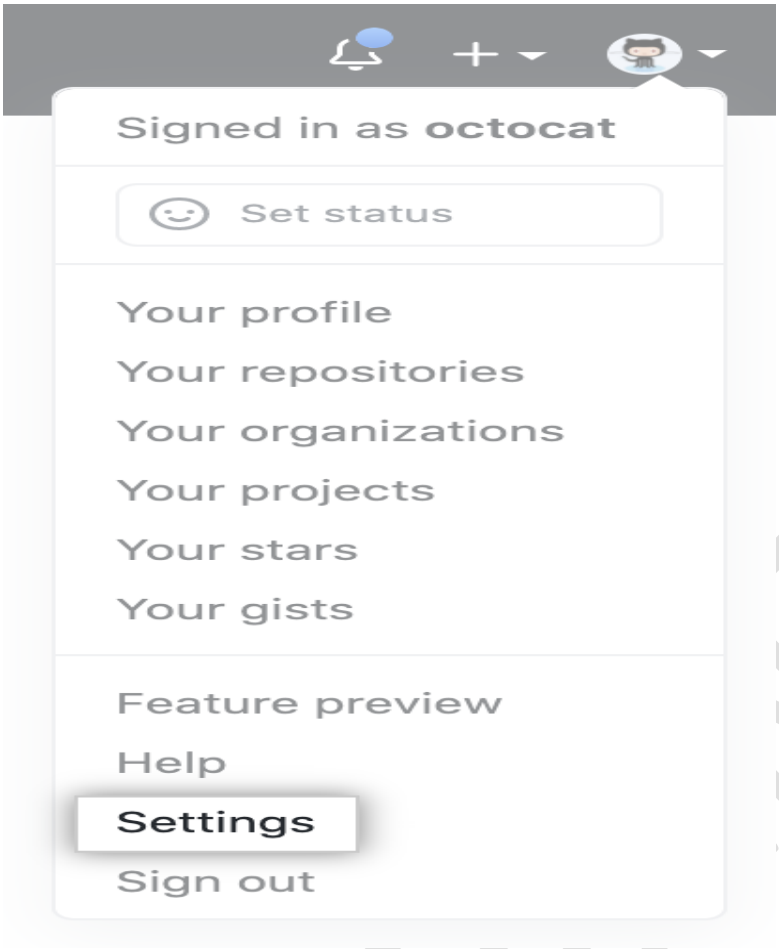
Developers can find more information about the GitHub flow in the GitHub Flow Video

https://youtu.be/47E-jcuQz5c?list=PLg7s6cbtAD17Gw5u8644bgKhgRLiJXdX4

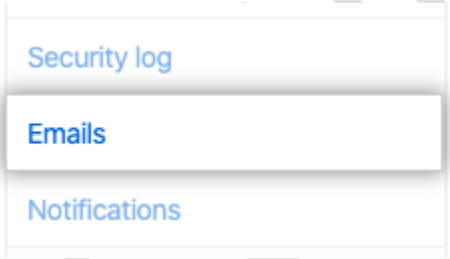
Setting your commit email address on GitHub

If you haven't enabled email address privacy, you can choose which verified email address to author changes with when you edit, delete, or create files or merge a pull request on GitHub. If you enabled email address privacy, then the commit author email address cannot be changed and is <username>@users.noreply.github.com by default.

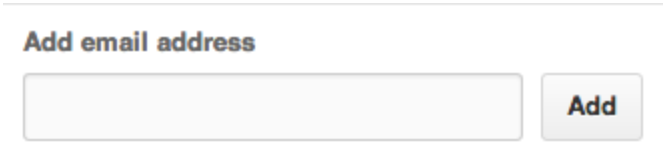
In the upper-right corner of any page, click your profile photo, then click **Settings**.



In the left sidebar, click **Emails**.

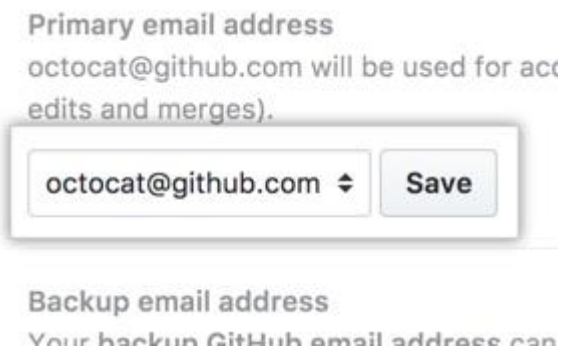


In "Add email address", type your email address and click **Add**.

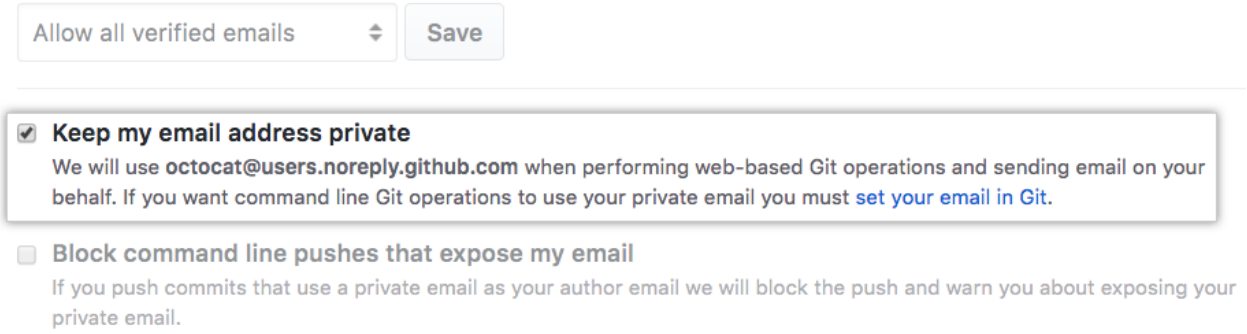


Verify your email address.

In the "Primary email address" list, select the email address you'd like to associate with your web-based Git operations.



To keep your email address private when performing web-based Git operations, click **Keep my email address private**.



GitHub and the command line

Example: Start a new repository and publish it to GitHub

First, you will need to create a new repository on GitHub.

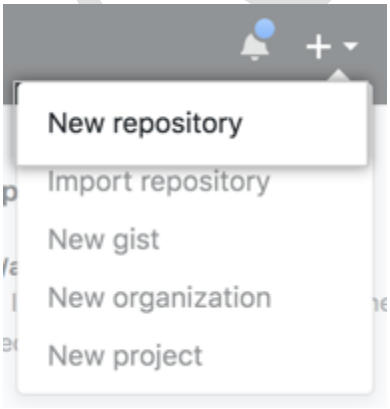
```
# create a new directory, and initialize it with git-specific functions

git init my-repo


# change into the `my-repo` directory

cd my-repo


# create the first file in the project
```

```
touch README.md

# git isn't aware of the file, stage it
git add README.md

# take a snapshot of the staging area
git commit -m "add README to initial commit"

# provide the path for the repository you created on github
git remote add origin https://github.com/YOUR-USERNAME/YOUR-REPOSITORY.git

# push changes to GitHub
git push --set-upstream origin main
```

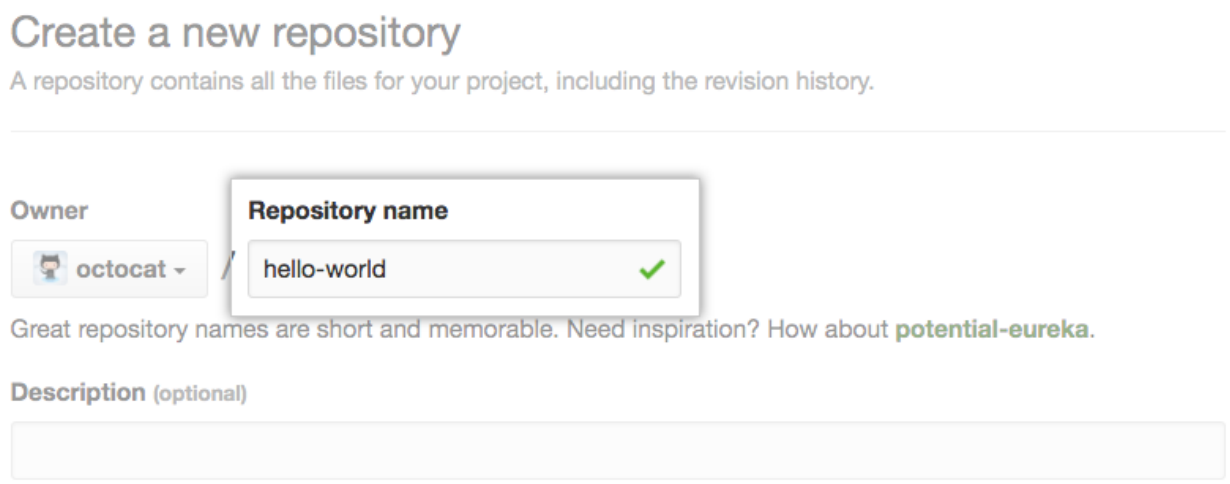
Create a repo

To put your project up on GitHub, you'll need to create a repository for it to live in.

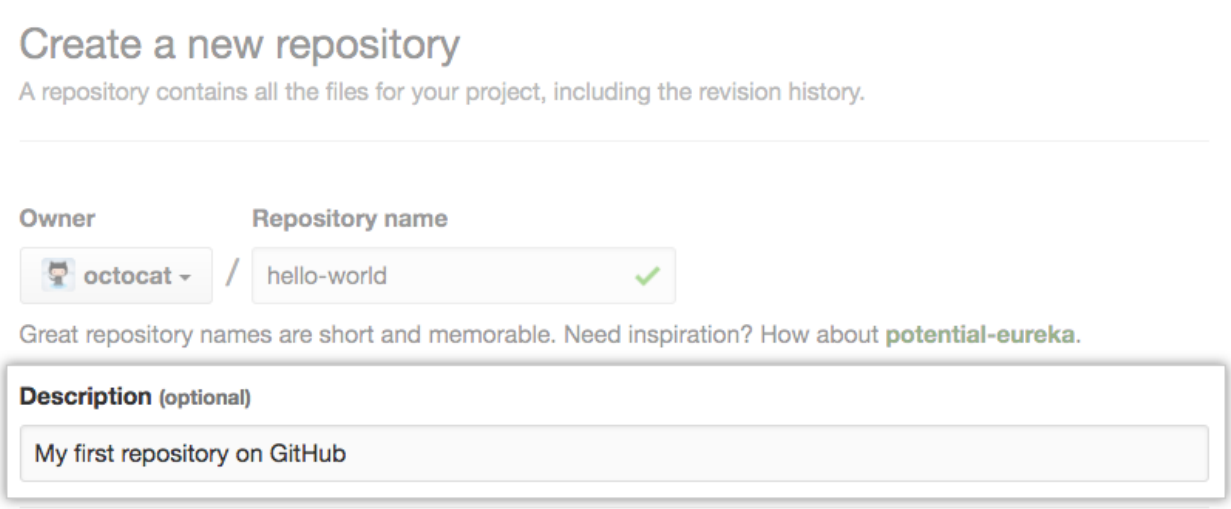
In the upper-right corner of any page, use the drop-down menu, and select New repository.



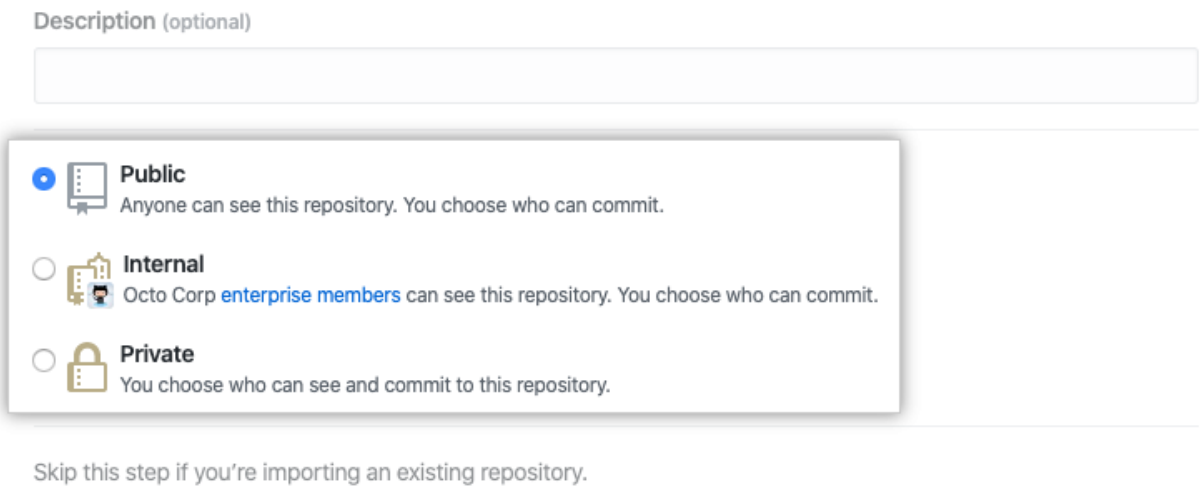
Type a short, memorable name for your repository. For example, "hello-world".



