



Version Control with Git and GitHub

James Balamuta

Department of Statistics, Informatics
University of Illinois at Urbana-Champaign

June 15, 2017

CC BY-NC-SA 4.0, 2016 - 2017, James J Balamuta

On the Agenda

1 Intro

- Background
- Pros and Cons
- Why git?
- Terminology

2 GitHub

- Background
- Accounts

3 Git

- Setup
- Workflow
- Merge Conflict
- Authorization

4 RStudio with git

- Sample Workflow

5 Extra

What are Version Control Systems?

Version Control Systems (VCS) are applications that seeks to provide a way to *manage* and *track* the evolution of work done on a project.

How does it work?

Think of a VCS like a camera. Each time you take a picture, time is being “frozen” in place and stored in a photo album.



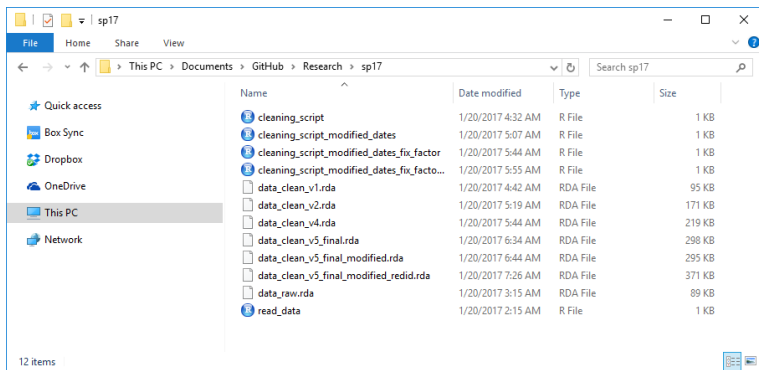
In essence, an *incomplete* history of your project will be etched into stone.

Why is the history incomplete?

*The pictures do not lie, but neither do they tell the whole story.
They are merely a record of time passing, the outward evidence.*
— Paul Auster, *Travels in the Scriptorium*

- Granularity of change
- Developers come and go; so does the context

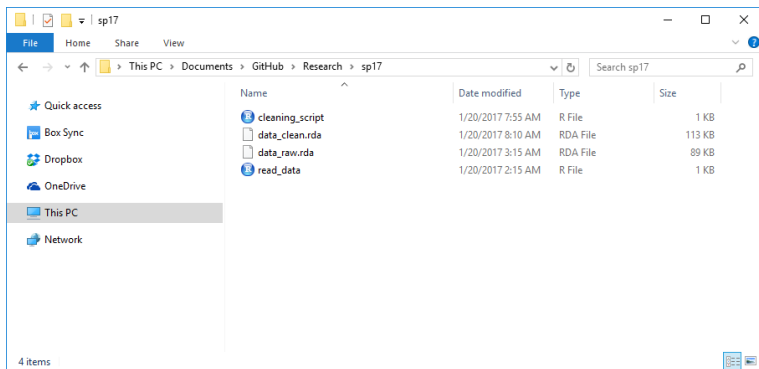
It's time to have 'the talk' about how we name files...



Inspired by PhD Comics: A Story told in File Names

Redux of File Naming

Under version control, the previous file structure would be changed to:



- All changes to the cleaning script and cleaning data set are stored within the VCS.
- **There is no need to label *version* information within the file**

Example of Versioning

Versioning by File Name

cleaning_script.R
data_clean_v1.rda



cleaning_script_modified_dates.R
data_clean_v2.rda



cleaning_script_modified_dates_fix_factor.R
data_clean_v4.rda



cleaning_script_modified_dates_fix_factor_redid.R
data_clean_v5.rda

Versioning by VCS

Files: cleaning_script.R, data_clean.rda
Commit: Added an initial cleaning script



Files: cleaning_script.R, data_clean.rda
Commit: Dates read as MM/DD/YYYY vs.
DD/MM/YYYY



Files: cleaning_script.R, data_clean.rda
Commit: Addressed incorrect factor labels



Files: cleaning_script.R, data_clean.rda
Commit: Applied a sweep operation to correctly
manipulate the files.

Pros v. Cons

Benefits of VCS

- Abridged history changes to your project
- Ability to transverse the space-time continuum
- Try out *new* ideas without worrying about breaking something!
- Collaborate with others *without* waiting for their component to be finished
- Continuous Integration (CI) allows for problems to be detected early

Downsides of VCS

- Not user-friendly
- Takes a bit of **time** and **practice** to understand.
- Windows support normally *lags* behind macOS and Linux

VCS Options

There are many VCS that are available:

- `git`: git
- `svn`: Subversion
- `cvs`: Concurrent Versions System

However, the general consensus is that `git` is the **best** because:

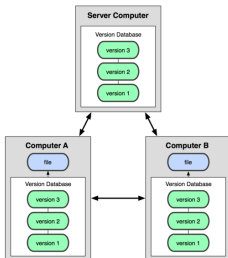
- It's free
- Multi-platform
- Rapidly growing ecosystem
- Distributed versioning model
- Large industry support

Therefore, the focus of the course will be in working with `git`.

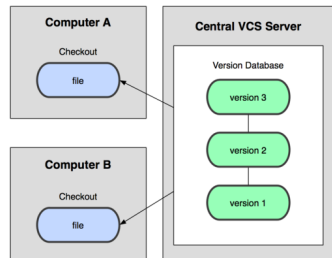
Distributed Model vs. Centralized Model

- git uses a **distributed model** that gives *everyone* a copy of the repository.
 - Making and distributing digital copies of the party's photos.
- Other VCS opt for a **centralized model** that requires everyone to be connected to a server to individually check out files.
 - Think of checking out the only book at a library.

