Version Control System

**Version Control System (VCS)** is a software that helps software developers to work together and maintain a complete history of their work.

Listed below are the functions of a VCS −

* Allows developers to work simultaneously.
* Does not allow overwriting each other’s changes.
* Maintains a history of every version.

Following are the types of VCS −

* Centralized version control system (CVCS).
* Distributed/Decentralized version control system (DVCS).

AD

Distributed Version Control System

Centralized version control system (CVCS) uses a central server to store all files and enables team collaboration. But the major drawback of CVCS is its single point of failure, i.e., failure of the central server. Unfortunately, if the central server goes down for an hour, then during that hour, no one can collaborate at all. And even in a worst case, if the disk of the central server gets corrupted and proper backup has not been taken, then you will lose the entire history of the project. Here, distributed version control system (DVCS) comes into picture.

DVCS clients not only check out the latest snapshot of the directory but they also fully mirror the repository. If the server goes down, then the repository from any client can be copied back to the server to restore it. Every checkout is a full backup of the repository. Git does not rely on the central server and that is why you can perform many operations when you are offline. You can commit changes, create branches, view logs, and perform other operations when you are offline. You require network connection only to publish your changes and take the latest changes.

Advantages of Git

Free and open source

Git is released under GPL’s open source license. It is available freely over the internet. You can use Git to manage property projects without paying a single penny. As it is an open source, you can download its source code and also perform changes according to your requirements.

Fast and small

As most of the operations are performed locally, it gives a huge benefit in terms of speed. Git does not rely on the central server; that is why, there is no need to interact with the remote server for every operation. The core part of Git is written in C, which avoids runtime overheads associated with other high-level languages. Though Git mirrors entire repository, the size of the data on the client side is small. This illustrates the efficiency of Git at compressing and storing data on the client side.

Implicit backup

The chances of losing data are very rare when there are multiple copies of it. Data present on any client side mirrors the repository, hence it can be used in the event of a crash or disk corruption.

Security

Git uses a common cryptographic hash function called secure hash function (SHA1), to name and identify objects within its database. Every file and commit is check-summed and retrieved by its checksum at the time of checkout. It implies that, it is impossible to change file, date, and commit message and any other data from the Git database without knowing Git.

No need of powerful hardware

In case of CVCS, the central server needs to be powerful enough to serve requests of the entire team. For smaller teams, it is not an issue, but as the team size grows, the hardware limitations of the server can be a performance bottleneck. In case of DVCS, developers don’t interact with the server unless they need to push or pull changes. All the heavy lifting happens on the client side, so the server hardware can be very simple indeed.

Easier branching

CVCS uses cheap copy mechanism, If we create a new branch, it will copy all the codes to the new branch, so it is time-consuming and not efficient. Also, deletion and merging of branches in CVCS is complicated and time-consuming. But branch management with Git is very simple. It takes only a few seconds to create, delete, and merge branches.

DVCS Terminologies

Local Repository

Every VCS tool provides a private workplace as a working copy. Developers make changes in their private workplace and after commit, these changes become a part of the repository. Git takes it one step further by providing them a private copy of the whole repository. Users can perform many operations with this repository such as add file, remove file, rename file, move file, commit changes, and many more.

Working Directory and Staging Area or Index

The working directory is the place where files are checked out. In other CVCS, developers generally make modifications and commit their changes directly to the repository. But Git uses a different strategy. Git doesn’t track each and every modified file. Whenever you do commit an operation, Git looks for the files present in the staging area. Only those files present in the staging area are considered for commit and not all the modified files.

Let us see the basic workflow of Git.

**Step 1** − You modify a file from the working directory.

**Step 2** − You add these files to the staging area.

**Step 3** − You perform commit operation that moves the files from the staging area. After push operation, it stores the changes permanently to the Git repository.



Suppose you modified two files, namely “sort.c” and “search.c” and you want two different commits for each operation. You can add one file in the staging area and do commit. After the first commit, repeat the same procedure for another file.

# First commit

[bash]$ git add sort.c

# adds file to the staging area

[bash]$ git commit –m “Added sort operation”

# Second commit

[bash]$ git add search.c

# adds file to the staging area

[bash]$ git commit –m “Added search operation”

Blobs

Blob stands for **B**inary **L**arge **Ob**ject. Each version of a file is represented by blob. A blob holds the file data but doesn’t contain any metadata about the file. It is a binary file, and in Git database, it is named as SHA1 hash of that file. In Git, files are not addressed by names. Everything is content-addressed.

Trees

Tree is an object, which represents a directory. It holds blobs as well as other sub-directories. A tree is a binary file that stores references to blobs and trees which are also named as **SHA1** hash of the tree object.

Commits

Commit holds the current state of the repository. A commit is also named by **SHA1** hash. You can consider a commit object as a node of the linked list. Every commit object has a pointer to the parent commit object. From a given commit, you can traverse back by looking at the parent pointer to view the history of the commit. If a commit has multiple parent commits, then that particular commit has been created by merging two branches.

Branches

Branches are used to create another line of development. By default, Git has a master branch, which is same as trunk in Subversion. Usually, a branch is created to work on a new feature. Once the feature is completed, it is merged back with the master branch and we delete the branch. Every branch is referenced by HEAD, which points to the latest commit in the branch. Whenever you make a commit, HEAD is updated with the latest commit.

Tags

Tag assigns a meaningful name with a specific version in the repository. Tags are very similar to branches, but the difference is that tags are immutable. It means, tag is a branch, which nobody intends to modify. Once a tag is created for a particular commit, even if you create a new commit, it will not be updated. Usually, developers create tags for product releases.