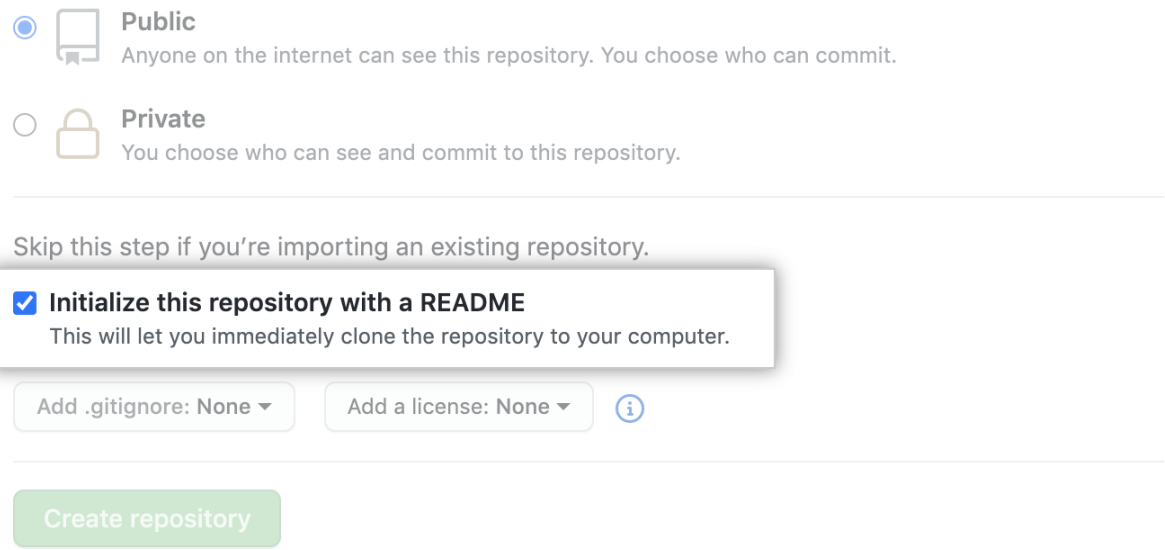
Optionally, add a description of your repository. For example, "My first repository on GitHub."



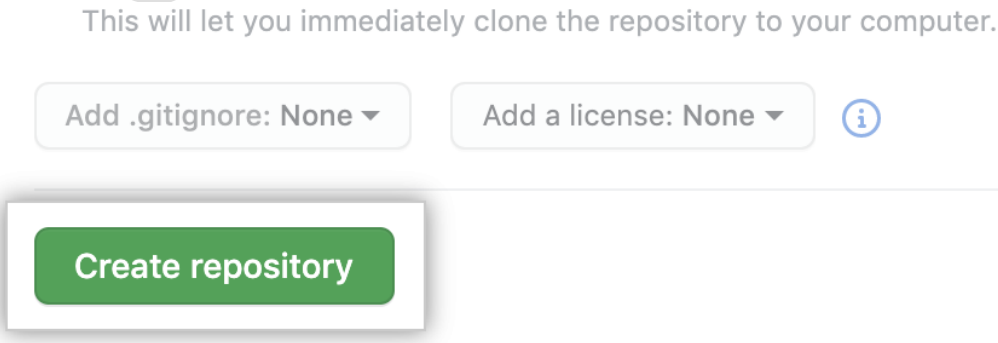
Choose a repository visibility. For more information, see "About repository visibility."



Select Initialize this repository with a README.



Click Create repository.



Congratulations! You've successfully created your first repository and initialized it with a *README* file.

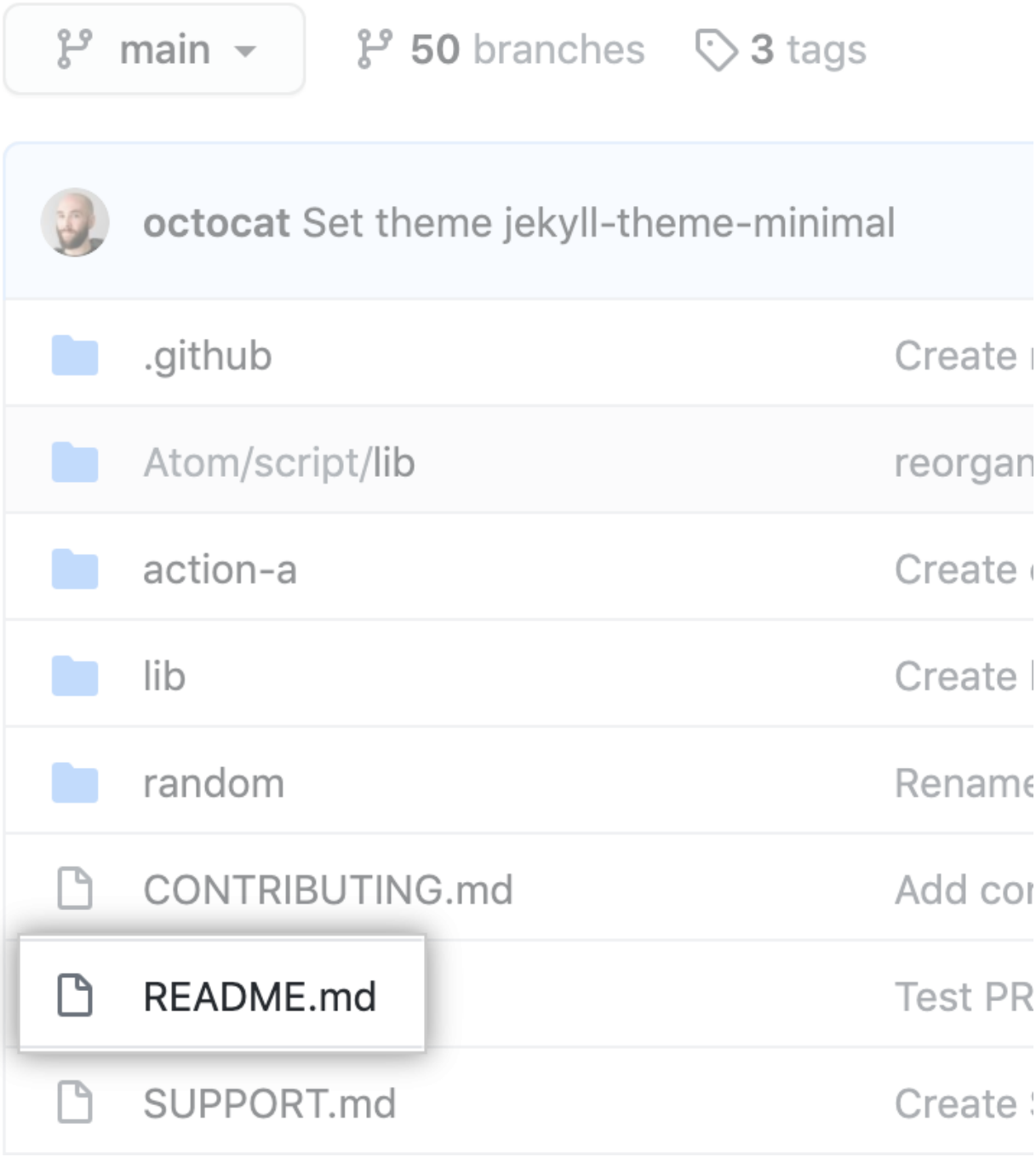
Commit your first change

A *commit* is like a snapshot of all the files in your project at a particular point in time.

When you created your new repository, you initialized it with a *README* file. *README* files are a great place to describe your project in more detail, or add some documentation such as how to install or use your project. The contents of your *README* file are automatically shown on the front page of your repository.

Let's commit a change to the *README* file.

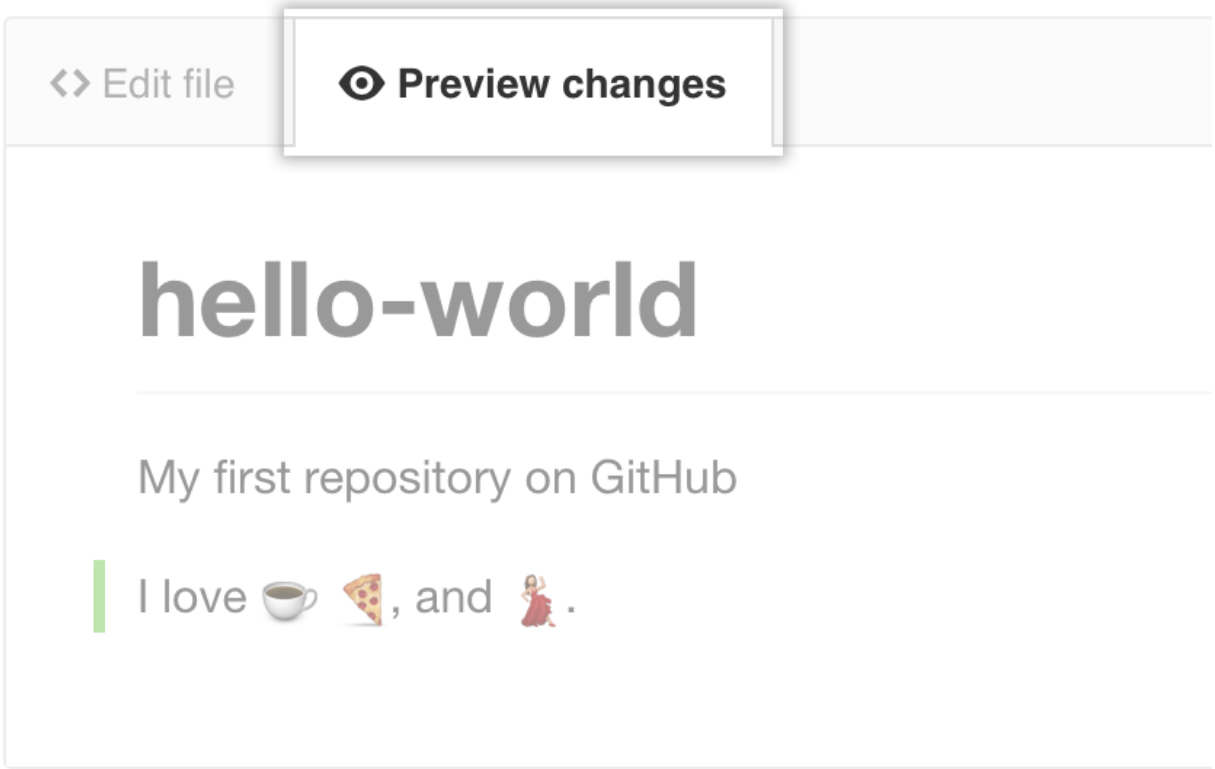
In your repository's list of files, click ***README.md***.



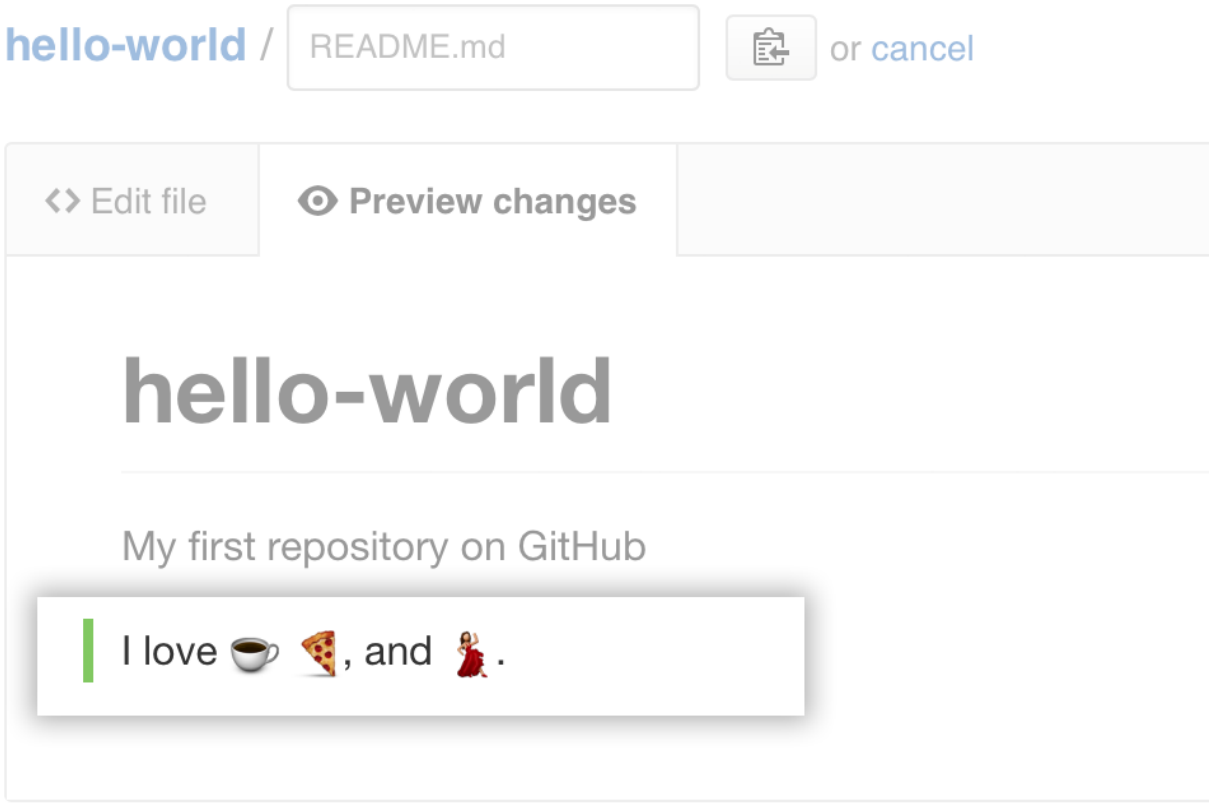
Above the file's content, click .

On the **Edit file** tab, type some information about yourself.