Distributed Version Control



Centralized Version Control



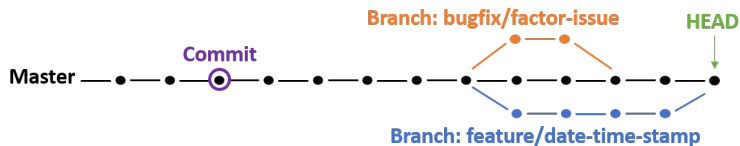
git Terminology

There are *many* terms associated with a `git` workflow.
We'll explore a few of those terms next.

General Terms

- **HEAD** refers to the point in time one is currently viewing.
- **Stage** is a holding area that indicates what files should be captured.
 - Place where pictures are being prepared to be taken.
- **Commits** are the saved records of the staged files.
 - Photos *taken* on the stage.
- **Repositories** (“repos”) hold all the **commits** to the project.
 - Photo album containing the photos.
- **Branches** experiments that do *not* influence the state of the **repo**
 - New ways to take pictures *without* affecting previously taken pictures.
- **Master** is the default **branch** that **commits** are saved to
 - General photo album of life.
- **Checkout** changes the **branch** being viewed or moves the **HEAD** back to a specific **commit**.
 - Open the photo album to a specific point in time or experiment.

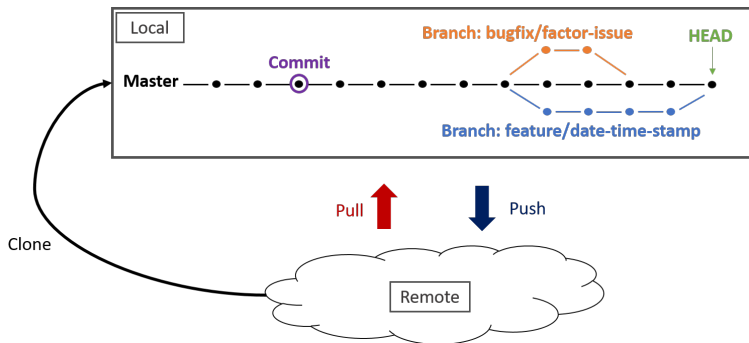
Repository Diagram



Sync Terms

- **Cloning** creates a copy of the **repo** onto ones computer.
 - Downloading the photo album in its entirety from the *cloud*.
- **Local** refers to the copy of the **repo** on your computer.
 - Photo album on your computer.
- **Remote/Upstream** location of where the **repo** was **cloned** from.
 - Where the photo album is stored in the *cloud*.
- **Origin** typically the name given to the **remote** location.
 - Nickname of the location of the photo album in the *cloud*.
- **Pull** retrieve commits from the **remote repo**.
 - Download *new* pictures from the *cloud* to your local album.
- **Push** send commits from the **local repo** to the **remote**.
 - Upload *new* pictures to the *cloud* photo album.

Repository Diagram with Remotes



On the Agenda

1 Intro

- Background
- Pros and Cons
- Why git?
- Terminology

2 GitHub

- Background
- Accounts

3 Git

- Setup
- Workflow
- Merge Conflict
- Authorization

4 RStudio with git

- Sample Workflow

5 Extra

What is GitHub?



GitHub

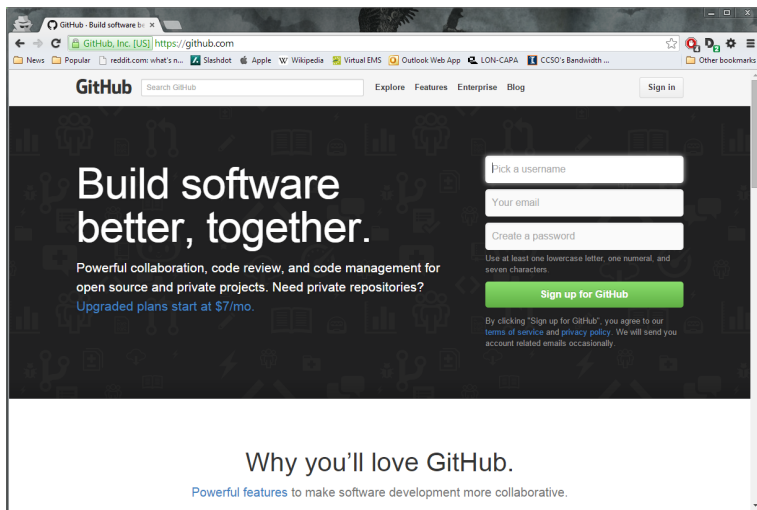
- Popular place to setup a remote repository
 - Over 14 million users and 35 million repos as of April 2016
- Provides a web interface for `git`
 - Search the space-time continuum of your project
- Promotes open source tenets
 - *View code instantly for any large open source project.*

Why use GitHub?

- Improves the user experience for using `git`
 - Visually explore code differences
- Enables collaboration with others via pull requests (PRs) and issue tickets
 - Changes the paradigm from *asking* for a fix to *providing* a fix
- Social media for the programmer
 - News feeds that track what your favorite programmer / projects are up to
- Accessible from all over the world
 - Except in China...

Create a GitHub Account

Register an account on [GitHub](https://github.com) using your @illinois.edu e-mail address.