Clone

Clone operation creates the instance of the repository. Clone operation not only checks out the working copy, but it also mirrors the complete repository. Users can perform many operations with this local repository. The only time networking gets involved is when the repository instances are being synchronized.

Pull

Pull operation copies the changes from a remote repository instance to a local one. The pull operation is used for synchronization between two repository instances. This is same as the update operation in Subversion.

Push

Push operation copies changes from a local repository instance to a remote one. This is used to store the changes permanently into the Git repository. This is same as the commit operation in Subversion.

HEAD

HEAD is a pointer, which always points to the latest commit in the branch. Whenever you make a commit, HEAD is updated with the latest commit. The heads of the branches are stored in **.git/refs/heads/** directory.

[CentOS]$ ls -1 .git/refs/heads/

master

[CentOS]$ cat .git/refs/heads/master

570837e7d58fa4bccd86cb575d884502188b0c49

Revision

Revision represents the version of the source code. Revisions in Git are represented by commits. These commits are identified by **SHA1** secure hashes.

URL

URL represents the location of the Git repository. Git URL is stored in config file.

[tom@CentOS tom\_repo]$ pwd

/home/tom/tom\_repo

[tom@CentOS tom\_repo]$ cat .git/config

[core]

repositoryformatversion = 0

filemode = true

bare = false

logallrefupdates = true

[remote "origin"]

url = gituser@git.server.com:project.git

fetch = +refs/heads/\*:refs/remotes/origin/\*

# Version Control

Version control is a tool that allows you to keep track of changes to a number of files. Using version control for package development means that you can easily revert to previous package versions, collaborate with multiple developers, and record reasons for the changes that are made.

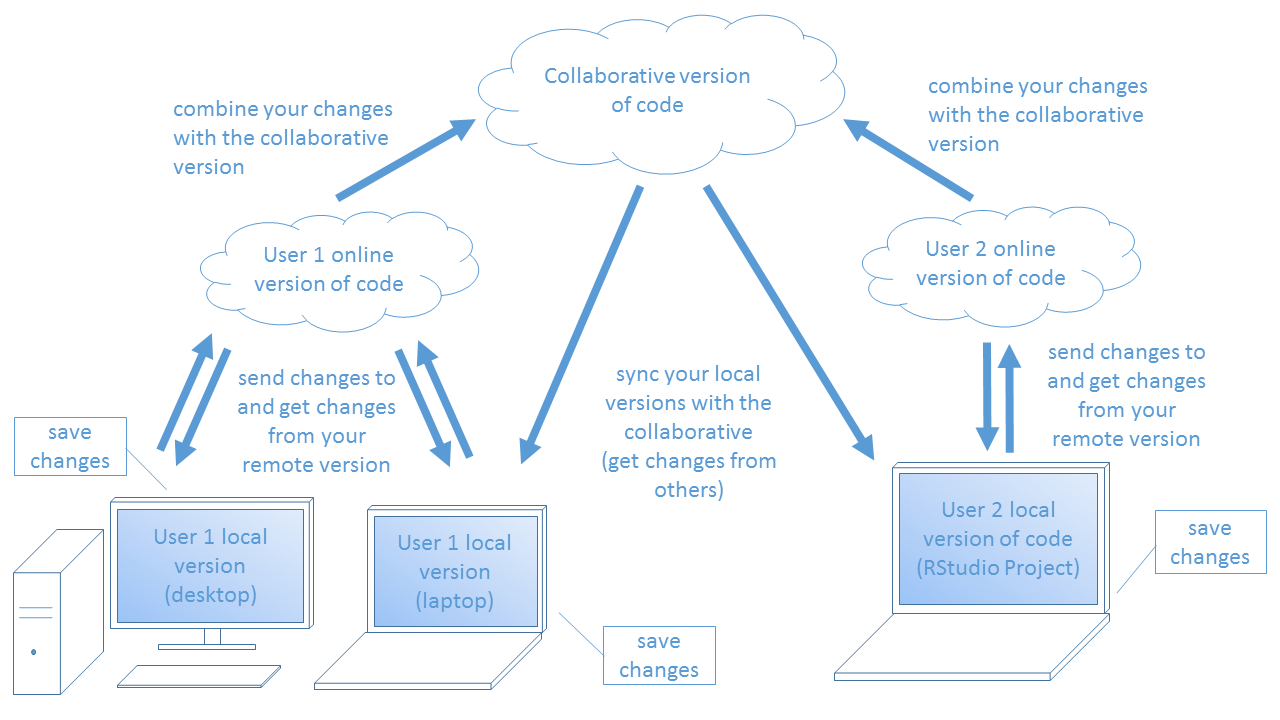
1. Define version control and give examples of how it is useful.
2. Navigate the GitHub interface.
3. Summarize a typical GitHub-to-R workflow.

## Why version control?

Version control systems allow you to have organized code repositories by tracking changes. There are many version control systems, but we will only be covering Git in this course. Git can track every change made to a file, annotate the change, and keep record of the change through time. Git maintains a history of the code base and allows you to revert to previous versions if necessary. Git refers to the version control language and commands are typed into the terminal starting with the word git.