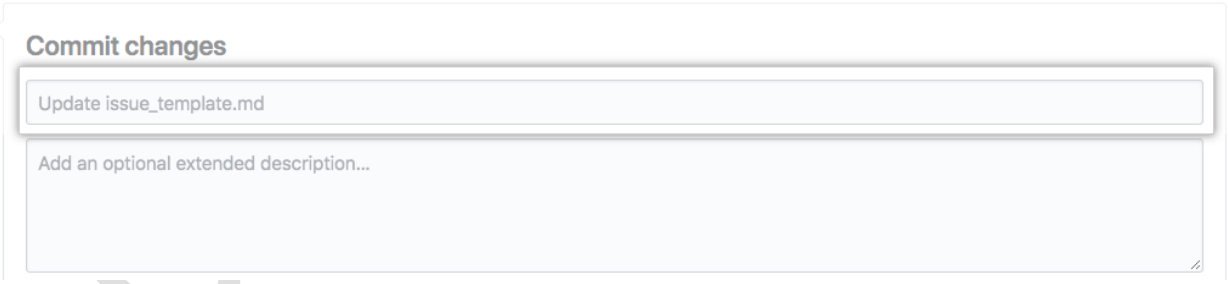
Above the new content, click **Preview changes**.



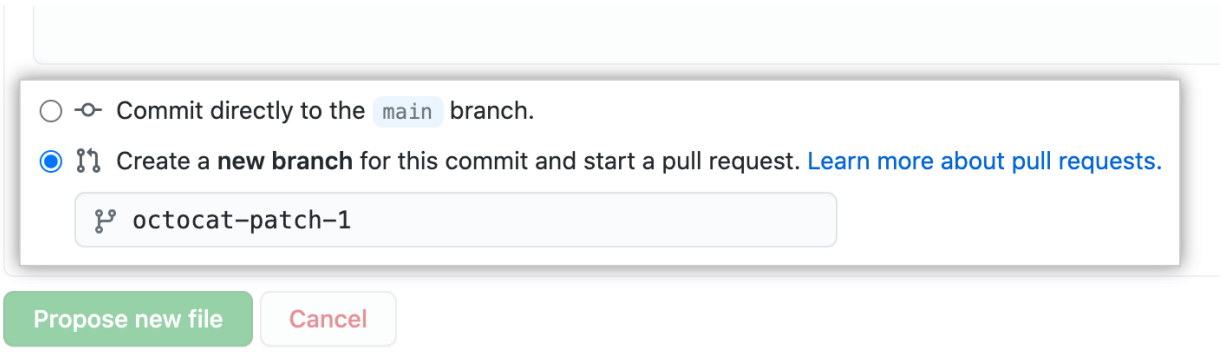
Review the changes you made to the file. You'll see the new content in green.



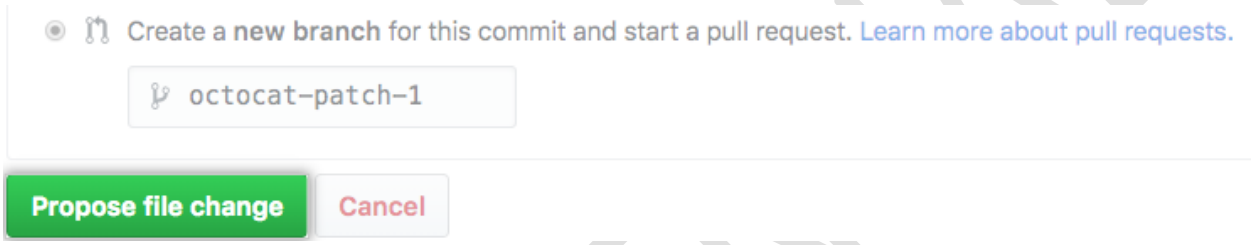
At the bottom of the page, type a short, meaningful commit message that describes the change you made to the file. You can attribute the commit to more than one author in the commit message. For more information, see "Creating a commit with multiple co-authors."



Below the commit message fields, decide whether to add your commit to the current branch or to a new branch. If your current branch is the default branch, you should choose to create a new branch for your commit and then create a pull request. For more information, see "Creating a new pull request."



Click **Propose file change**.

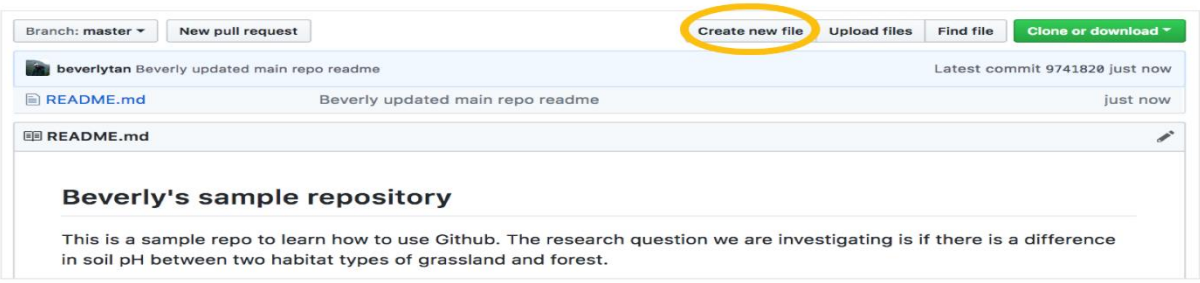


Developing folders within the repository

Earlier, we decided that a folder structure of Code, Data, Final Products and Intermediate Products will be used in this repository. We now have to create these folders in our main repository. Return to your main repository page by clicking “sample-repo” at the top of the page. Click “Create new file”, and a new editor page will come up.

In the box that says “Name your file”, first type the name of your folder, in this case “code”, and type a slash “/” to signify the end of the folder name, and then type “README.md” to create a README file in the code folder (similar to how we clicked “Initialize repository with a README file).

Remember that we can’t make empty folders, so we will need to have to create a README.md file within the folder. In the “Edit new file” window, type some information about the folder, in this case just tell yourself that it is the Code folder, and what it will contain.

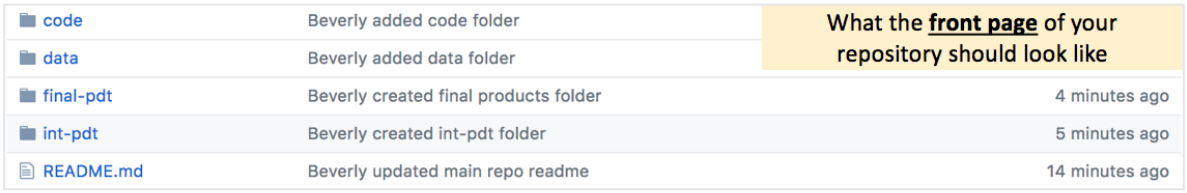


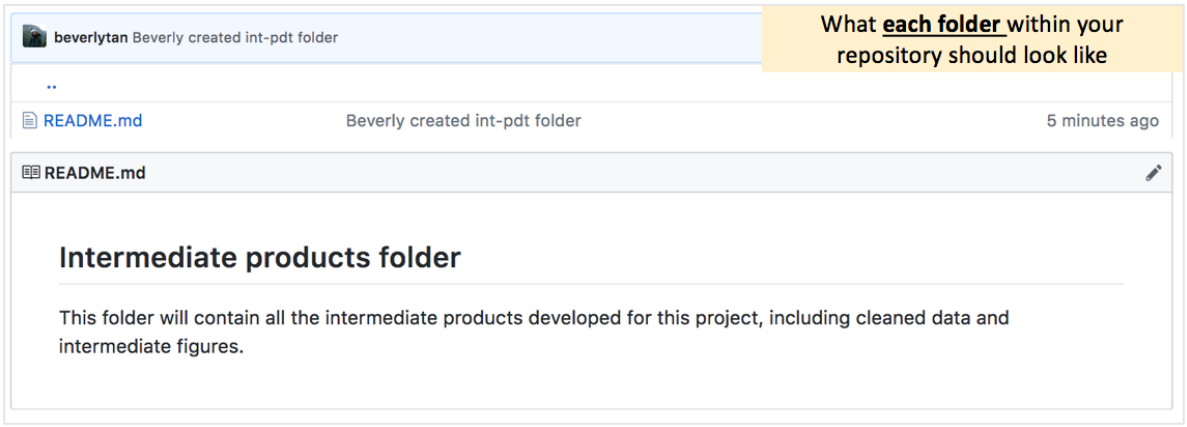
Tip: Using the # in a Markdown file will make the text a header, in this case I'll be making the words "Code folder" a header. The more #'s you add, the smaller the font size will be, to signify a subheader.

Commit these changes with an informative commit message again, such as "Beverly added code folder".

Now navigate back to the main repository page, and repeat the above process to create folders for Code, Data, Intermediate Products and Final Products. Note that if we didn't navigate back to the main repository page, and you click Create new file > "examplefolder/README.md" while you're in the Code folder, you would create that "examplefolder" folder within the Code folder. Remember that every time you create a new folder, you will need to have a new README for each subfolder - you're not able to create empty folders!

Your repository should now have the 4 folders and the main README.md file, and each folder should have a README.md within, telling you what the folder contains. Your repository and associated folders should look like this:





Perfect! You now have an organized repository with clear folder structure and you can now add material into your folders!

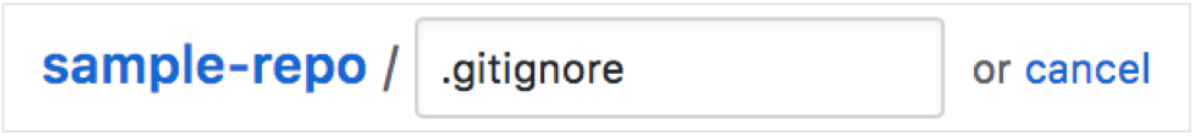
Adding a gitignore and repository etiquette file to our repository

Beyond the README files that have to be found in every folder, the two other files that we always have to add to the repository are a gitignore file and a repository etiquette file. You will just need to add these two files to the main repository, not in each folder.

gitignore file

A gitignore file is a configuration file that uses a series of rules to identify files that Git should not track. The files we want to exclude are generally hidden, temporary, machine generated files, such as: temporary Microsoft Doc / Excel / Powerpoint file and Mac users' hidden .DS_Store files. In collaborative work, if you don't create a gitignore file and specify these rules, someone who comes along and downloads your repository will download all these unnecessary files!

We add the gitignore file at the highest level of your repository. Making sure you've returned to the front page of your repository, click "Create new file" and then type .gitignore.



Now, copy the following gitignore template into the editor:

```
# User RProject files

.Rproj.user

*.Rproj


# User .RData and .Rhistory files

.RData

.Rhistory

.Rapp.history


# Temporary files

*~

~$*.doc*

~$*.xls*

*.xlk

~$*.ppt*


# Mac users' hidden .DS_Store files

*.DS_Store
```

Users README files created by RStudio

*README.html

*README_cache/

*README_files/

You might be wondering why we don't want Git to track our .Rproj file. We don't need to, since we already track the changes of the R script itself - we can always create the objects and run the code in our R Project file again.

Commit your change with a message e.g. "Beverly added gitignore file", and return to the main repository page.

Repository etiquette file

We now have to add in a repository etiquette file. This is especially important if you are adding collaborators to your repository. In projects that allow other people to edit the content of your repo, you want to tell them how you'd prefer for things to be done, and rules to follow.

To add a repository etiquette file in the main repository, click "Create new file", and then type repo-etiquette.md. This would just be a Markdown file, and you can name it how you'd like, but "repo-etiquette" would work just fine.

Copy and paste the following example repository etiquette into the editor.

Repository etiquette

- File paths to be kept short and sensible, use Github relative file paths instead of local computer file paths.

- Do not use capital letters, funky characters, symbols or spaces in file names

- Always pull before you push to avoid coding conflicts

- Commit messages

  - To be kept short and informative

  - Each message should revolve around one change or fix

Again, commit your change with “Beverly added repo etiquette file”.

Fork a repo

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

Fork an example repository

Keep your fork synced

Find another repository to fork

Celebrate

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.

Propose changes to someone else's project

For example, you can use forks to propose changes related to fixing a bug. 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.

Use someone else's project as a starting point for your own idea.

Open source software is based on the idea that by sharing code, we can make better, more reliable software. For more information, see the "About the Open Source Initiative" on the Open Source Initiative.

For more information about applying open source principles to your organization's development work on GitHub, see GitHub's white paper "An introduction to innersource."

When creating your public repository from a fork of someone's project, make sure to include a license file that determines how you want your project to be shared with others. For more information, see "Choose an open source license" at choosealicense.com.

For more information on open source, specifically how to create and grow an open source project, we've created Open Source Guides that will help you foster a healthy open source community by recommending best practices for creating and maintaining repositories for your

open source project. You can also take a free GitHub Learning Lab course on maintaining open source communities.

Note: You can use GitHub Desktop to fork a repository. For more information, see "Cloning and forking repositories from GitHub Desktop."

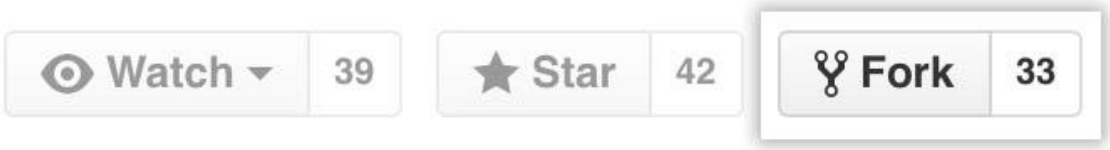
Tip: You can also fork a repository using the GitHub CLI. For more information, see "gh repo fork" in the GitHub CLI documentation.

Fork an example repository

Forking a repository is a simple two-step process. We've created a repository for you to practice with.

On GitHub, navigate to the octocat/Spoon-Knife repository.