The screenshot shows the GitHub homepage in a web browser. The browser's address bar displays "https://github.com". The page features the GitHub logo and a search bar. The main heading reads "Build software better, together." Below this, it states "Powerful collaboration, code review, and code management for open source and private projects. Need private repositories? Upgraded plans start at \$7/mo." On the right side, there is a sign-up form with three input fields: "Pick a username", "Your email", and "Create a password". Below these fields is a green button labeled "Sign up for GitHub". A note below the button states: "Use at least one lowercase letter, one numeral, and seven characters." At the bottom of the sign-up section, it says: "By clicking 'Sign up for GitHub', you agree to our [terms of service](#) and [privacy policy](#). We will send you account related emails occasionally." The footer of the page includes the text "Why you'll love GitHub." and "Powerful features to make software development more collaborative."

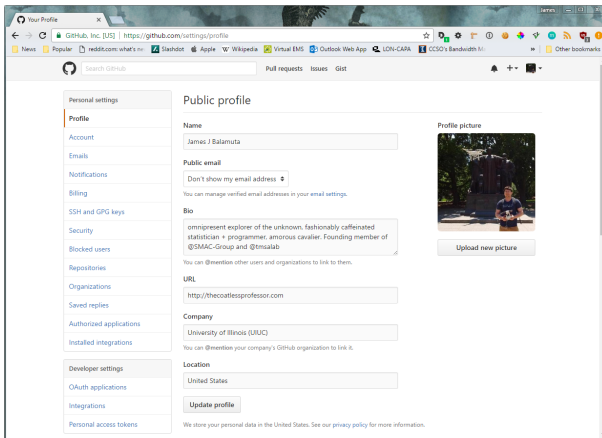
Request a Student Account

Place a request using [GitHub Education](#) to obtain **unlimited** free private repositories. Also, consider forming an organization, which is able to receive **unlimited** free private repositories with fine permission control.

The screenshot shows a web browser window with the URL https://education.github.com/discount_requests/new. The page is titled "Request a discount" and states "Discounted and free plans are available for educational use". The form is divided into two steps: "Step 1: Tell us what you need" and "Step 2: Tell us about you". Under Step 1, there is a question "Which best describes you?" with five radio button options: "Student", "Teacher", "Researcher", "Administrator/staff", and "Other". Below this is another question "What are you looking to get a discount for?" with two radio button options: "Individual account" and "Organization account". A green "Next" button is at the bottom of the form. The footer of the page includes copyright information "© 2014 GitHub, Inc.", links for "Terms", "Privacy", "Security", and "Contact", a GitHub logo, and social media links for "@GitHubEducation", "Status", "Blog", and "About".

Fill in your GitHub Profile

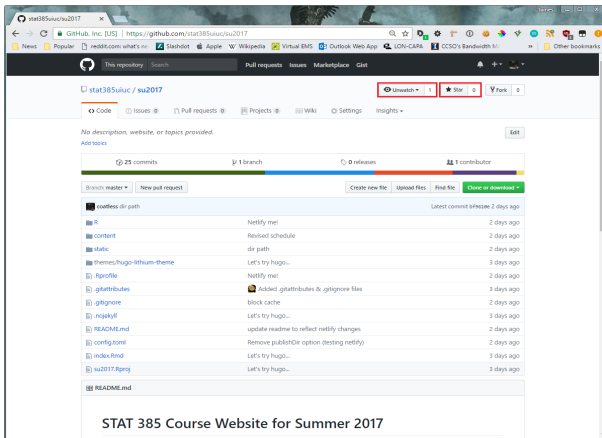
After you create an account, feel free to fill in your profile information by clicking on the **down arrow** in the upper right hand corner and selecting **Settings**



Star and Follow the Course Repository

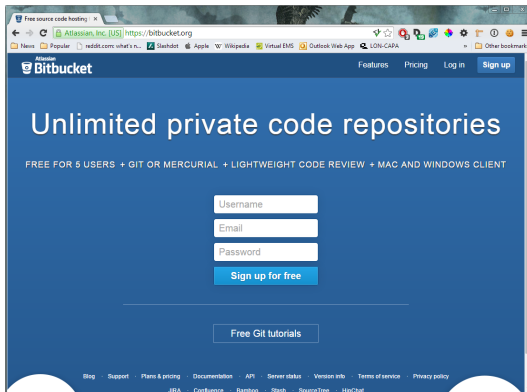
Receive updates when the course repository gets updated by **Watching** and **Starring** the course repository

<https://github.com/stat385uiuc/su2017>



BitBucket: An alternative to GitHub

Dislike GitHub? Consider using **BitBucket**. The repositories are generally more closed source than GitHub. However, for researchers and students, BitBucket provides **unlimited private repositories with unlimited contributors**.



On the Agenda

1 Intro

- Background
- Pros and Cons
- Why git?
- Terminology

2 GitHub

- Background
- Accounts

3 Git

- Setup
- Workflow
- Merge Conflict
- Authorization

4 RStudio with git

- Sample Workflow

5 Extra

Tutorial

Please follow the tutorial to setup git found here:

[http://thecoatlessprofessor.com/tutorials/
downloading-and-installing-git/](http://thecoatlessprofessor.com/tutorials/downloading-and-installing-git/)

git Commands

Unlike the shell commands, where each command was separated (e.g. head, tail), git has a series of verbs associated with a single command aptly named git.