A web interface called GitHub allows users to visually see their tracked changes and has additional features, such as issues, milestones, review requests, and commenting. With GitHub, changes to code can be associated with bugs and feature requests. GitHub also enables open science practices by sharing what goes on “behind-the-scenes” in the code. In addition, GitHub is a great tool for collaborative work because issues, comments, and peer reviews can be associated with a specific GitHub user account. Each user can edit the code at the same time and handle conflicts appropriately.

In this course, we will be using Git and GitHub in conjunction with RStudio to complete version control workflows. Below is a generic depiction of what a version control workflow might be.



There is a main version of the code that people are collaboratively developing. Each contributor has their own version of this code online and locally. Changes are made locally, sent to their online version, and then combined with the collaborative version of the code. Contributors are able to get the changes from other users by syncing their local version with the collaborative version of the code. That is the main concept of version control.

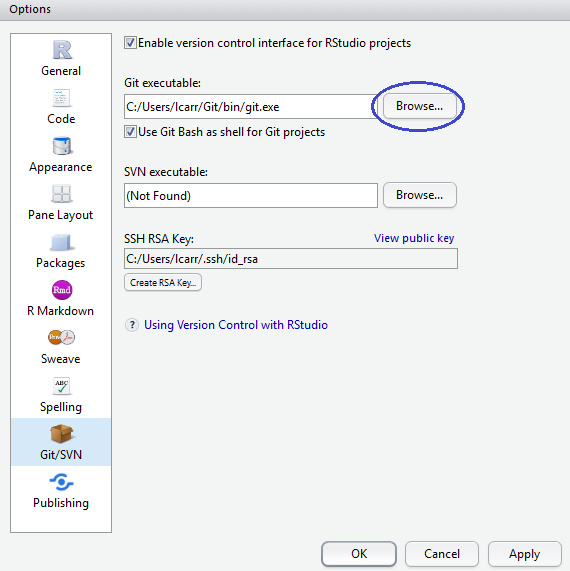
## Setting up Git to work with RStudio

To use Git for version control, you will need to download it, configure RStudio, and should get set up with SSH keys. SSH keys allow you to connect to GitHub without specifying your username and password each time. Follow these steps before continuing with this lesson. [This RStudio blog post](https://support.rstudio.com/hc/en-us/articles/200532077-Version-Control-with-Git-and-SVN) is another good resource for installing Git and using SSH.

1. Download and install [Git](https://git-scm.com/downloads).
2. Tell RStudio where to find the Git executable. See [details below](https://owi.usgs.gov/R/training-curriculum/r-package-dev/git/index.html#config-git)
3. Create an SSH key following [these instructions](https://help.github.com/articles/adding-a-new-ssh-key-to-your-github-account/).

#### ****Configure Git in RStudio****

Once you have downloaded and installed Git, you will need to tell RStudio where it can be found. In RStudio, navigate to Tools >> Global Options, and then click the Git/SVN tab. At the top, there is a place for the filepath of the Git executable. Click Browse and select your Git executable (a .exe file extension).



## Git/GitHub Definitions

Here are some terms to be familiar with as we go through our recommended version control workflow.

| Table 1. Common Git and GitHub definitions | |
| --- | --- |
| **Term** | **Definition** |
| repository | a collection of files, aka repo |
| fork | a user's version of the original repository |
| commit | saved change(s) to code |
| pull request | a request to merge changes from one repository to another, aka PR |
| branch | specific instance of your fork; there can be more than one branch on any fork (all will have 'master' which is the main branch) |
| upstream | used to refer to the original repository |
| origin | used to refer to your forked repository |
| master | refers to the main branch in a repository |

## 

## Recomended workflow

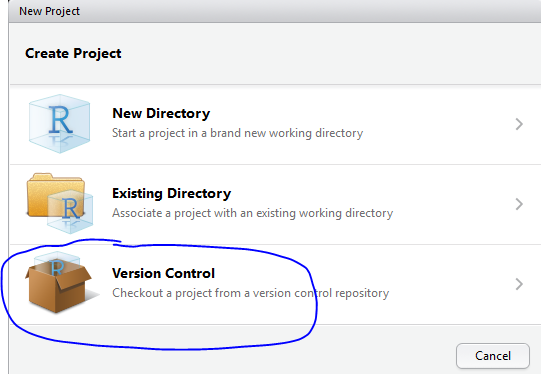
There are many ways to use Git, GitHub, and RStudio in your version control workflow. We will discuss the method USGS-R has predominately used. It is most similar to the “fork-and-branch” workflow (see the [additional resources](https://owi.usgs.gov/R/training-curriculum/r-package-dev/git/index.html#additional-resources) section below). There are three locations of the repository: 1) canonical on GitHub (“upstream”), 2) forked repository on GitHub (“origin”), and 3) the user’s local repository.

### Initial version control setup for a project

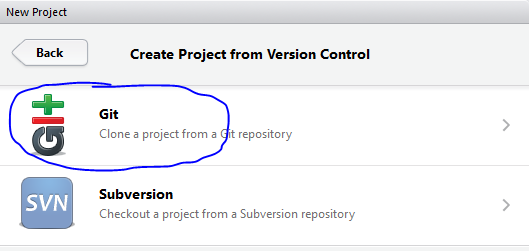
The initial setup requires a canonical repository on GitHub. To create a new repo on GitHub, follow [these instructions](https://help.github.com/articles/creating-a-new-repository/). Once there is a canonical repository, the user looking to contribute to this code base would Fork the repository to their own account.