In the top-right corner of the page, click **Fork**.



Keep your fork synced

You might fork a project to propose changes to the upstream, or original, repository. In this case, it's good practice to regularly sync your fork with the upstream repository. To do this, you'll need to use Git on the command line. You can practice setting the upstream repository using the same octocat/Spoon-Knife repository you just forked.

Step 1: Set up Git

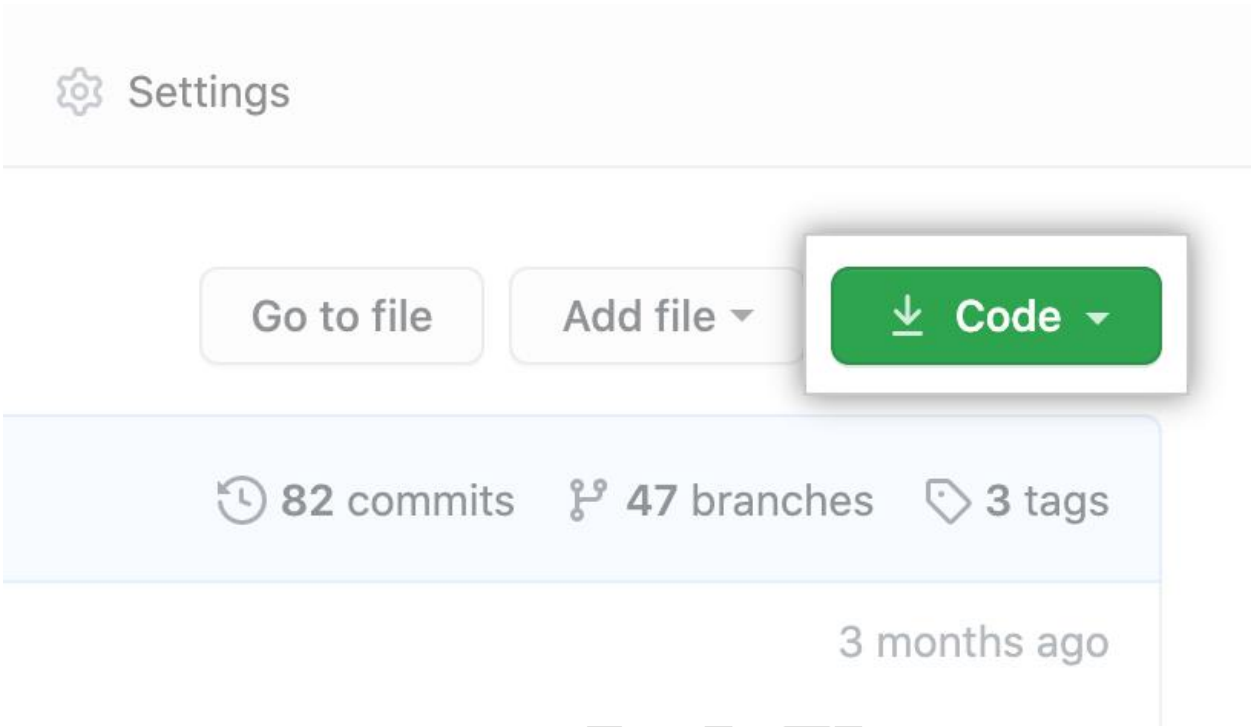
If you haven't yet, you should first set up Git. Don't forget to set up authentication to GitHub from Git as well.

Step 2: Create a local clone of your fork

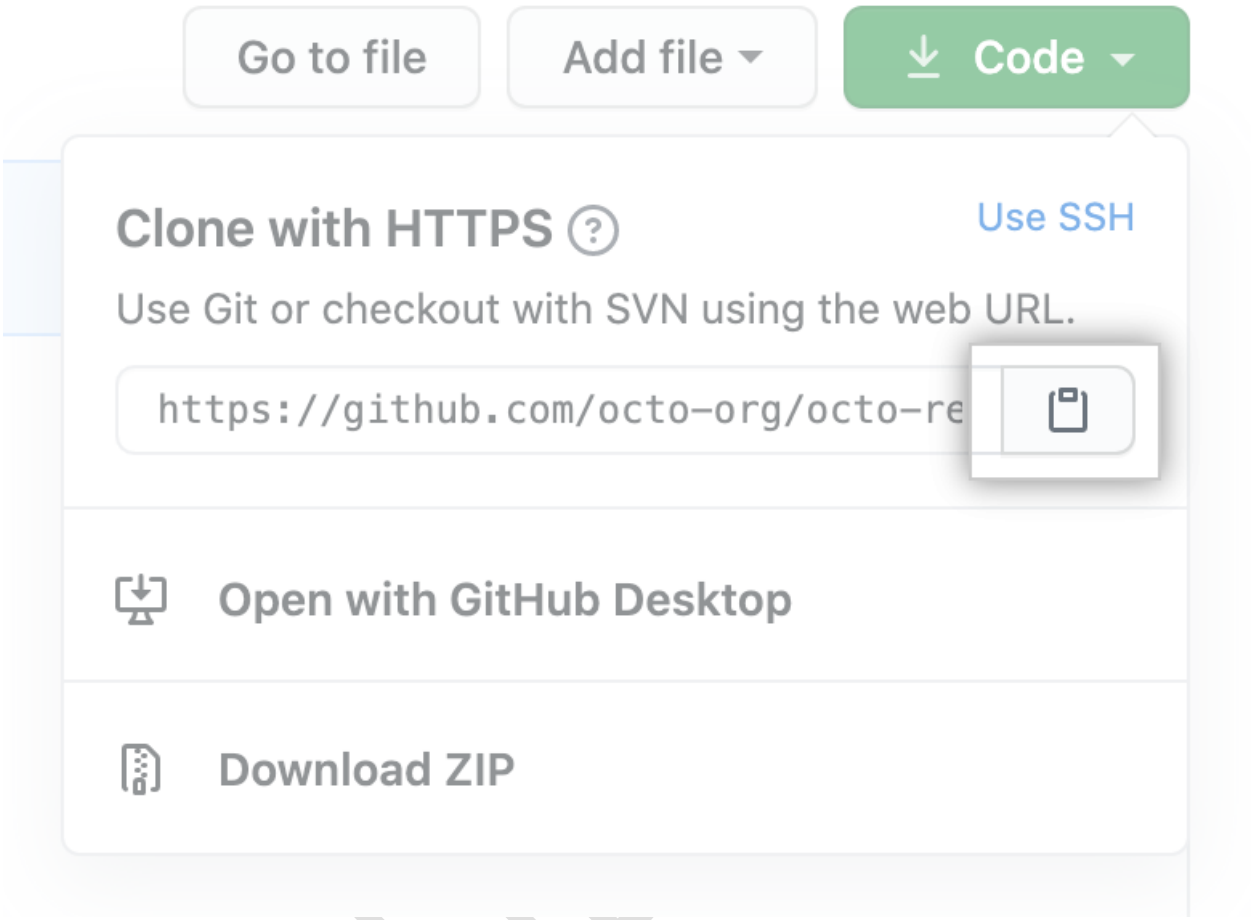
Right now, you have a fork of the Spoon-Knife repository, but you don't have the files in that repository on your computer. Let's create a clone of your fork locally on your computer.

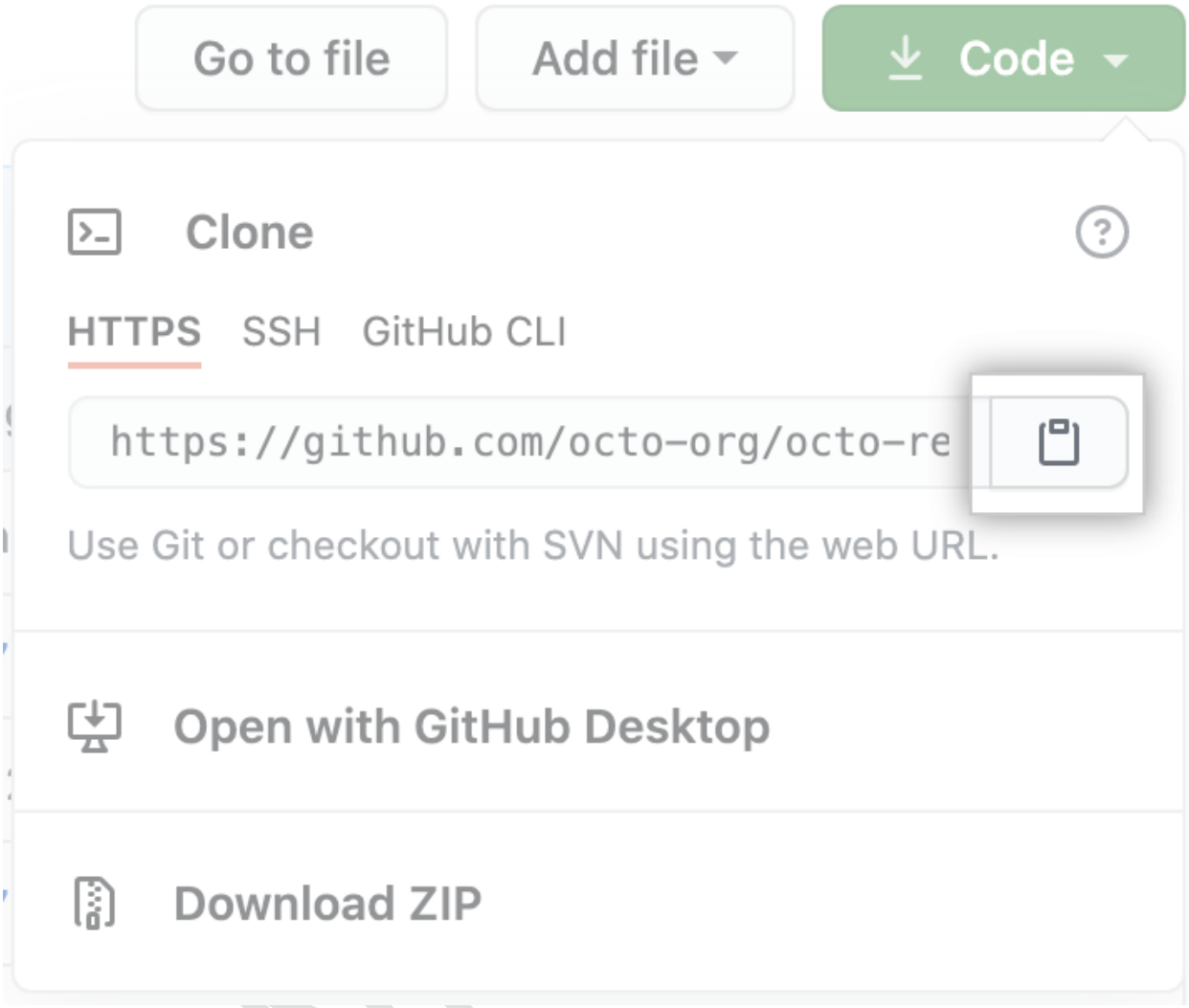
On GitHub, navigate to **your fork** of the Spoon-Knife repository.

Above the list of files, click **Code**.



To clone the repository using HTTPS, under "Clone with HTTPS", click . To clone the repository using an SSH key, including a certificate issued by your organization's SSH certificate authority, click **Use SSH**, then click . To clone a repository using GitHub CLI, click **Use GitHub CLI**, then click .





Open Git Bash.

Change the current working directory to the location where you want the cloned directory.

Type git clone, and then paste the URL you copied earlier. It will look like this, with your GitHub username instead of YOUR-USERNAME:

$ git clone https://github.com/*YOUR-USERNAME*/Spoon-Knife

Press **Enter**. Your local clone will be created.

$ git clone https://github.com/*YOUR-USERNAME*/Spoon-Knife

> Cloning into `Spoon-Knife`...

> remote: Counting objects: 10, done.

> remote: Compressing objects: 100% (8/8), done.

> remove: Total 10 (delta 1), reused 10 (delta 1)

> Unpacking objects: 100% (10/10), done.

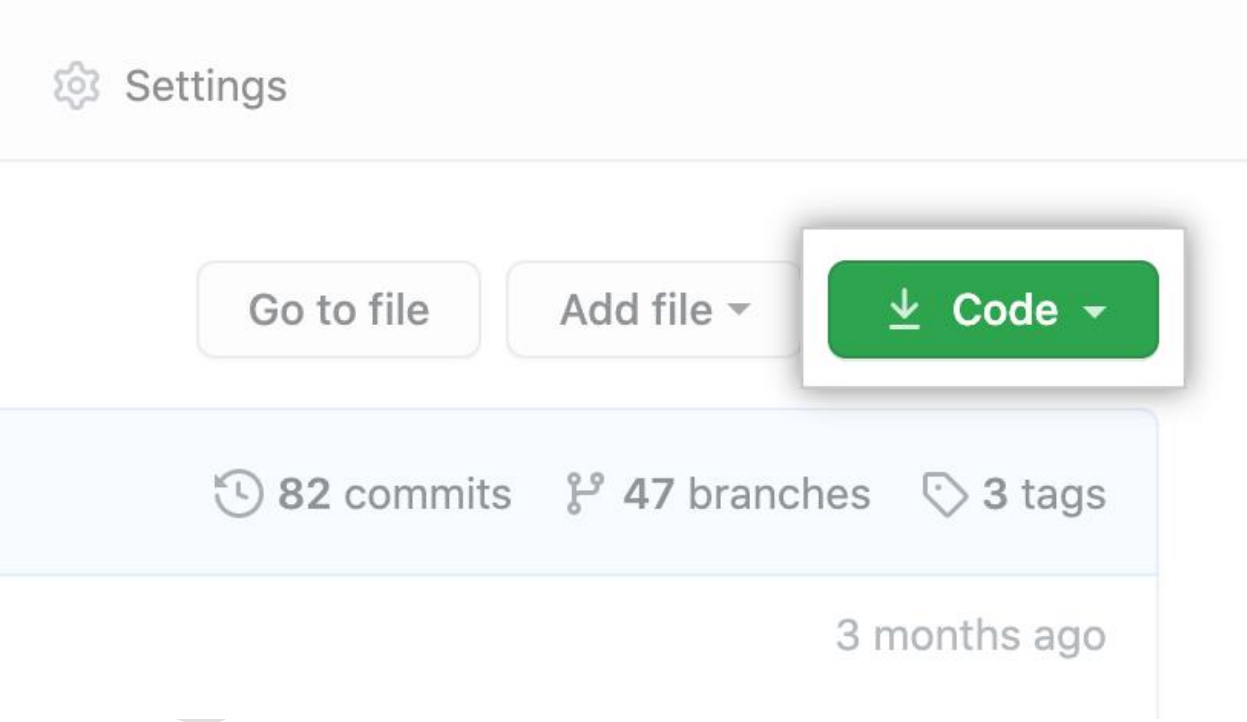
Now, you have a local copy of your fork of the Spoon-Knife repository.