To use git commands use the form of:

```
git <verb>
```

To obtain help regarding a specific command, use:

```
git <verb> --help  
man git-<verb>
```

General Workflow Overview

The general workflow with git is as follows

```
git init           # Create a repo or initialize an prior repo
git pull           # Retrieve latest commits from remote repo
git status         # Figure out the state of files
git add <file>     # Stage a specific file(s) to be committed
                  ## OR ##
git add .          # All files within the working directory
git commit         # Commit the staged file(s) into the repo
git status         # Verify the proper files have been committed
git push           # Sync to remote repository
```

Initializing a Repository

The initialization of a git repository requires the directory to be empty.

- Create an empty directory called `helloworld` and change into the directory.

```
mkdir helloworld && cd helloworld
```

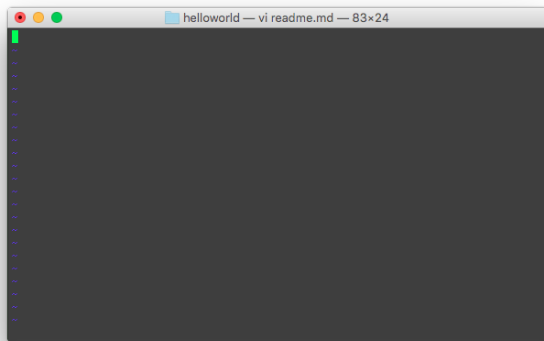
- Initialize a new git repository

```
git init
```

Adding Sample Content

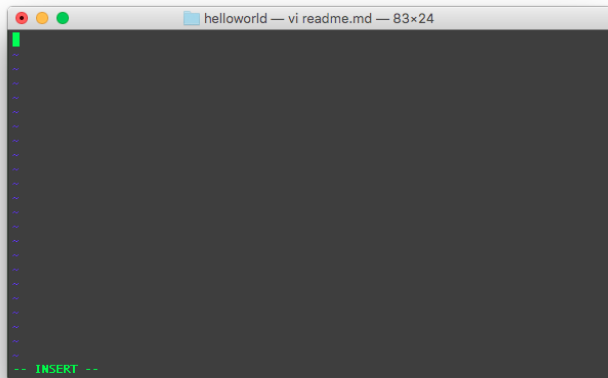
To add content, first create a file using `touch` and then open it with your favorite CLI editor.

```
touch readme.md  
vi readme.md
```



Enable Insert Mode

To be able to add text via a CLI, you must enter into **INSERT** mode. This is accomplished by pressing **i** or the **Insert** button.



Example of Content

If you wish, feel free to type in the following:

```
# Example git workflow
```

Within this repository we will be practicing how to commit with git.

Example of Content in CLI Editor

A screenshot of a terminal window titled "helloworld — vi readme.md — 83x24". The terminal has a dark gray background with green text. The first line is "# Example git workflow". The second line is "Within this repository we will be practicing how to commit with git.", followed by a cursor. Below this are several tilde (~) characters representing blank lines. At the bottom, there is a "-- INSERT --" prompt.

```
helloworld — vi readme.md — 83x24
# Example git workflow

Within this repository we will be practicing how to commit
with git.~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
-- INSERT --
```

Saving and Closing the CLI Editor

To close and save the file, press **ESC** and then type **:wq**.

In instances where you do *not* want to modify the file but close it, use `:q!`

A screenshot of a terminal window titled "helloworld — vi readme.md — 83x24". The terminal has a dark gray background with green text. The first line is "# Example git workflow". The second line is "Within this repository we will be practicing how to commit with git.". This is followed by thirteen tilde (~) characters, one per line. The last line shows the command ":wq" at the prompt.

```
helloworld — vi readme.md — 83x24
# Example git workflow
Within this repository we will be practicing how to commit
with git.
~
~
~
~
~
~
~
~
~
~
~
~
~
:wq
```

Checking the Status of git

Often it is helpful to know the present status of git. The status tells you:

- Untracked files
 - Yet to be staged
- Staged files
 - Need to be committed into the repo
- File conflicts from mergs
 - More on this later...

Check with:

```
git status
```

Checking the Status of git

```
git status
```

```
On branch master
```

```
Your branch and 'origin/master' have diverged,  
and have 1 and 1 different commits each, respectively.
```

```
(use "git pull" to merge the remote branch into yours)
```

```
You have unmerged paths.
```

```
(fix conflicts and run "git commit")
```

```
(use "git merge --abort" to abort the merge)
```

```
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")
```

Stage the File

To prepare the file for inclusion within our repository, we must first *stage* it. By staging the file, we are telling `git` that we intend for the file to be apart of the next `commit`.

```
git add readme.md
```

Note: There will be *no* output indicating whether the file is staged.

Checking the Status of git

Let's verify the file has been *staged* by running a status check:

```
git status
```

```
On branch master
```

```
Your branch and 'origin/master' have diverged,  
and have 1 and 1 different commits each, respectively.
```

```
(use "git pull" to merge the remote branch into yours)
```

```
All conflicts fixed but you are still merging.
```

```
(use "git commit" to conclude merge)
```

```
Changes to be committed:
```

```
    modified:   readme.md
```

Writing a Commit Message

With the file now staged, the file can be included within the repository by committing it. All commits must have a comment indicating what is being committed into the repository.

There are two forms of commits:

- Long form: `git commit` (spawns CLI)
- Short form: `git commit -m "Message here"`

Depending on the project, there may be a preference for one over the other.

Preferred Git Commit

The preferred git commit is done with:

Short description of 50 characters indicating a change

Long description of what changed and why the change was necessary. This description may be multilined.

Note: There are two returns between the short and long description.