What button to press for forking a repo

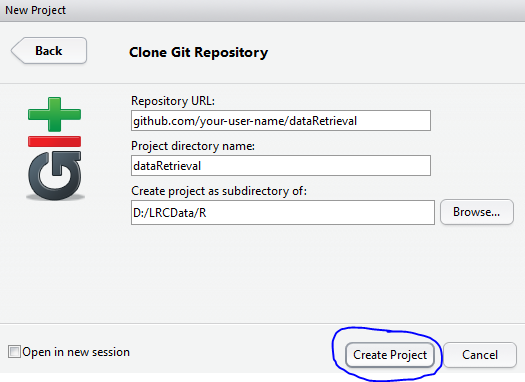
Next, the user would create the local version of the forked repo in an RStudio project. When creating a new RStudio project, select Version Control,



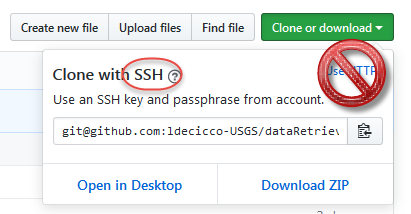
then select Git,



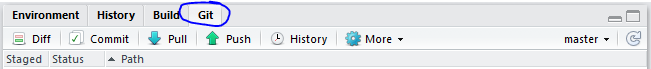
and then paste the SSH address to the forked repository. It should automatically populate the Project directory name from the key.



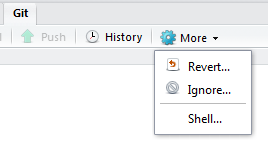
To find the SSH address, click “Clone or download” on the GitHub repo.



Then you can select “Create Project” and it will open a new RStudio project. You should see a new tab in the environment pane that you have not seen before called “Git”.



Next, you need to setup your local repository to recognize the main repository as the “upstream” version. To do this, click the “More” drop down in your RStudio Git tab, then select “Shell…”.



In the command prompt, type git remote -v and hit enter. This will show you which remote repositories (available on online) are connected to your local repository. You should initially only see your forked repository and it is labeled “origin”. To add the main repo as an “upstream” repository, type git remote add upstream <SSH address> with the correct SSH address and hit enter.

E.g. git remote add upstream git@github.com:USGS-R/dataRetrieval.git. Now when you run git remote -v in the shell, you should see both origin and upstream listed. This means you are set up to get new changes from the main repo and add your local changes to your remote repo. We will see how to do that next.

### Workflow for every change you want to submit

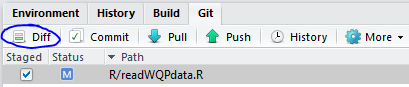
Now that you have the three repositories set up, you can start making changes to the code and commit them. First, you would make a change to a file or files in RStudio. When you save the file(s), you should see them appear in the Git tab. A blue “M” icon next to them means they were existing files that you modified, a green “A” means they are new files you added, and a red “D” means they were files that you deleted.

#### ****Getting upstream changes****

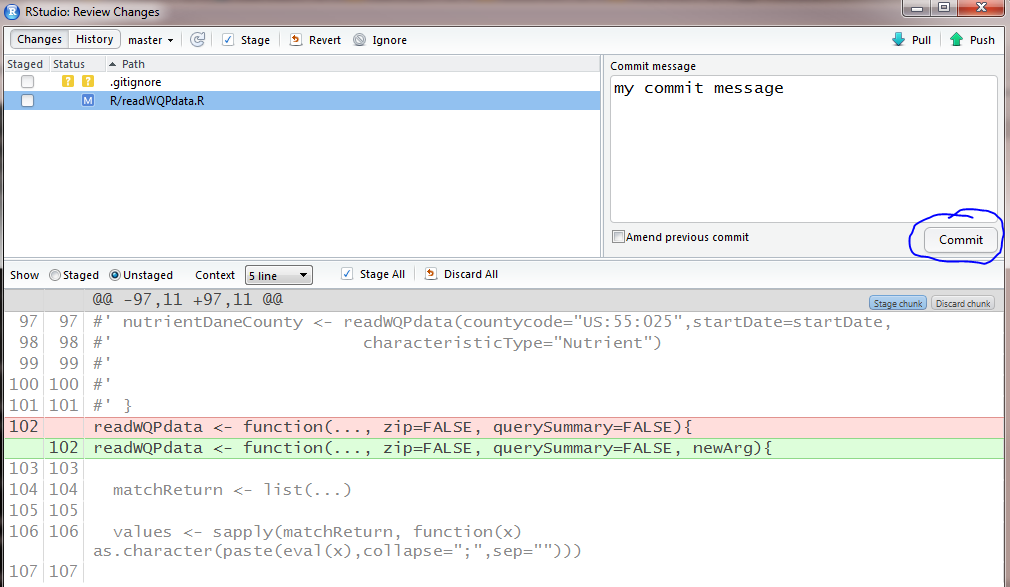
To get changes available on the remote canonical fork to your local repository, you will need to “pull” those changes down. To do this, go to the Git shell through RStudio (Git tab >> More >> Shell) and use the command git pull with the name of the remote fork followed by the name of your local repo, e.g. git pull upstream master. It is generally a good idea to do this before you start making changes to avoid [conflicts](https://owi.usgs.gov/R/training-curriculum/r-package-dev/git/index.html#handling-merge-conflicts).

#### ****Committing changes****

Click the check box next to the file(s) you would like to commit. To view the changes, select “Diff”.

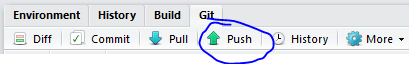


You can select the different files and it will show what was added (highlighted green) and what was deleted (highlighted red). Then, type your message about the commit and click “Commit”.