Step 3: Configure Git to sync your fork with the original Spoon-Knife repository

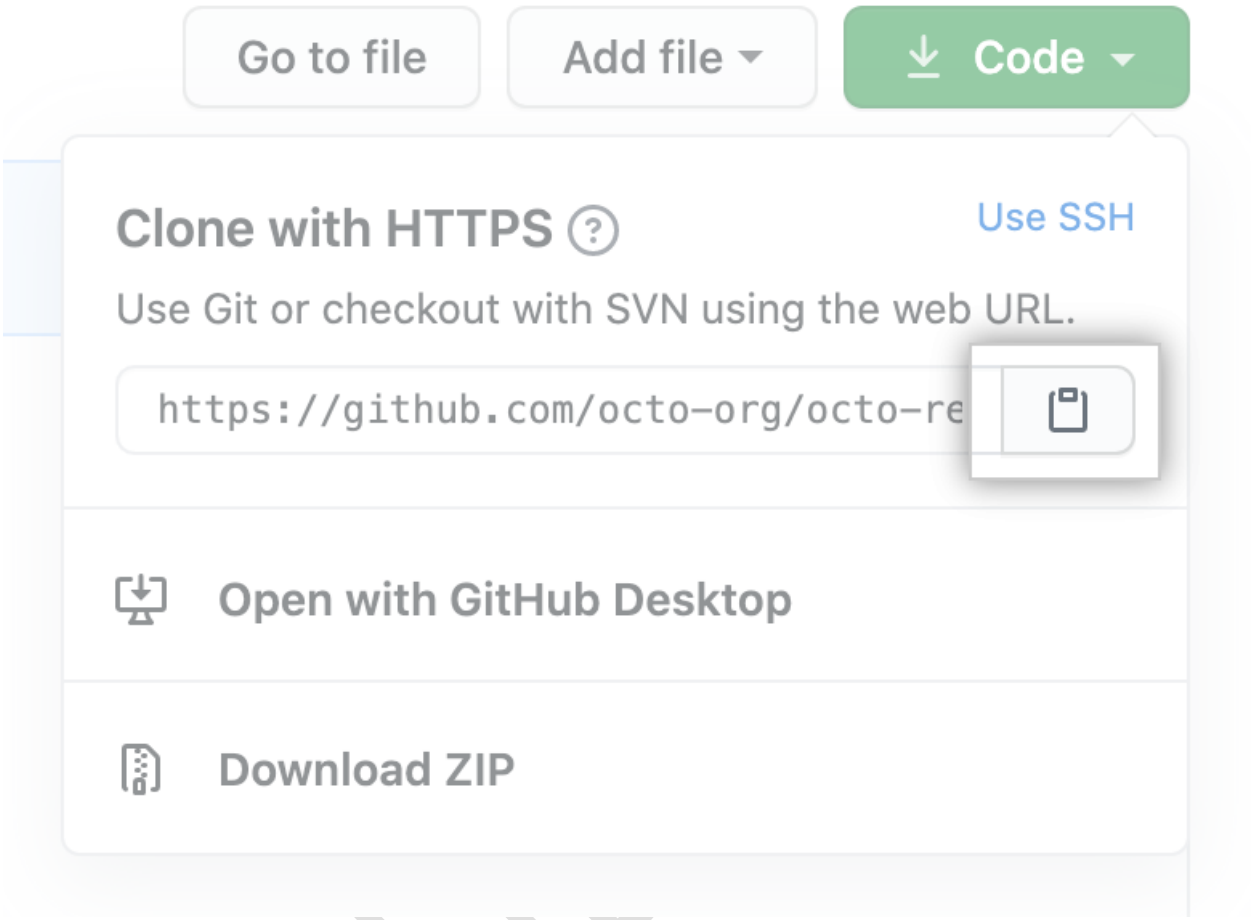
When you fork a project in order to propose changes to the original repository, you can configure Git to pull changes from the original, or upstream, repository into the local clone of your fork.

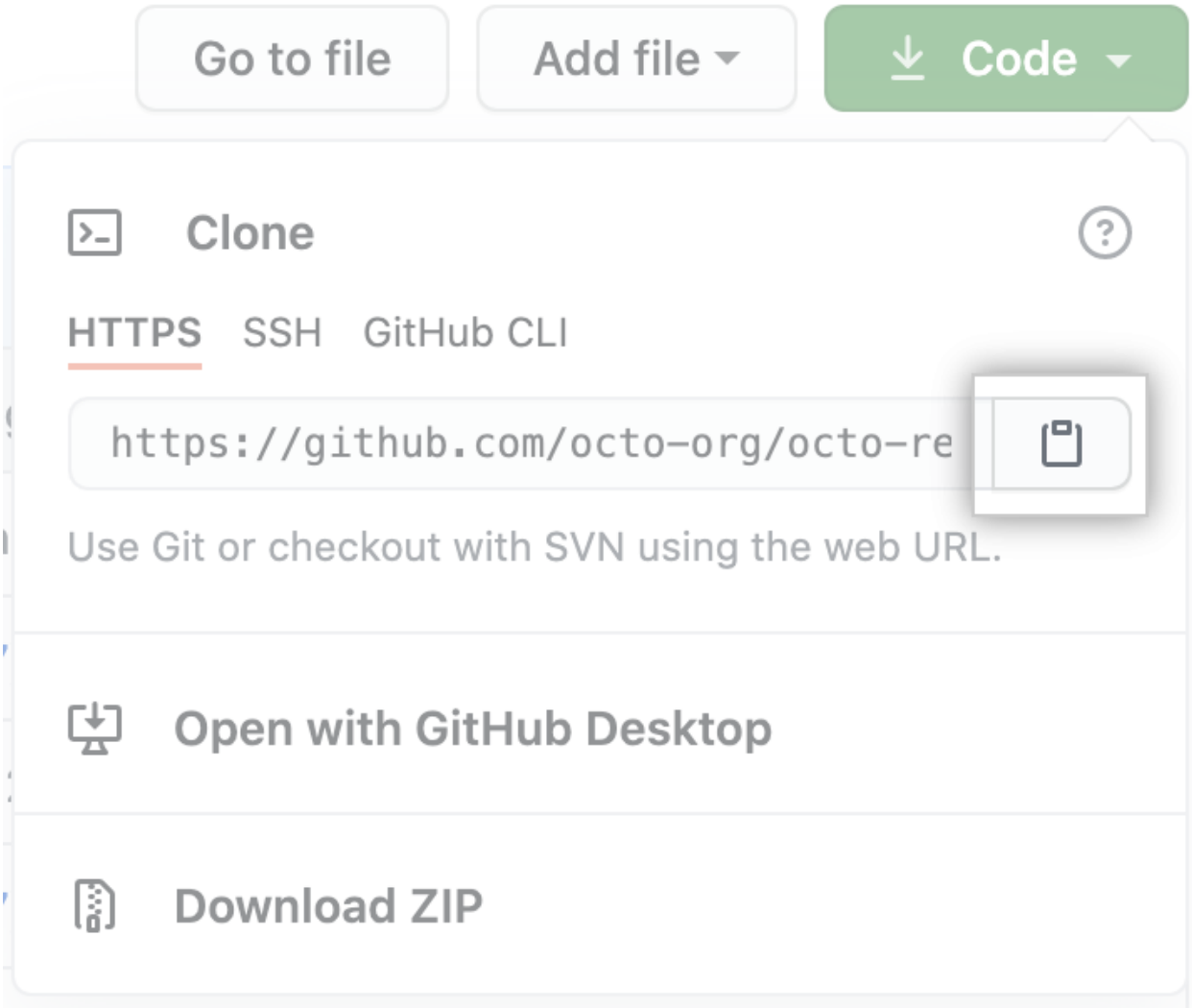
On GitHub, navigate to the octocat/Spoon-Knife repository.

Above the list of files, click **Code**.



To clone the repository using HTTPS, under "Clone with HTTPS", click . To clone the repository using an SSH key, including a certificate issued by your organization's SSH certificate authority, click **Use SSH**, then click . To clone a repository using GitHub CLI, click **Use GitHub CLI**, then click .





Open Git Bash.

Change directories to the location of the fork you cloned in Step 2: Create a local clone of your fork.

To go to your home directory, type just cd with no other text.

To list the files and folders in your current directory, type ls.

To go into one of your listed directories, type cd your_listed_directory.

To go up one directory, type cd ..

Type git remote -v and press **Enter**. You'll see the current configured remote repository for your fork.

```
$ git remote -v
> origin  https://github.com/YOUR_USERNAME/YOUR_FORK.git (fetch)
> origin  https://github.com/YOUR_USERNAME/YOUR_FORK.git (push)
```

Type `git remote add upstream`, and then paste the URL you copied in Step 2 and press **Enter**. It will look like this:

```
$ git remote add upstream https://github.com/octocat/Spoon-Knife.git
```

To verify the new upstream repository you've specified for your fork, type `git remote -v` again. You should see the URL for your fork as `origin`, and the URL for the original repository as `upstream`.

```
$ git remote -v
> origin    https://github.com/YOUR_USERNAME/YOUR_FORK.git (fetch)
> origin    https://github.com/YOUR_USERNAME/YOUR_FORK.git (push)
> upstream
https://github.com/ORIGINAL_OWNER/ORIGINAL_REPOSITORY.git (fetch)
> upstream
https://github.com/ORIGINAL_OWNER/ORIGINAL_REPOSITORY.git (push)
```

Now, you can keep your fork synced with the upstream repository with a few Git commands. For more information, see "Syncing a fork."

Next steps

You can make any changes to a fork, including:

Creating branches: *Branches* allow you to build new features or test out ideas without putting your main project at risk.

Opening pull requests: If you are hoping to contribute back to the original repository, you can send a request to the original author to pull your fork into their repository by submitting a pull request.

Find another repository to fork

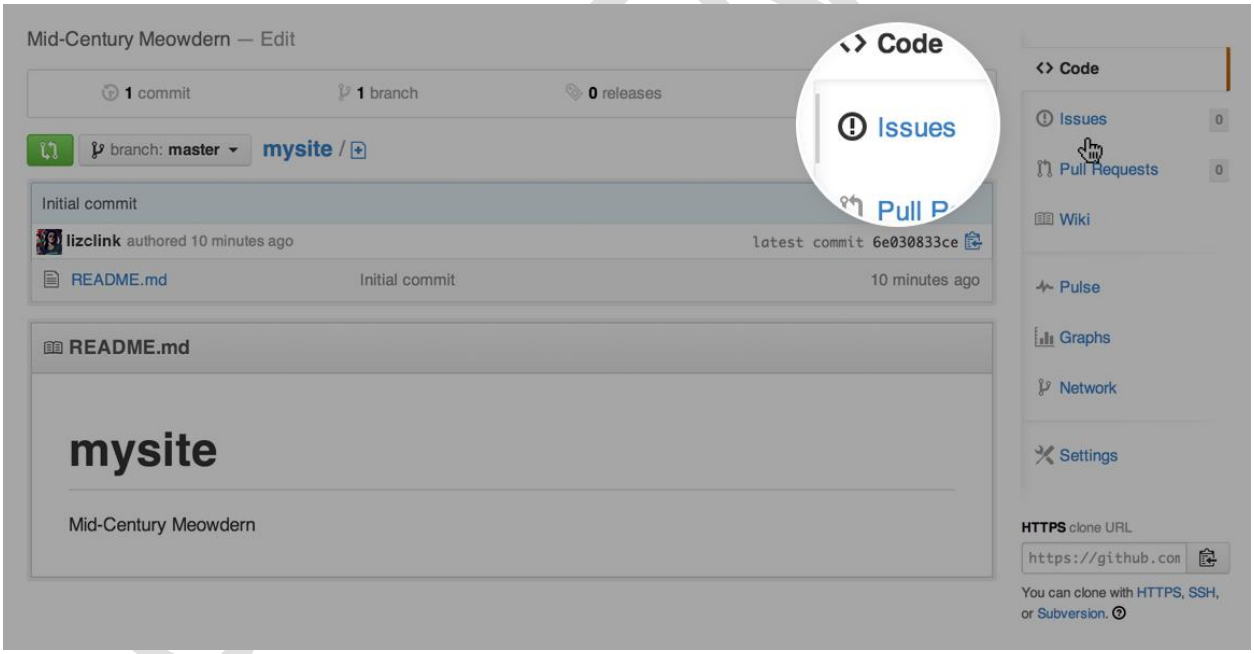
Fork a repository to start contributing to a project. You can fork a repository to your user account or any organization where you have repository creation permissions. For more information, see "Permission levels for an organization."