The Reality

	COMMENT	DATE
○	CREATED MAIN LOOP & TIMING CONTROL	14 HOURS AGO
○	ENABLED CONFIG FILE PARSING	9 HOURS AGO
○	MISC BUGFIXES	5 HOURS AGO
○	CODE ADDITIONS/EDITS	4 HOURS AGO
○	MORE CODE	4 HOURS AGO
○	HERE HAVE CODE	4 HOURS AGO
○	AAAAAAA	3 HOURS AGO
○	ADKJTSJKDFJSDKLFJ	3 HOURS AGO
○	MY HANDS ARE TYPING WORDS	2 HOURS AGO
○	HAAAAAAAAAANDS	2 HOURS AGO

AS A PROJECT DRAGS ON, MY GIT COMMIT
MESSAGES GET LESS AND LESS INFORMATIVE.

Source [XKCD: Git Commit](#)

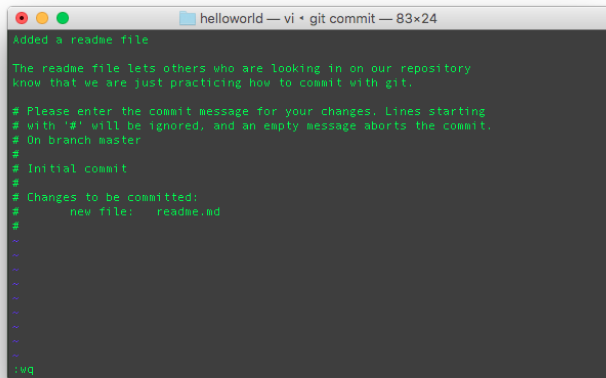
The Long Form git commit

View when entering into a proper long form `git` commit.

[illegible]

The Long Form `git commit`

- 1 Enable **INSERT** mode
- 2 Write the long form `git commit`
- 3 Exit from the CLI text editor using `:wq`



```
helloworld — vi • git commit — 83x24
Added a readme file

The readme file lets others who are looking in on our repository
know that we are just practicing how to commit with git.

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   new file:   readme.md
#
~
~
~
~
~
~
~
~
~
~
:wq
```

Did you forget to introduce yourself?

If during your first commit, you received:

```
*** Please tell me who you are.
```

Run

```
git config --global user.email "you@example.com"  
git config --global user.name "Your Name"
```

to set your account's default identity.

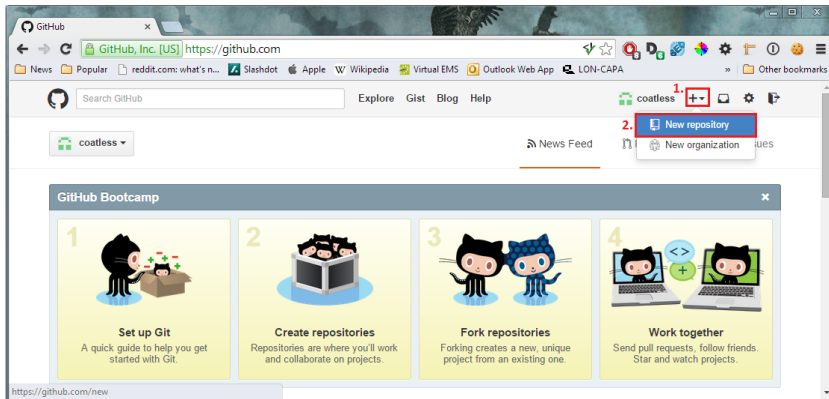
Omit `--global` to set the identity only in this repository.

You will need to introduce yourself to git! Otherwise, it won't be a very productive partnership.

Result of the Commit

```
[master (root-commit) 780ef6c] Added Added a readme file  
1 files changed, 4 insertions(+)  
create mode 100644 readme.md
```

Create a GitHub Repository



Create a GitHub Repository

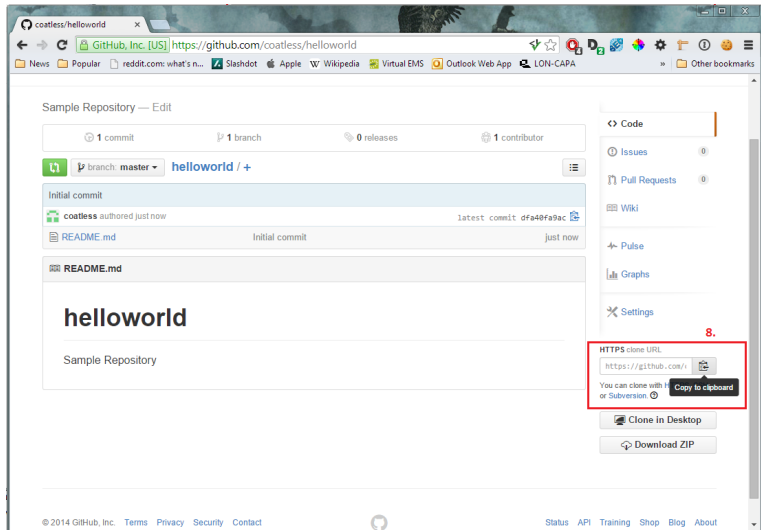
When establishing the repository, consider adding a license to your code.