#### ****Pushing local changes to a remote fork****

It’s best to keep commits as concise and specific as possible. So, commit often and with useful messages. When you are ready to add these changes to the main repository, you need to create a pull request. First, push your changes to your remote fork (aka origin). Either use the “push” button in RStudio (this only works when you are on your master branch) OR type the git command into the shell.



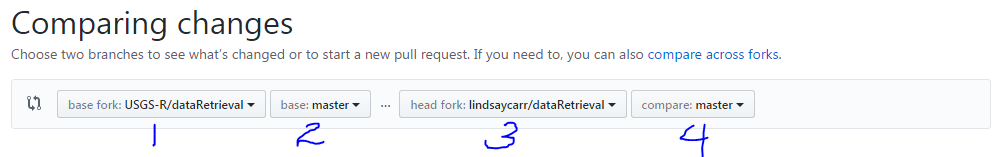
To get to the shell, go to the “Git” tab, then click “More”, and then “Shell…”. Now type your git command specifying where changes are going, and which repository is being pushed: git push origin master will push commits from the local repo (“master”) to your remote repo on GitHub (“origin”).

#### ****Submitting a pull request****

To submit a pull request, you need to be on your remote fork’s GitHub page. The URL would say github.com/YOUR\_USERNAME/REPO\_NAME, e.g. github.com/lindsaycarr/dataRetrieval. It also shows where your repo was forked from:

Example of name on remote fork URL

From this page, click “New pull request”. Now, you should have a screen that is comparing your changes. Double check that the left repo name (1 in the figure) is the canonical repository that you intend to merge your changes into. Then double check that the fork you are planning to merge is your remote fork (3 in the figure). For now, branches should both be “master” (2 and 4 in the figure). See [the section on branching to learn more](https://owi.usgs.gov/R/training-curriculum/r-package-dev/git/index.html#branching).

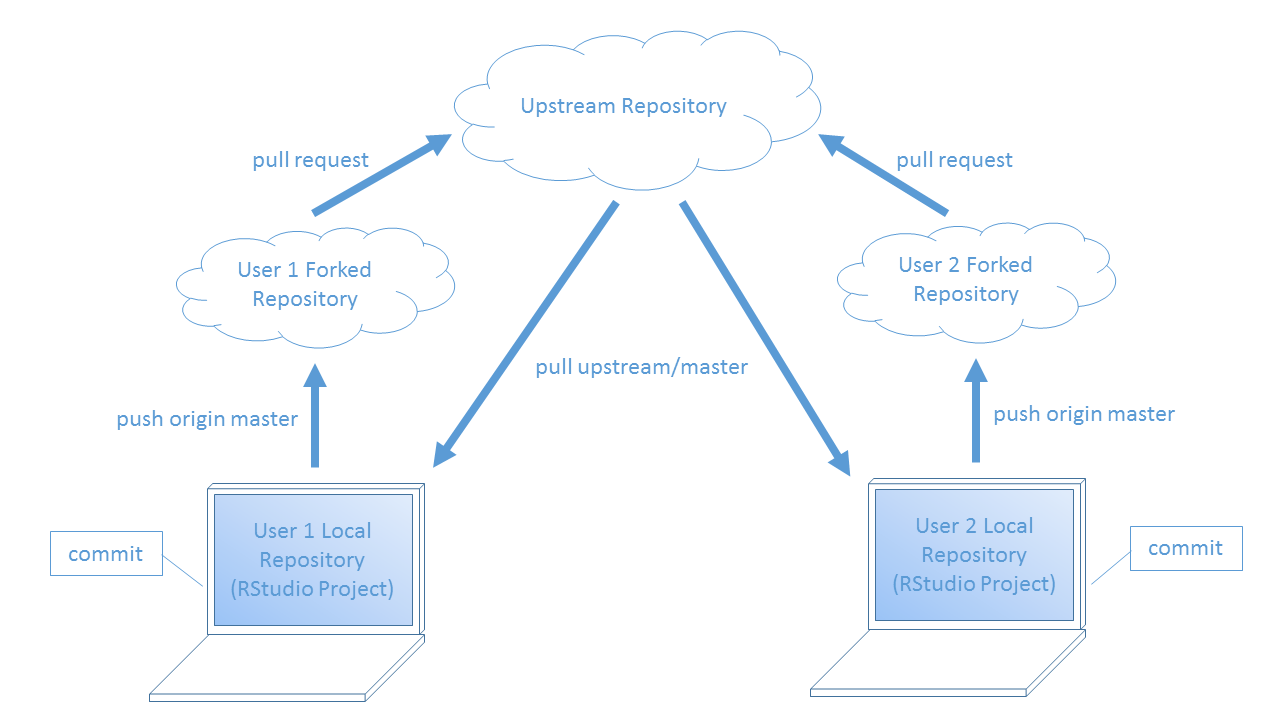


Once you have verified that you are merging the correct forks and branches, you can select “Create Pull Request”. Be sure to describe your changes sufficiently (see [this wiki](https://github.com/erlang/otp/wiki/Writing-good-commit-messages) for more tips):

* add a title and comments
* include information for whoever reviews this, e.g. what should it do in order for them to approve it?
* link to any existing issues or related pull requests by typing # and the number of the issue or PR

Now, you wait while someone else reviews and merges your PR. To learn how to merge a pull request, see the [section on reviewing code changes](https://owi.usgs.gov/R/training-curriculum/r-package-dev/git/index.html#code-review). You should avoid merging your own pull requests, and instead should always have a peer review of your code.