If you have access to a private repository and the owner permits forking, you can fork the repository to your user account or any organization on GitHub Team where you have repository creation permissions. You cannot fork a private repository to an organization using GitHub Free. For more information, see "GitHub's products."

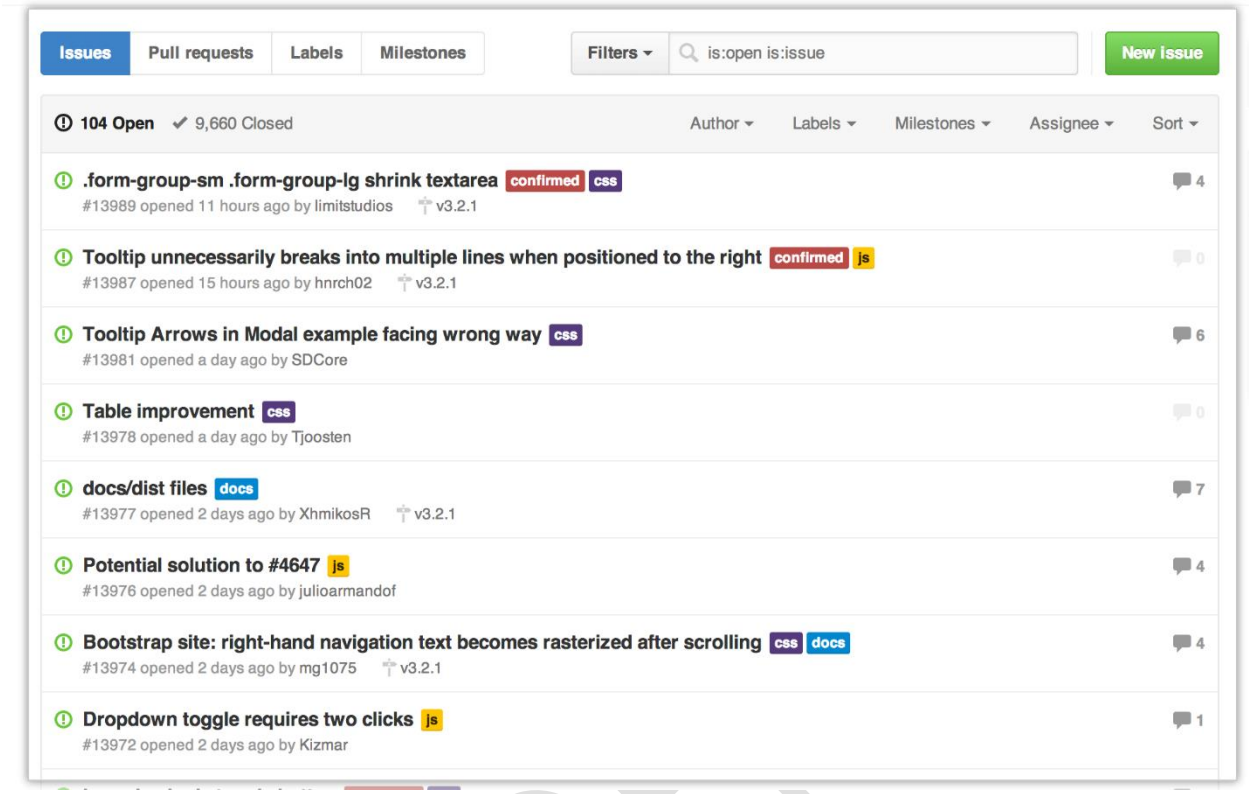
You can browse Explore to find projects and start contributing to open source repositories. For more information, see "Finding ways to contribute to open source on GitHub."

Mastering Issues

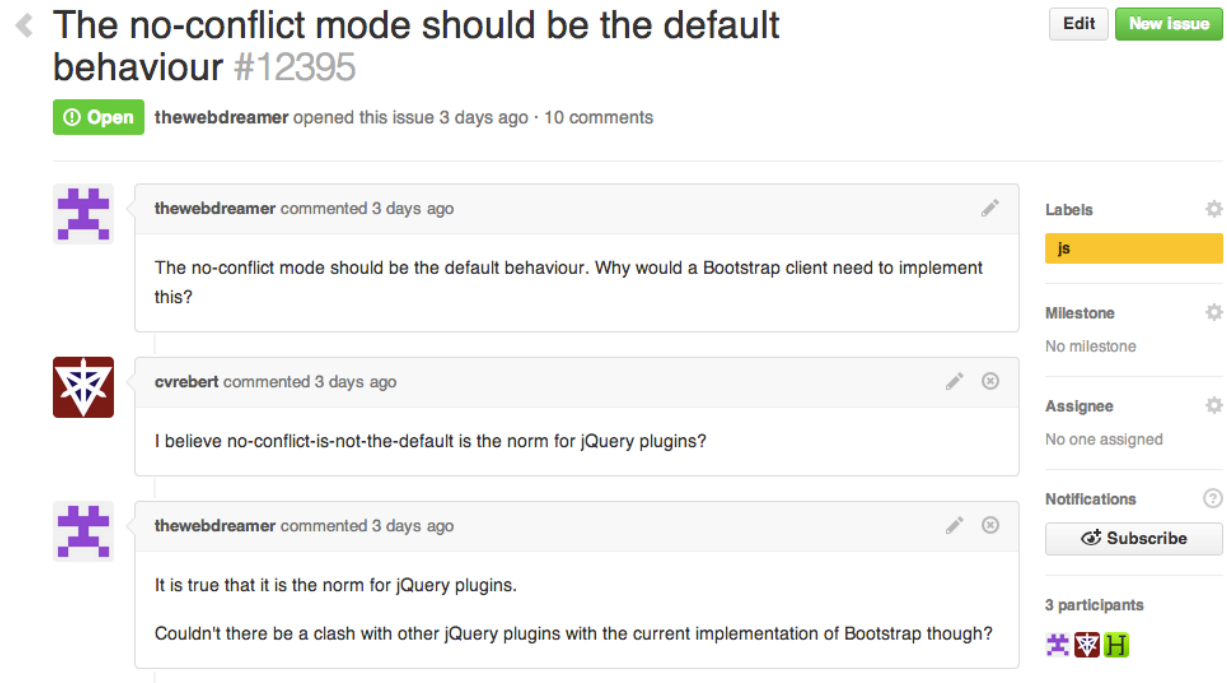
Issues are a great way to keep track of tasks, enhancements, and bugs for your projects. They're kind of like email—except they can be shared and discussed with the rest of your team. Most software projects have a bug tracker of some kind. GitHub's tracker is called **Issues** and has its own section in every repository.



For example, let's take a look at Bootstrap's Issues section:



GitHub's issue tracking is special because of our focus on collaboration, references, and excellent text formatting. A typical issue on GitHub looks a bit like this:



A **title** and **description** describe what the issue is all about.

Color-coded **labels** help you categorize and filter your issues (just like labels in email).

A **milestone** acts like a container for issues. This is useful for associating issues with specific features or project phases (e.g. *Weekly Sprint 9/5-9/16* or *Shipping 1.0*).

One **assignee** is responsible for working on the issue at any given time.

Comments allow anyone with access to the repository to provide feedback.

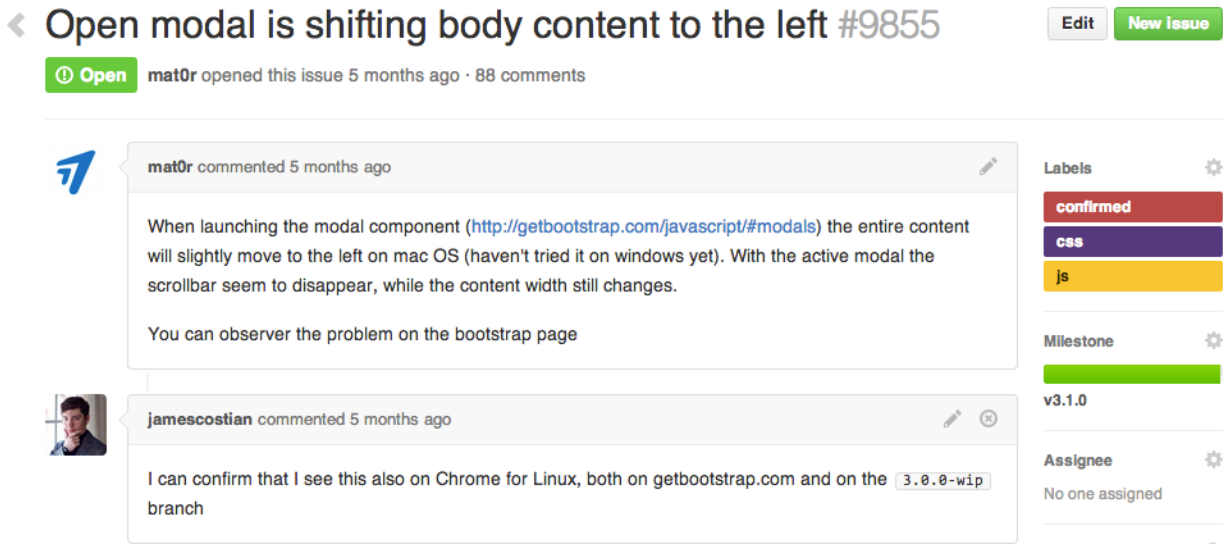
This is generally used as a way to bring up problems to other collaborators in projects that have more than one collaborator.

However, they are still very useful even if you are not using Github collaboratively, and just for your own independent work. They give you a platform to record problems that you might be having, and document how you're solving the problem - which might be helpful if you ever run into the problem again and want to check how you worked through it.

Milestones, Labels, and Assignees

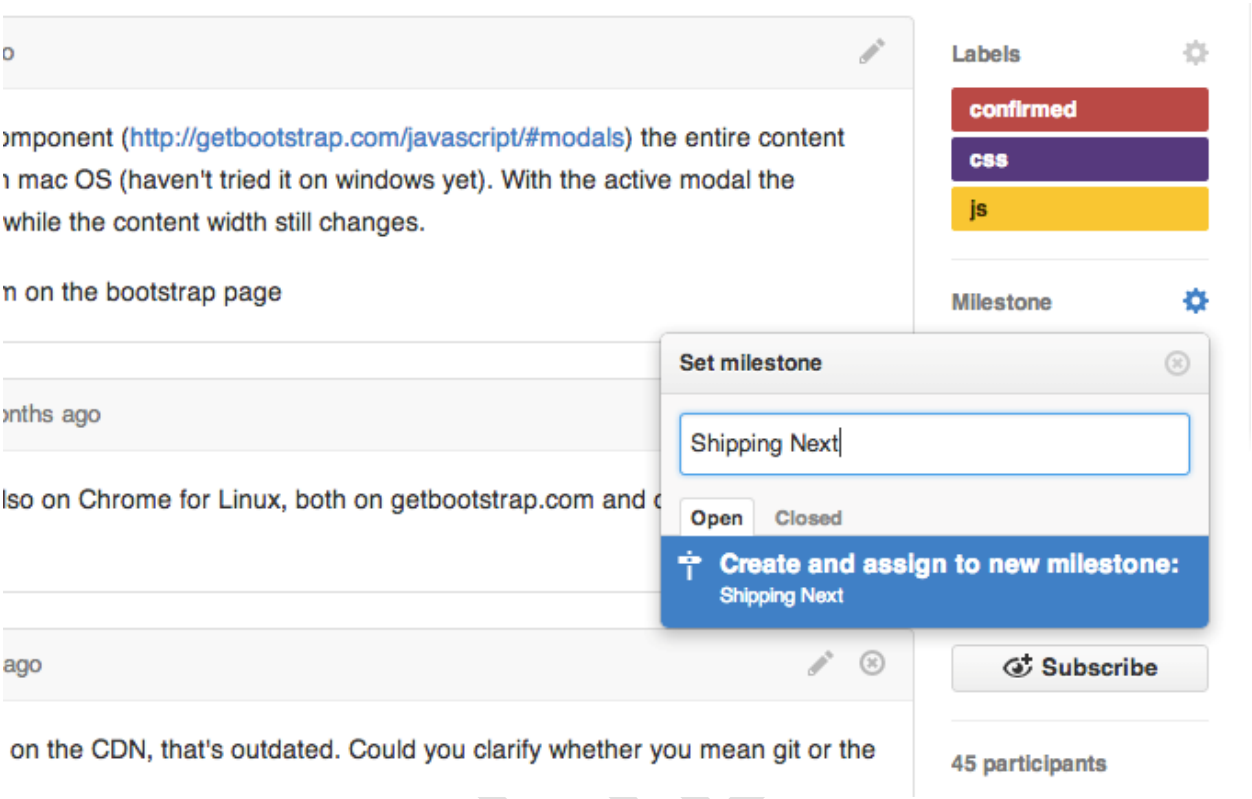
Once you've collected a lot of issues, you may find it hard to find the ones you care about. **Milestones**, **labels**, and **assignees** are great features to filter and categorize issues.

You can change or add a milestone, an assignee, and labels by clicking their corresponding gears in the sidebar on the right.



If you don't see edit buttons, that's because you don't have permission to edit the issue. You can ask the repository owner to add you as a collaborator to get access.