The screenshot shows the GitHub 'Create a New Repository' page. The browser address bar displays 'https://github.com/new'. The page has a search bar and navigation links (Explore, Gist, Blog, Help). The main form includes:

- Owner:** A dropdown menu showing 'coatless'.
- Repository name:** A text input field containing 'helloworld' with a green checkmark icon to its right. This field is highlighted with a red box and labeled '3.'.
- Description (optional):** A text input field containing 'Sample Repository'. This field is highlighted with a red box and labeled '4.'.
- Visibility:** Two radio buttons: 'Public' (selected) and 'Private'.
- Initialize this repository with a README:** A checked checkbox. Below it, a note says: 'This will allow you to `git clone` the repository immediately. Skip this step if you have already run `git init` locally.'
- License:** A dropdown menu showing 'None'. This dropdown is highlighted with a red box and labeled '6.'.
- Create repository:** A green button at the bottom of the form. This button is highlighted with a red box and labeled '7.'.

At the bottom of the page, there is a footer with copyright information, links (Terms, Privacy, Security, Contact), and a status bar with links (Status, API, Training, Shop, Blog, About).

Obtain GitHub Repository Link



Add the Remote

To let your local git repository know about the new remote we must use:

```
git remote add origin <github_url>
```

In my case that would be:

```
git remote add origin git@github.com:coatless/helloworld.git
```

Verify the Remote

```
git remote -v
```

```
origin  git@github.com:coatless/helloworld.git (fetch)
```

```
origin  git@github.com:coatless/helloworld.git (push)
```

Push the changes

To send your changes to the remote simply:

```
git push
```

Though, there are two key issues that may arise:

- 1 The dreaded merge conflicts.
- 2 You must be authenticated on GitHub with an ssh key.

Merge Conflicts

To address merge conflicts take the following steps:

- 1 Figure out the files that are in conflict by using `git status`
- 2 Open each conflicted file.
- 3 Decide which change set to keep or remove.

```
<<<<<<< HEAD
```

```
Change in the local file
```

```
=====
```

```
Change that someone made in the remote file
```

```
>>>>>>> 5e2443102fbf316b88dc3cd3922bc680668de655
```

- <<<<<<< - Beginning of merge conflict where *YOUR* changes reside
- ===== - The end of your changes and the beginning of the remotes
- >>>>>>> - End of the remotes changes for that merge section conflict.

Merge Conflicts

To retain my commit, I delete all other changes.

Before:

```
<<<<<< HEAD
Change in the local file
=====
Change that someone made in the remote file
>>>>>> 5e2443102fbf316b88dc3cd3922bc680668de655
```

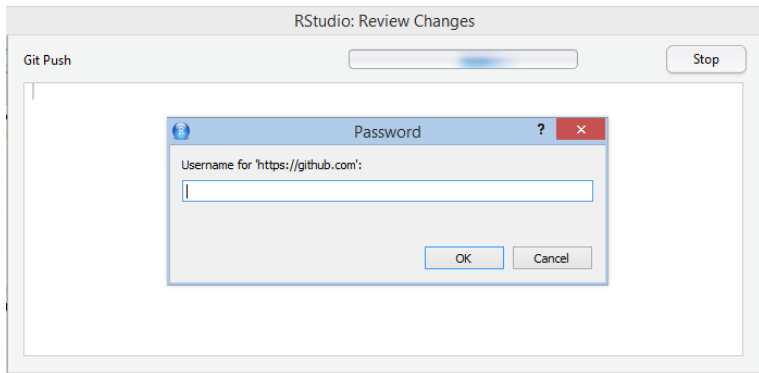
After:

```
Change in the local file
```

Then git add, git commit, and git push my changes.

Authorizing...

One of the downsides of pushing to GitHub is the need to be authenticated...



Good Authorization via Public and Private SSH Keys

To simplify this process, we opt to authorize ourselves to GitHub using an encryption technique that uses a **public and private key scheme**.

In this case, the public key is a string of symbols that is out there for the world to see.

On the other hand, the private key is considered to be confidential and only viewable by its owner.

This scheme enables messages signed with the private key to only be viewable via its public key (and vice versa).

Good Authorizing - SSH Key Generation

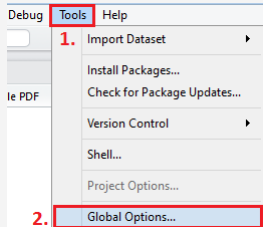
Instructions for generating an SSH key are given in two flavors:

- 1 Using RStudio's GUI
- 2 Using Shell/Terminal

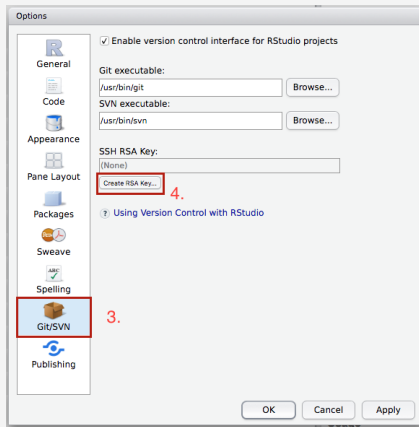
You only need to pick **one** route.

Good Authorizing - SSH Key Generation via RStudio

Click 'Tools'
Select 'Global Options'

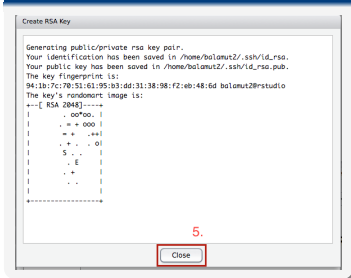


Click 'Git/SVN'
Click the 'Create RSA Key...'