#### ****Commit workflow overview****



## Handling merge conflicts

Even though Git and GitHub make simultaneous code development easier, it is not entirely fool-proof. If a code line you are working on was edited by someone else since the last time you synced with the upstream branch, you might run into “merge conflicts”. When you encounter conflicts during a pull from the upstream repo, you will see all changed files since the previous sync in your Git tab. Files with checkmarks are just fine. Any file with a filled in checkbox means that only part of the changes are being committed - this is where you have merge conflicts.

When you open the file(s) with merge conflicts, look for the section that looks like this:

<<<<<<< HEAD

some code

some code

some code

=======

your code

your code

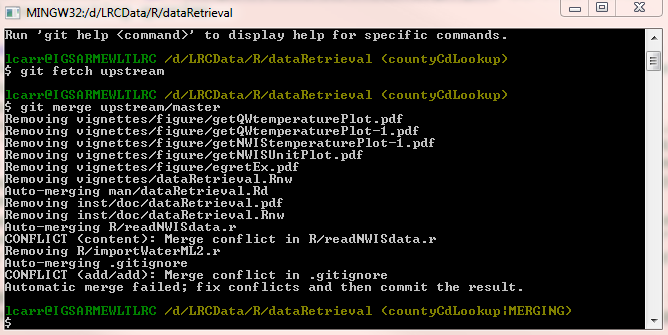
your code

>>>>>>> upstream/master

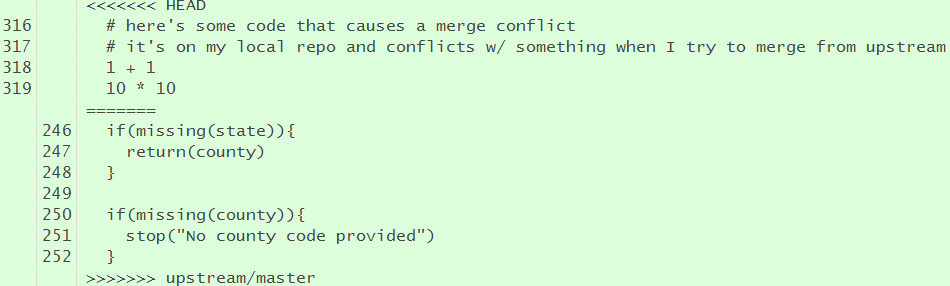
The chunk of code wrapped in <<<<<<< HEAD and ======= (the first chunk) is the code that exists in the local repository. The chunk of code wrapped in ======= and then >>>>>>> upstream/master (the second chunk) is the code from upstream that you are trying to merge. To reconcile these differences, you need to pick which code you are keeping and which you aren’t. Once you correctly edit the code, make sure to delete the conflict markers (<<<<<<< HEAD, =======, and >>>>>>> upstream/master). Then, save the file.

Now that you’ve addressed the merge conflict in the file, it’s time to commit those changes. All the non-conflicted files should still have a checkmark next to them in the Git tab. Check the box next to your reconciled file and select commit. Add a message about these changes, such as “merged conflicts” or something similar. Then commit. Now, you should be back on track to continue your edits.

Here’s an example. When you try to merge upstream with your local code, the shell will say something similar to CONFLICT ... Merge conflict in [filename].



The actual code will indicate the conflicting lines. Something similar to:



The Git tab will also indicate what files have conflicts by a colored-in check box.



Once you edit and save the file, just check the box and commit along with everything else that is in the Git tab. Usually, the commit message can just say “Merging conflics”.

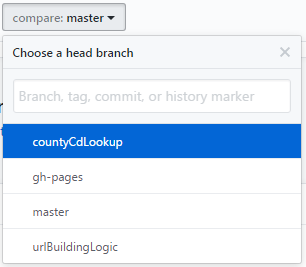
## Branching

Branches are an optional feature of Git version control. It allows you to have a non-linear commit history where multiple features/bug fixes could be developed and merged independently. You have been working on the “master” branch for the previous sections. We could add another branch off of this called “bug-fix”, and another from the master called “new-feature”. You could change the code on either of those branches independed of one another, and merge when one is done without the need to have the other completed at the same time. Be careful though, you can create branches from a non-master branch which is often not the behavior you want. Just be aware of your current branch when you are creating a new one.

When the time comes to merge the branch, you can either merge it with the master branch locally or create a pull request to the main repository specifying changes from your new branch. Follow [this blog](https://nicercode.github.io/git/branches.html) to learn how to do the former method.

If you’d prefer the latter method, follow the blog until the “Merging branches back together” section. Instead when you’re ready to merge your branch with the main repository through a PR, follow these instructions.

1. Open the Git shell window and push your local branch to your remote fork via the command git push origin/new-branch-name.
2. On GitHub, go to your remote fork page and click “New pull request”.
3. As noted in the [section on submitting a pull request](https://owi.usgs.gov/R/training-curriculum/r-package-dev/git/index.html#submitting-pr), double check that your repositories and branches are correct on the “Comparing changes” page. The only difference is that you want to change the farthest right drop-down to your branch.
4. Now follow the rest of steps for completing your PR submission as described in the [how-to-submit-a-PR section](https://owi.usgs.gov/R/training-curriculum/r-package-dev/git/index.html#submitting-pr).