Milestones



Milestones are groups of issues that correspond to a project, feature, or time period. People use them in many different ways in software development. Some examples of milestones on GitHub include:

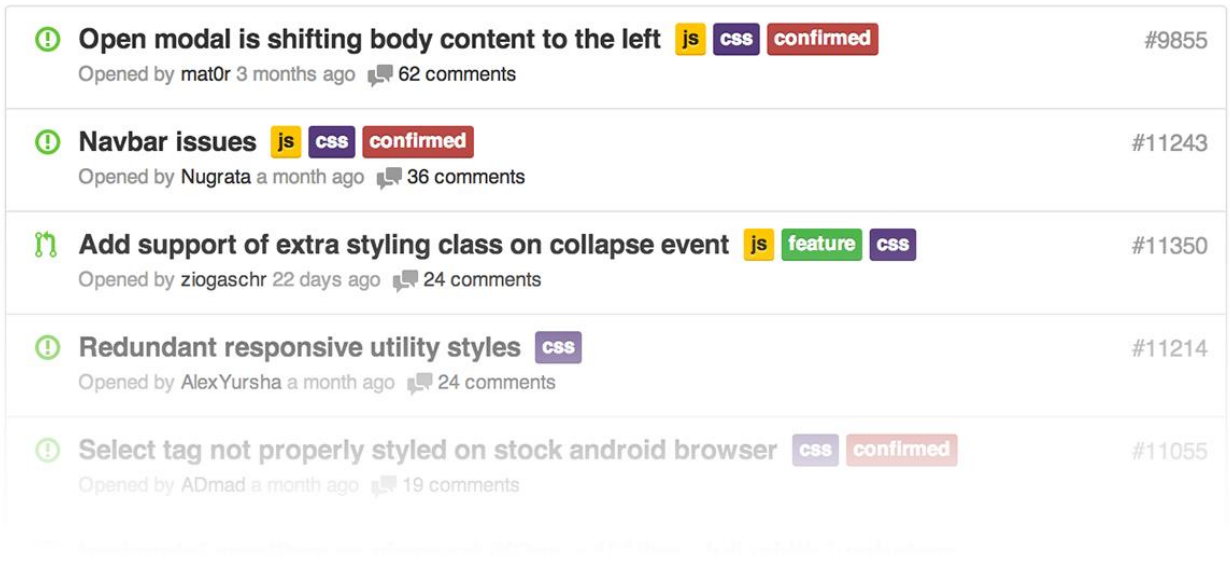
Beta Launch — File bugs that you need to fix before you can launch the beta of your project. It's a great way to make sure you aren't missing anything.

October Sprint — File issues that you'd like to work on in October. A great way to focus your efforts when there's a lot to do.

Redesign — File issues related to redesigning your project. A great way to collect ideas on what to work on.

Labels

Labels are a great way to organize different types of issues. Issues can have as many labels as you want, and you can filter by one or many labels at once.

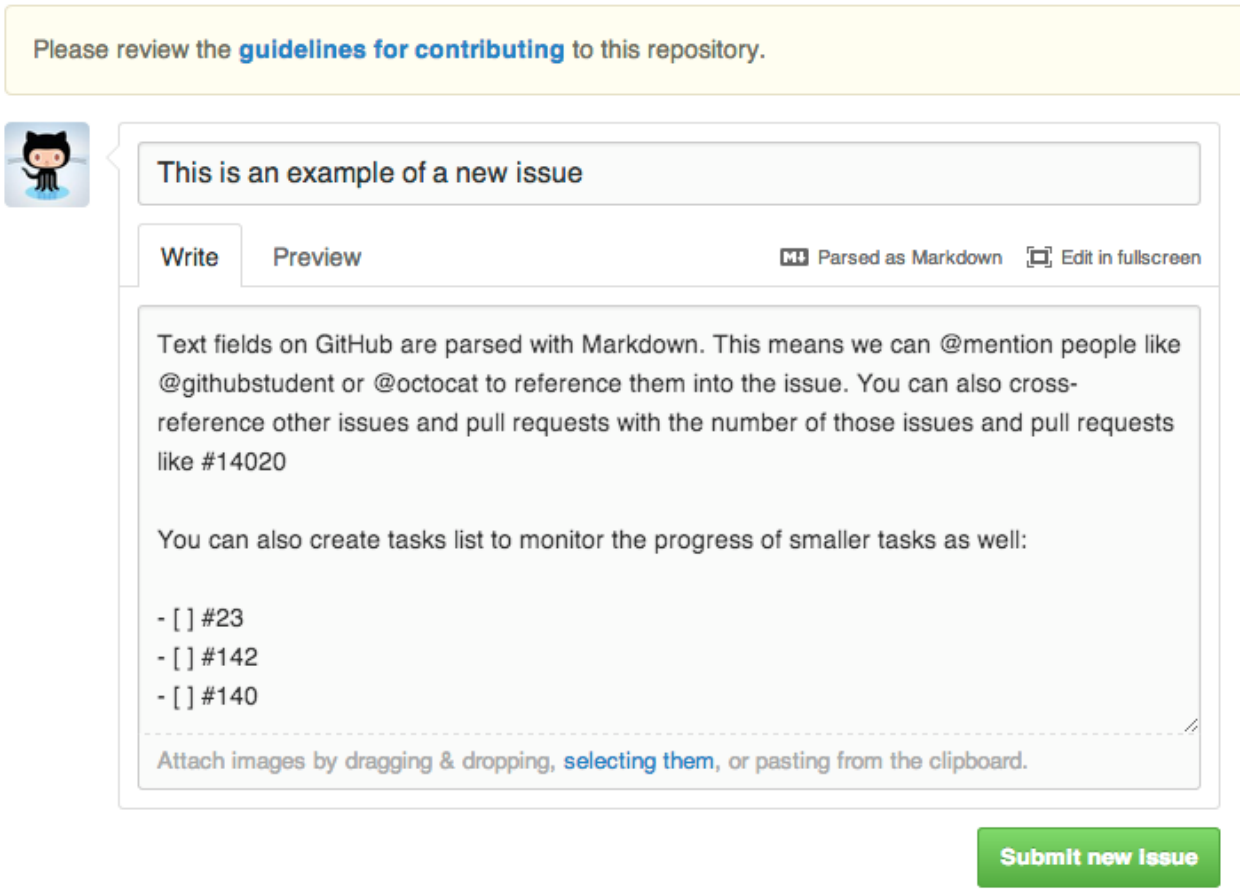


Assignees

Each issue can have an assignee — one person that's responsible for moving the issue forward. Assignees are selected the same way milestones are, through the grey bar at the top of the issue.

Notifications, @mentions, and References

By using @mentions and references inside of Issues, you can notify other GitHub users & teams, and cross-connect issues to each other. These provide a flexible way to get the right people involved to resolve issues effectively, and are easy to learn and use. They work across all text fields on GitHub — they're a part of our text formatting syntax called GitHub Flavored Markdown.

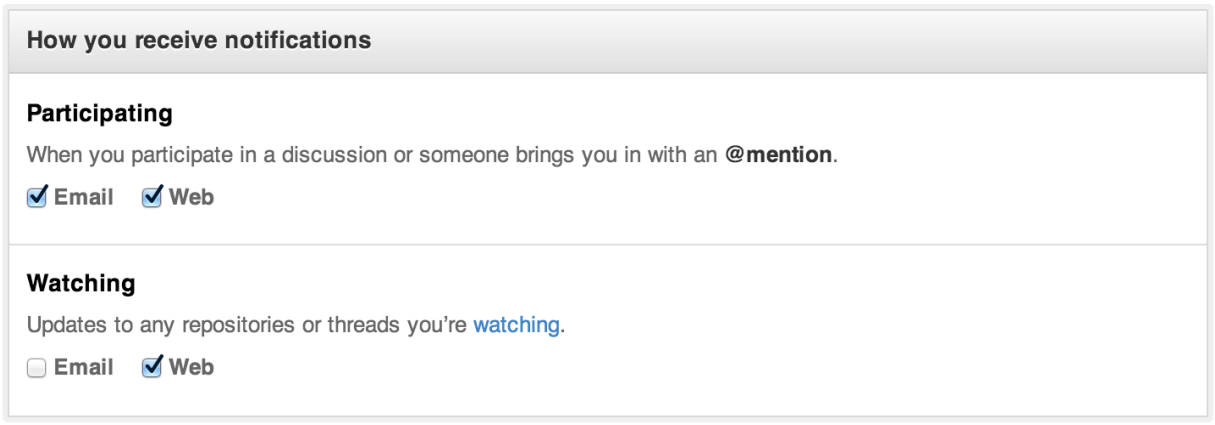


If you'd like to learn more, have a look at Mastering Markdown.

Notifications

Notifications are GitHub's way to keep up to date with your Issues. You can use them to find out about new issues on repositories, or just to know when someone needs your input to move forward on an issue.

There are two ways to receive notifications: via email, and via the web. You can configure how you receive notifications in your settings. If you plan on receiving a lot of notifications, we like to recommend that you receive web + email notifications for **Participating** and web notifications for **Watching**.



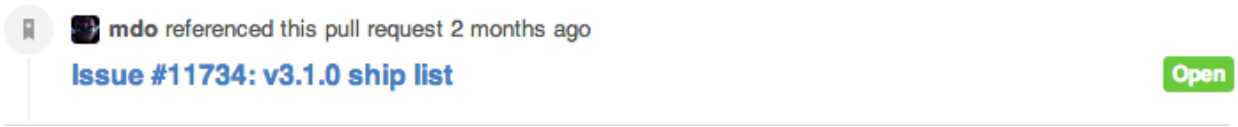
With these settings, you receive emails when people specifically mention you, then visit the web-based interface to keep up to date with repositories you're interested in.

You can access your notifications through the notifications screen. This screen is nice for scanning many notifications at once and marking them as read or muting the thread. Try using keyboard shortcuts to speed up your workflow here — press ? on the page to see which shortcuts are available.

Muted threads won't show up as unread again until you are specifically @mentioned again. This makes muting a great strategy for threads that you have little interest in (perhaps a sub-system that you aren't familiar with). If you mark an issue as read, it will stay that way until someone comments on the thread again.

GitHub also syncs read/unread status for email notifications — if you read a notification in your email client, it will be marked as read in the web-based interface (make sure you allow your email client to display images if you'd like this functionality).

When you do this, we'll create an event inside of issue #42 that looks something like this:



Issue in another repository? Just include the repository before the name like kneath/example-project#42.

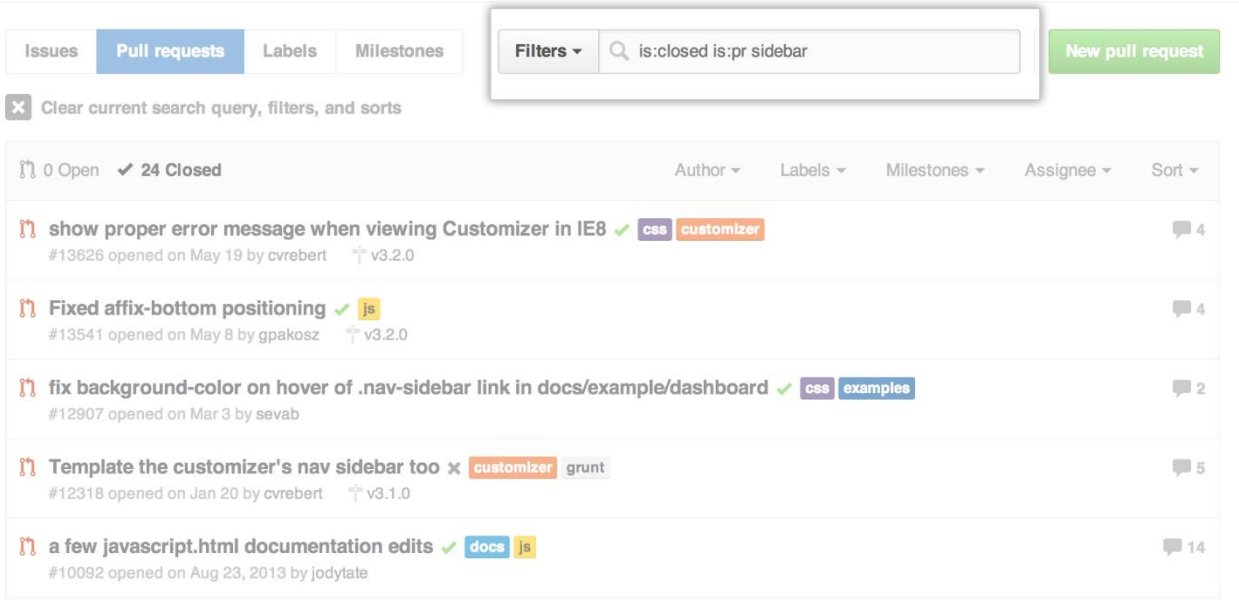
One of the more interesting ways to use GitHub Issues is to reference issues directly from commits. Include the issue number inside of the commit message.

By prefacing your commits with “Fixes”, “Fixed”, “Fix”, “Closes”, “Closed”, or “Close” when the commit is merged into main, it will also automatically close the issue.

References make it possible to deeply connect the work being done with the bug being tracked, and are a great way to add visibility into the history of your project.

Search

At the very top of each page is a search box that lets you search through issues.



Overviews & Reports

Outside of the Issues section, there are two other pages that help summarize what’s going on with Issues across your repository and across all of your repositories.