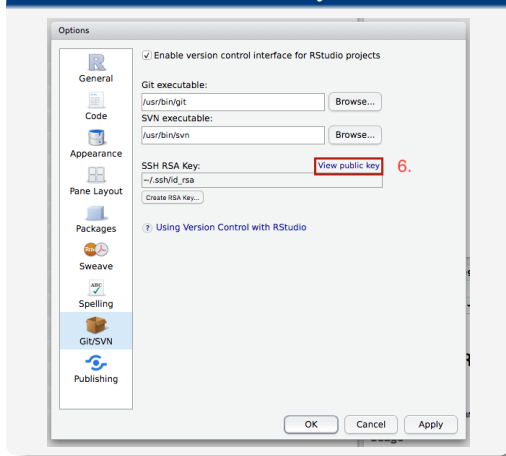


Good Authorizing - SSH Key Generation via RStudio

Click 'Close'



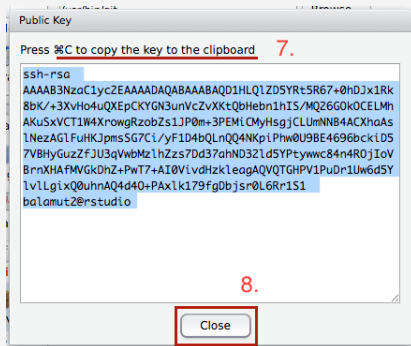
Click 'View Public Key'



Good Authorizing - SSH Key Generation via RStudio

Copy the public key with either

macOS: Command + C or Windows: Cntrl + C

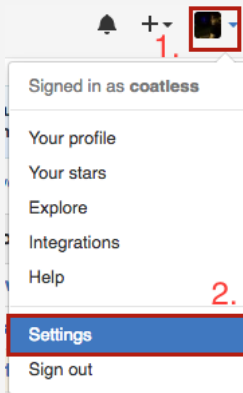


With the key being generated and on our clipboard, we can now add it to our GitHub account. . .

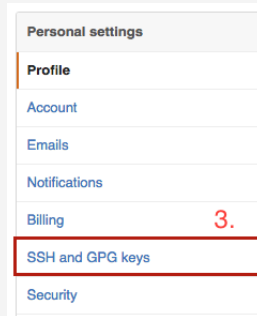
Good Authorizing - Add Public SSH Key to GitHub

Click on the picture in the upper right corner

Select 'Settings'



Click 'SSH and GPG Keys'



Good Authorizing - Add Public SSH Key to GitHub

Press the 'New SSH Key' button

SSH keys

4.

New SSH key

This is a list of SSH keys associated with your account. Remove any keys that you do not recognize.

Fill in the 'Title' input and paste the SSH into 'Key' with either macOS: Command + V or Windows: Cntrl + V

Title

rstudio.stat.illinois.edu

5.

Key

```
ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAQAD1HLQIZD5YRt5R67+0hDjx1Rk8bK/+3XvHo4uQXEpCKYGN3unVcZvXKt
QbHebn1hIS/MQ26GOkOCELMhAKuSxVCT1W4XrowgRzobZs1JP0m+3PEMiCMYHsgjCLUmNNB4ACXhaAslNezAGI
FuHKJpmsSG7CiyF1D4bQLnQQ4NKpiPhw0U9BE4696bckiD57VBHyGuzZfJU3qVwbMzlhZzs7Dd37ahND32ld5YPtyw
wc84n4ROjloVBmXHAfMVGkDhZ+PwT7+Al0VivdHzkleagAQVQTGHPV1PuDr1Uw6d5YlviLgixQ0uhnAQ4d4O+PAxlk1
79fgDbsr0L6Rr1S1 balamut2@rstudio
```

6.

Add SSH key

7.

Good Authorizing - Add Public SSH Key to GitHub

Check that the key has been added



rstudio.stat.illinois.edu

Fingerprint: 94:1b:7c:70:51:61:95:b3:dd:31:38:98:f2:eb:48:6d

Added on Jun 15, 2016 — Never used

SSH

Delete

If you have made it to this step, well done!

You can now officially commit via SSH.

Good Authorizing - SSH Key Generation

As promised, next up are the instructions for SSH key generation within Terminal/Shell...

Do **NOT** repeat this if you opted for to generate the SSH key using the RStudio approach.

Good Authorizing - SSH Authorization via Terminal

Use SSH Authorization by opening terminal/shell

(in RStudio access it via Tools → Shell)

```
ssh-keygen -t rsa -b 4096 -C "your_email@example.com" # Creates new SSH Key

# After the key is generated, you will be prompted with
Enter a file in which to save the key (/Users/you/.ssh/id_rsa): [Press enter]

# Create a password for the key (needs to be rememberable)
Enter passphrase (empty for no passphrase): [Type a passphrase]
Enter same passphrase again: [Type passphrase again]

# Start the SSH Agent
eval "$(ssh-agent -s)"

# Add the key
ssh-add ~/.ssh/id_rsa
```


Copy key to clipboard

Windows:

```
clip < ~/.ssh/id_rsa.pub
```

macOS:

```
pbcopy < ~/.ssh/id_rsa.pub
```

Then we add the key to GitHub using the previous steps...

Bad Authorizing - Revealing Login Information

Sometimes, `ssh` may be limited due to the internet options you have at your disposal. At times like this, there is one last option.

Authenticate over HTTPS.

The approach detailed next is suboptimal as you end up having to *store* your login credentials alongside the repository URL.

Bad Authorization - Use this approach when everything fails

This is a very bad option since R Studio does NOT hash .proj files.