## gitignore file

It is sometimes useful to have a text file name .gitignore. This file let’s Git know which files it should not worry about tracking. For RStudio projects, it’s a good idea to have the .Rproj and .Rhistory files specified in a gitignore. You can use \* before a file extension to say any file with that extension should be ignored (including those in sub-folders). Here’s an example of what the .gitignore content might look like:

.Rproj.user

.Rhistory

.RData

\*.Rproj

Mac users might want to consider adding \*.DS\_Store to their .gitignore file. .DS\_Store is a file automatically created containing information about icons and their positions. There is no need to include this in your repo.

## Stashing

If you have uncommitted changes on your local repository and try to pull down updates from the upstream repository, you’ll notice that you get an error message:



If you’re ready, you can go ahead and commit those changes. Then try pulling from upstream again. If you’re not ready to commit these changes, you can “stash” them, pull from upstream, and then bring them back as uncommitted changes.

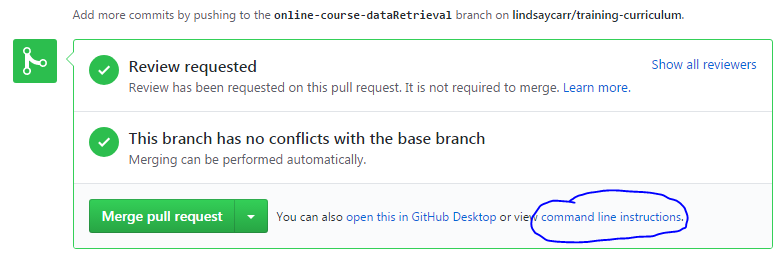
To stash all uncommitted changes, run git stash in your Git shell (Git tab >> More >> Shell). To see what you stashed, run git stash list. It will automatically put you in the VIM text editor mode, so type “q” and hit enter before try to do anything else. To get your stashed changes back, run git stash apply.

That is the basic use of stashing, but there are more complicated ways to stash uncommited changes.

Please visit the [git documentation page on stashing](https://git-scm.com/docs/git-stash) for more information.

## Reviewing code changes

If you are the reviewer for an open pull request, you will likely need to pull down the suggested changes and test them out locally before approving the PR. It’s pretty simple to do this because you can copy and paste git commands for making a new branch of the PR. Next to the “Merge pull request” button, select “command line instructions”.

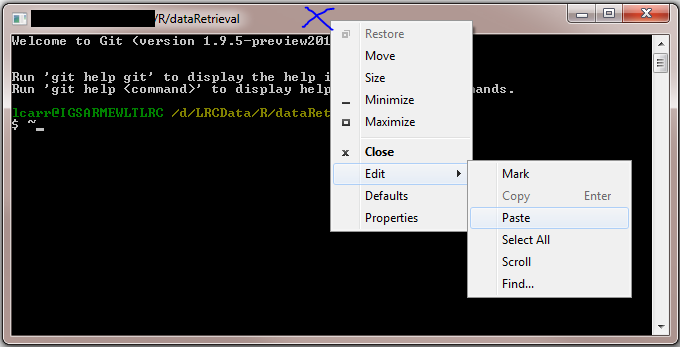


Copy the two git commands from Step 1, From your project repository, check out a new branch and test the changes. Paste these lines into your Git shell (Git tab >> More >> Shell).

git checkout -b otherusername-master master

git pull https://github.com/otherusername/dataRetrieval.git master

You might not be able to right click and paste, or use the CTRL + V method. Instead, right click the top bar of the shell window, hover over “Edit”, then click “Paste”. Once the code is in the shell, hit enter.



RStudio should now have a different branch name in the top right. Before you can test the changes available in this branch, you need to build and reload the package.

Once you have thoroughly tested this branch and approve of it, go back to the PR on GitHub. Write a few comments about what you tested and why you are accepting these changes, then click “Merge pull request” and then “Confirm”. You can now delete the branch you were using to test these changes. Continuing the example for checking out a branch called otherusername-master above,

git branch -d otherusername-master

Don’t forget to pull down these new changes to your local repository master branch!

## Common Git commands

| Table 2. Summary of common git commands | |
| --- | --- |
| **Command** | **Description** |
| git remote -v | view the remote repos linked to this local repository |
| git remote add upstream | add a remote repo at the specified url named 'upstream' |
| git push origin/master | move changes from the local repo (origin) to your remote fork |
| git pull upstream master | get changes from the remote main repository (upstream) and merge with your local repo (master) |
| git checkout -b | create a new branch from the current branch and switch to it |
| git branch -d | delete the specified branch |
| git status | look at what is committed (pushed or not) and not committed |
| git stash | stash all uncommitted changes |
| git stash list | look at what is currently stashed |
| git stash apply | restore all stashed changes to the repo |

# Difference Between Git and GitHub

**Git:** Git is a distributed version control system for tracking changes in source code during software development. It is designed for coordinating work among programmers, but it can be used to track changes in any set of files. Its goals include speed, data integrity, and support for distributed, non-linear workflows.