The Issue Dashboard

If you’re looking for a broader listing of all of your issues across many projects, the Issues Dashboard can be a great tool. The dashboard works very similar to the issues section, but collects issues differently:

All issues in repositories you own and collaborate on

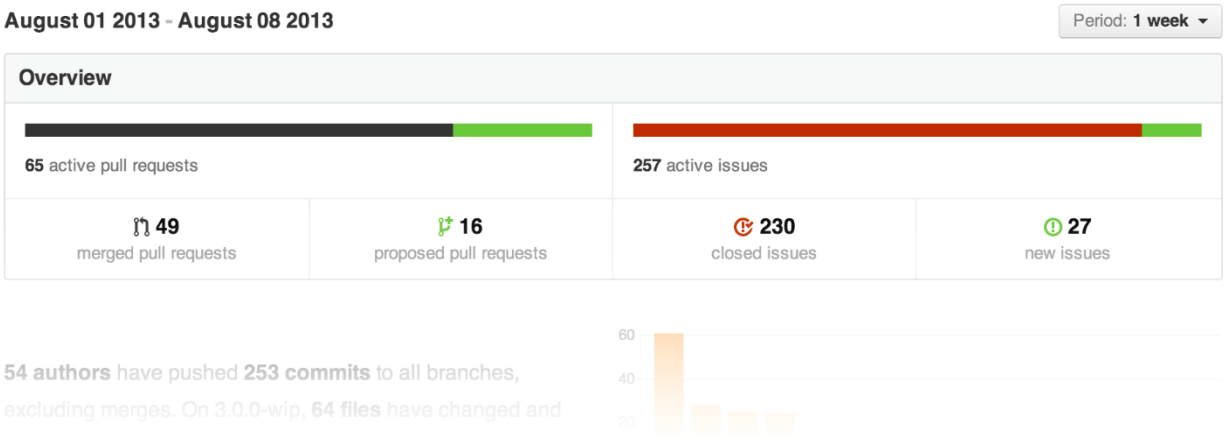
Issues assigned to you

Issues you've created

If you use organizations, each one has its own Issues dashboard that separates out Issues within the organization.

Pulse

Underneath each repository is a section called **Pulse** — Pulse is a snapshot of everything that's happened in the repository in the past week (or day, or past 3 months, etc).



It's a great way to catch up with repositories when you've been away and don't want the granularity notifications offer when watching a repository.

Documenting your projects on GitHub

Good documentation is key to the success of any project. Making documentation accessible enables people to learn about a project; making it easy to update ensures that documentation stays relevant.

Two common ways to document a project are *README files* and *wikis*:

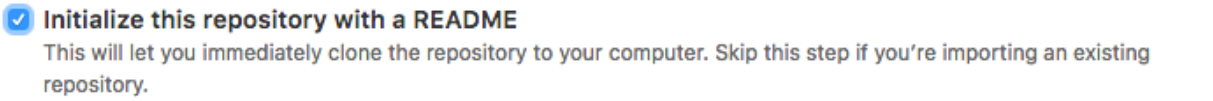
README files are a quick and simple way for other users to learn more about your work.

Wikis on GitHub help you present in-depth information about your project in a useful way.

It's a good idea to at least have a README on your project, because it's the first thing many people will read when they first find your work.

Creating your README

When you create a new repository though GitHub, select "Initialize this repository with a README" unless you plan to import an existing repository.



Your README.md file is now available for editing in your brand-new repository. Your project's name is at the top, followed by any description you chose to include when creating the repository. READMEs are easy to modify, both on GitHub or locally. Check out the Mastering Markdown guide to learn more about how to modify the text within the file after you've made it.

Formatting your README

READMEs generally follow one format in order to immediately orient developers to the most important aspects of your project.

Project name: Your project's name is the first thing people will see upon scrolling down to your README, and is included upon creation of your README file.

Description: A description of your project follows. A good description is clear, short, and to the point. Describe the importance of your project, and what it does.

Table of Contents: Optionally, include a table of contents in order to allow other people to quickly navigate especially long or detailed READMEs.

Installation: Installation is the next section in an effective README. Tell other users how to install your project locally. Optionally, include a gif to make the process even more clear for other people.

Usage: The next section is usage, in which you instruct other people on how to use your project after they've installed it. This would also be a good place to include screenshots of your project in action.

Contributing: Larger projects often have sections on contributing to their project, in which contribution instructions are outlined. Sometimes, this is a separate file. If you have specific contribution preferences, explain them so that other developers know how to best contribute to your work. To learn more about how to help others contribute, check out the guide for setting guidelines for repository contributors.

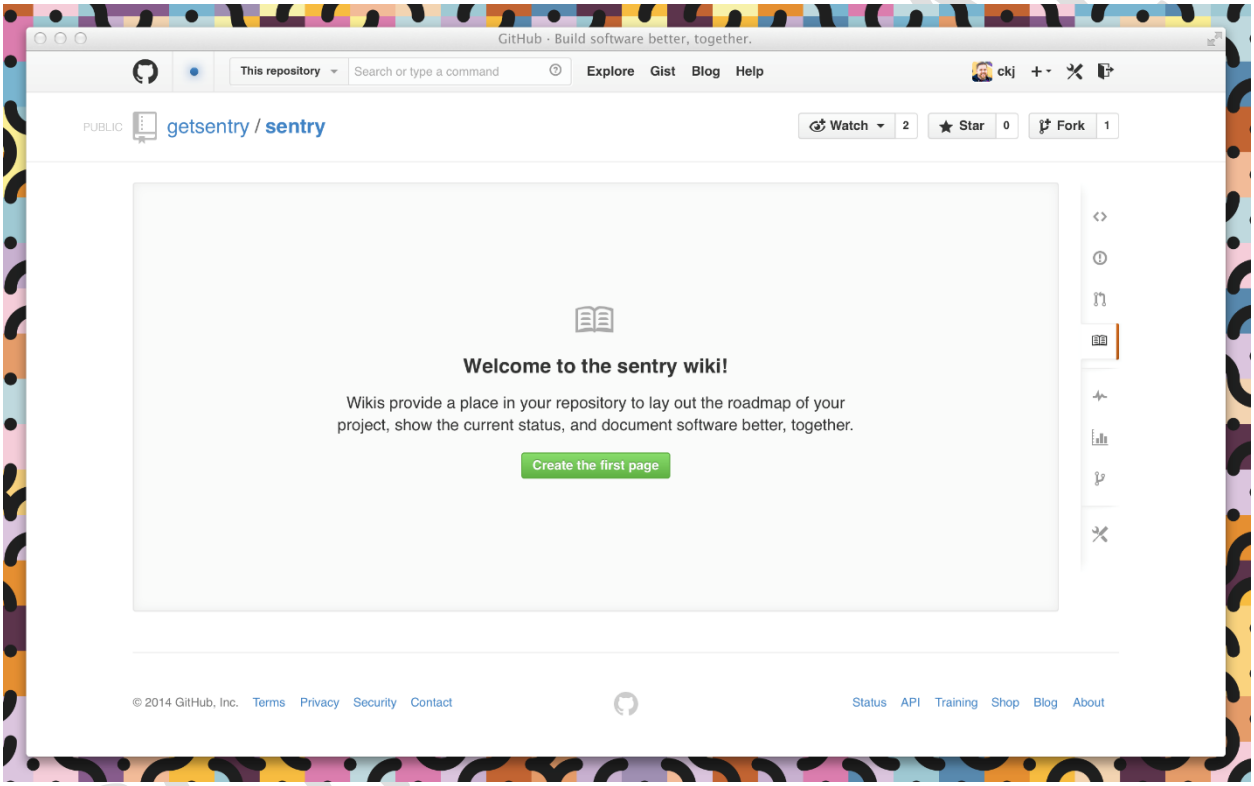
Credits: Include a section for credits in order to highlight and link to the authors of your project.

License: Finally, include a section for the license of your project. For more information on choosing a license, check out GitHub's licensing guide!

Your README should contain only the necessary information for developers to get started using and contributing to your project. Longer documentation is best suited for wikis, outlined below.

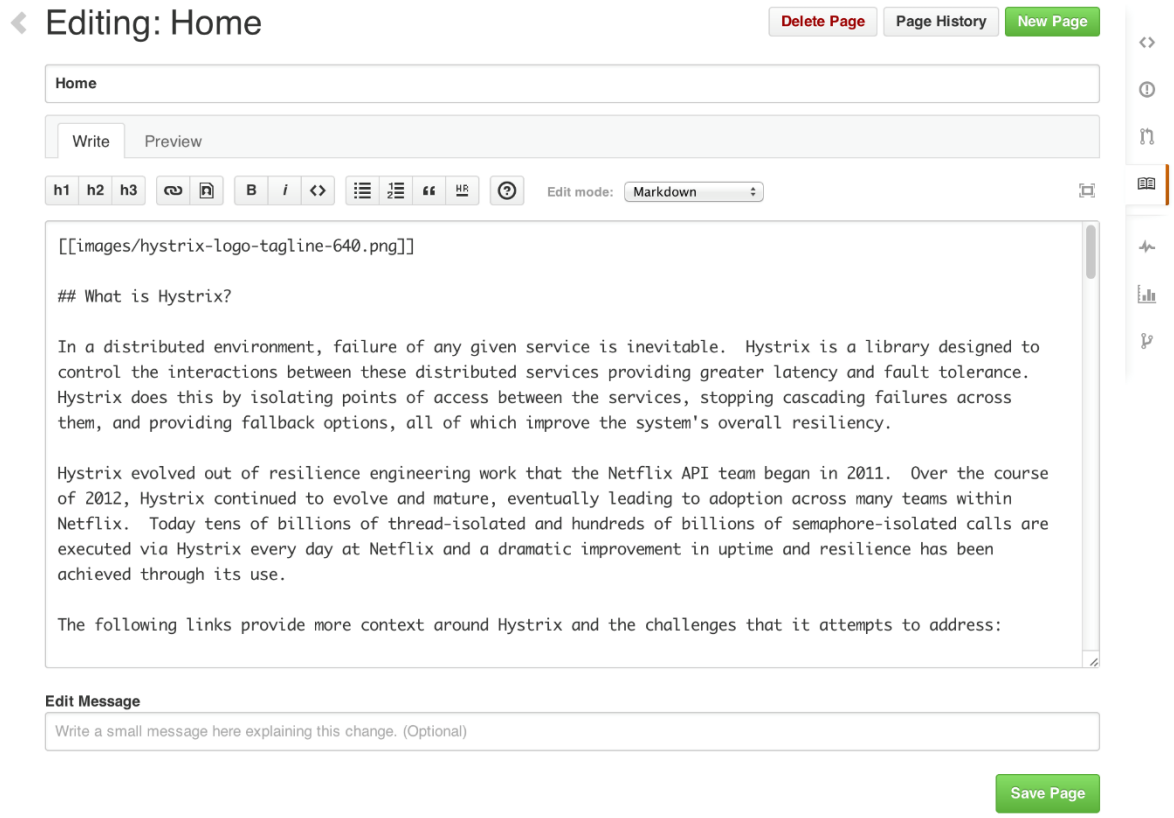
Creating your wiki

Every repository on GitHub comes with a wiki. After you've created a repository, you can set up the included wiki through the sidebar navigation. Starting your wiki is simply a matter of clicking the wiki button and creating your first page.



Adding content

Wiki content is designed to be easily editable. You can add or change content on any wiki page by clicking the **Edit** button located in the upper right corner of each page. This opens up the wiki editor.

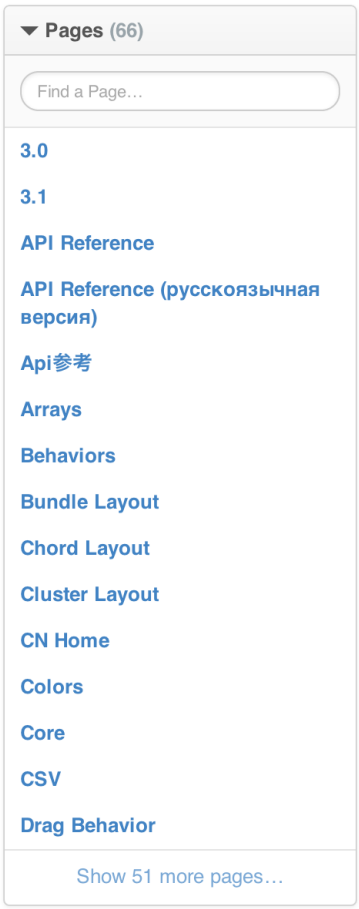


Wiki pages can be written in any format supported by GitHub Markup. Using the drop-down menu in the editor, you can select the format of your wiki, and then use wiki toolbar to create and include content on a page. Wikis also give you the option of including a custom footer where you can list things like contact details or license information for your project.

GitHub keeps track of changes made to each page in your wiki. Below a page title, you can see who made the most recent edits, in addition to the number of commits made to the page. Clicking on this information will take you to the full page history where you can compare revisions or see a detailed list of edits over time.

Adding pages

You can add additional pages to your wiki by selecting **New Page** in the upper right corner. By default, each page you create is included automatically in your wiki's sidebar and listed in alphabetical order.



t library for manipulating documents based on data. **D3** helps you bring data to
G and CSS. D3's emphasis on web standards gives you the full capabilities of
ithout tying yourself to a proprietary framework, combining powerful
nents and a data-driven approach to DOM manipulation.

es

ery
alks

Overflow
roup

(русскоязычная версия)

Support

You can also add a custom sidebar to your wiki by clicking the **Add custom sidebar** link. Custom sidebar content can include text, images, and links.

Note: The page called "Home" functions as the entrance page to your wiki. If it is missing, an automatically generated table of contents will be shown instead.

If you're knowledgable with the command line, you can also modify wikis locally. Check out our help article for more info.

Syntax highlighting

Wiki pages support automatic syntax highlighting of code for a wide range of languages by using the following syntax:

```ruby
def foo
  puts 'bar'
```

```
  end
```

The block must start with three backticks, optionally followed by the the name of the language that is contained by the block.

The block contents should be indented at the same level as the opening backticks. The block must end with three backticks indented at the same level as the opening backticks.