When you enter in your GitHub repository to create your R Project, append your username and password like so to the URL:

```
https://username:password@github.com/username/helloworld.git
```

On the Agenda

1 Intro

- Background
- Pros and Cons
- Why git?
- Terminology

2 GitHub

- Background
- Accounts

3 Git

- Setup
- Workflow
- Merge Conflict
- Authorization

4 RStudio with git

- Sample Workflow

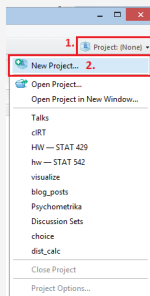
5 Extra

RStudio git Client

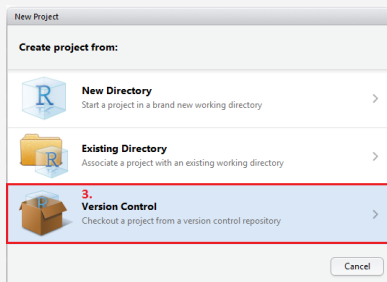
Previously, details were given as to how to work *solely* with a CLI git client. However, RStudio does have a built in git client that is not necessarily ideal. But, definitely worth mentioning in the scope of this git tutorial. The reason is because there are a few limitations to the client and it also involves using a GUI.

Create a new RStudio Project

Open Project Menu Select 'New Project'




Select Version Control



Create a new R Studio Project

New Project

[Back](#) **Clone Git Repository**



4. Repository URL:

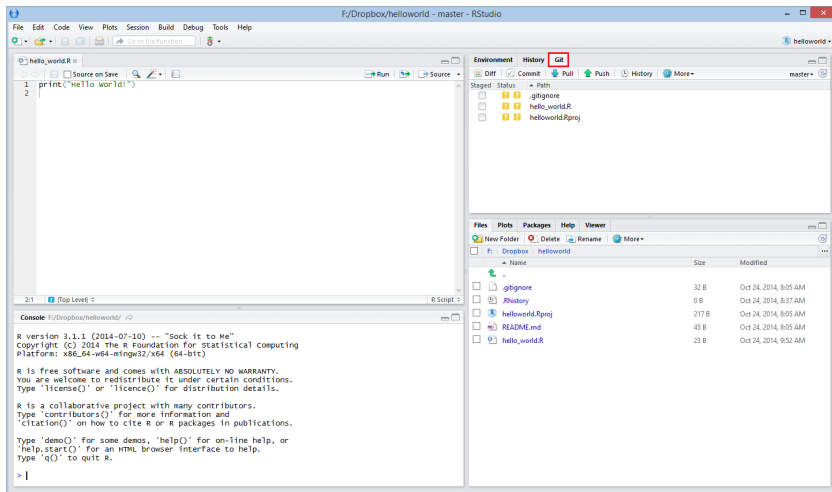
5. Project directory name:

Create project as subdirectory of: 6. [Browse...](#)

☐ Open in new window

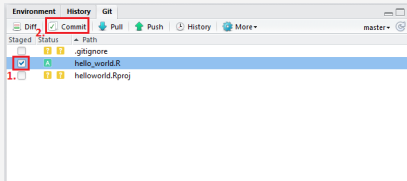
7. [Create Project](#) [Cancel](#)

RStudio - Project View

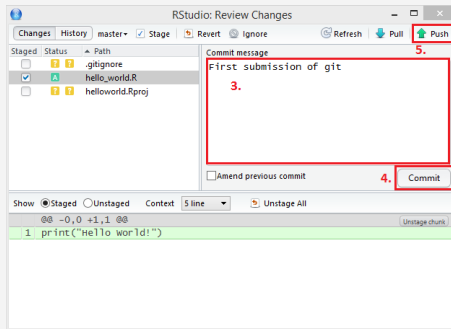


Using git via R Studio

Select file to be staged
Press 'Commit'



Enter commit message
Press 'Commit'
Press 'Push'



On the Agenda

1 Intro

- Background
- Pros and Cons
- Why git?
- Terminology

2 GitHub

- Background
- Accounts

3 Git

- Setup
- Workflow
- Merge Conflict
- Authorization

4 RStudio with git

- Sample Workflow

5 Extra

Git Commands

Basic Git Commands:

`git init`: Initializes a new git repository. (Must be run to setup repo, no repo = no commands working)

`git config <option>`: Configure git options

`git help <command>`: Provides information on how to use and configure a specific git command.

`git status`: See what files are in the repository, what changes need to be committed, and what branch of the repository is active.

Git Commands

Feature Development Commands:

`git add <file>`: Add a new or changed file or files (.) into a "staging" area.

`git commit -m "Message here"` : Pushes change into the repository with message

`git checkout`: A way to select which line of development you're working on. (e.g. Master or your own branch)

`git branch <name>`: Build a new branch off of the active repository to make changes and file additions that are completely your own.

`git merge`: Merge changes in your branch back to the master branch.

Git Commands

Syncing Git Commands:

- `git remote`: Create, view, and delete connections to other repositories.
- `git fetch`: Imports commits from a remote repository into your local repository. Helpful for reviewing changes before integrating them into the master branch.
- `git push`: Push the local changes to the repository up to the version control server.
- `git pull`: Pull the newest changes from the version control server to your local environment.

Sample git workflow

```
# Make sure repo is up to date
git checkout master
git fetch origin master
git pull --rebase origin

# Start a new feature
git checkout master
git branch AdvOfJames
git checkout AdvOfJames
# or: git checkout -b AdvOfJames master

# Add a new file
git add GiantPeach.r
git commit -m "New feature started"

# Update a file with changes
git add GiantPeach.r
git commit -m "New feature finished"

# Merge in the AdvOfJames branch
git checkout master      # Switch to master
git merge AdvOfJames     # Merge in change
git branch -d AdvOfJames # Remove development branch

# Push update
git push origin master
```