**GitHub:** GitHub is a web-based Git repository hosting service, which offers all of the distributed revision control and source code management (SCM) functionality of Git as well as adding its own features.

| S.No. | Git | GitHub |
| --- | --- | --- |
| 1. | Git is a software. | GitHub is a service. |
| 2. | Git is a command-line tool | GitHub is a graphical user interface |
| 3. | Git is installed locally on the system | GitHub is hosted on the web |
| 4. | Git is maintained by linux. | GitHub is maintained by Microsoft. |
| 5. | Git is focused on version control and code sharing. | GitHub is focused on centralized source code hosting. |
| 6. | Git is a version control system to manage source code history. | GitHub is a hosting service for Git repositories. |
| 7. | Git was first released in 2005. | GitHub was launched in 2008. |
| 8. | Git has no user management feature. | GitHub has a built-in user management feature. |
| 9. | Git is open-source licensed. | GitHub includes a free-tier and pay-for-use tier. |
| 10. | Git has minimal external tool configuration. | GitHub has an active marketplace for tool integration. |
| 11. | Git provides a Desktop interface named Git Gui. | GitHub provides a Desktop interface named GitHub Desktop. |
| 12. | Git competes with CVS, Azure DevOps Server, Subversion, Mercurial, etc. | GitHub competes with GitLab, Git Bucket, AWS Code Commit, etc. |

# Difference Between GitLab and GitHub

**GitLab:** GitLab is a repository hosting manager tool that is developed by GitLab Inc and is used for the software development process. It provides a variety of management by which we can streamline our collaborative workflow for completing the software development lifecycle. It also allows us to import the repository from Google Code, Bitbucket, etc.   
Following are some features of GitLab:

* Open-source community edition repository management platform.
* Easy Maintaining of a repository on a server.
* Offers tools like Group Milestones, Time Tracking and Issue Tracker, etc. for effective development.
* More Spontaneous User interface and authentication features.
* User Permission and Branch protection are enhanced.

**GitHub:** GitHub is a repository hosting service tool that features collaboration and access control. It is a platform for programmers to fix bugs together and host open-source projects. GitHub is designed for the developers and to help them track their changes into a project through the repository.   
Following are some features of GitHub:

* Specifies milestones and labels to the projects.
* Comparison view between branches is allowed.
* GitHub Pages allows us to publish and host websites within GitHub.
* Syntax highlight feature.
* It allows third-party API integrations for bug tracking and cloud hosting.

| Parameters | GitLab | GitHub |
| --- | --- | --- |
| Developed by | GitLab was developed by Dmitriy Zaporozhets and Valery Sizov. | GitHub was developed by Chris Wanstrath, Tom Preston-Werner, P. J. Hyett, and Scott Chacon. |
| Open-sourced | GitLab is open-source for community edition. | GitHub is not open source. |
| Public Repository | It allows users to make public repository. | It allows users to have unlimited free repository. |
| Private Repository | GitLab also provides free private repository. | GitHub allows users to have free private repository but with a maximum of three collaborators. |
| Navigation | GitLab provides the feature of navigation into the repository. | GitHub allows users to navigate usability. |
| Project Analysis | GitLab provides user to see project development charts. | GitHub doesn’t have this feature yet but they can check the commit history. |
| Advantages | * GitLab is freely available and open is source for community edition * It is a cloud-native application and is highly secure. | * It helps us create an organized document for the project. * It is used for sharing the work in front of the public. |
| Disadvantages | * GitLab is available with many bugs and it makes the user experience sloppy. * It is difficult to manage code reviews for first-timers. | * There is a limited private repository. * It supports only Git version control. |
| Company | It is owned by GitLab Inc. | It is owned by Microsoft Corporation. |
| Security | More secure than Github. | It is less secure as security Dashboard, License Compliance is missing in GitHub. |
| Attachments | Gitlab supports adding other types of attachments. | GitHub does not allow adding other types of attachments. |

# Difference Between GIT and SVN

**GIT:** Git is an open-source distributed version control system developed by Linus Torvalds in 2005. Its emphasis on speed and data integrity in which there is no centralized connectivity is needed. It is powerful and cheap branching with easy merge in which each developer has his repository and have a local copy in which they can change history. It supports non-linear development branches and applications with a large number of codes files.

**Here are some .git directory structures used in GIT:**

* **HEAD/:** A pointer structure used in git.
* **Config/:** Contains all configuration preferences.
* **description/:** Description of your project.
* **index/:** It is used as a staging area between working directory.
* **object/:** All the data are stored here.
* **logs/:** Keeps record to change that are made.

**SVN:** Apache Subversion is an open-source software version and revision control system under the Apache license. It managed files and folders that are present in the repository. It can operate across the network, which allows it and used by people on different computer .we can say that a repository is like an ordinary file server which allows it to be used by people on a different computer.

Below is a table of differences between GIT and SVN:

| GIT | SVN |
| --- | --- |
| Git is open source distributed vice control system developed by Linus Torvalds in 2005. It emphasis on speed and data integrity | Apache Subversion is an open source software version and revision control system under Apache license. |
| Git has a Distributed Model. | SVN has a Centralized Model. |
| In git every user has their own copy of code on their local like their own branch. | In SVN there is central repository has working copy that also make changes and committed in central repository. |
| In git we do not required any Network to perform git operation. | In SVN we required Network for runs the SVN operation. |
| Git is more difficult to learn. It has more concepts and commands. | SVN is much easier to learn as compared to git. |
| Git deals with large number of files like binary files that change quickly that why it become slow. | SVN control the large number of binary files easily. |
| In git we create only .git directory. | In SVN we create .svn directory in each folder. |
| It does not have good UI as compared to SVN. | SVN has simple and better user interface . |
| Features of GIT:   * Distributed System. * Branching. * Compatibility. * Non-linear Development. * Lightweight. * Open source. | Features of SVN:   * Directories are versioned * Copying, deleting, and renaming. * Free-form versioned metadata . * Atomic commits. * Branching and tagging. * Merge tracking. * File locking. |