SDSU Git Basics

SDSU Rocket Project

December 12, 2018

Contents

1 Git Basics

1.1 Git Overview

Git is a Version Control System (VCS) designed to keep track of changes to files across multiple authors and machines. GitHub is a free public service that hosts the server-side copy (the “remote”) of all our projects (“repositories” or “repos”).
Each user can copy (“clone”) the remote to their machine to have their own copy (their “local”) that they can edit however they like.
Each repo has a main working tree (usually the “master” branch) and can have an infinite number of other “branches” that can be edited before their changes are “merged” back into master. Git will automatically check that a merge will not cause conflicts (such as multiple versions of the same file in the same place) and even offers tools to resolve such conflicts.

1.2 Basic Steps of Git

1. A branch is created
   When created, a branch is identical to its base, which is usually Master.
2. Files are created, changed, or deleted
3. New changes (called “unstaged” can be made “staged” to mark them as part of the next commit
4. Those changes are stored as commits
   Each commit has an associated hash, generated from a SHA1 checksum of the contents of the commit.
   Commits are referred to using the first 7 characters of the hash.
   Each commit also has at least one line of text, to let you describe what was changed in that commit.
5. Commits are then uploaded to the remote from your local
   This process is called “pushing” your commit.
   Pushing will upload any commits since your last push, effectively updating the remote to your local.
6. Someone else can then download your commits to their local
   This process is called “pulling” a commit
   Pulling will download any commits since the last pull, effectively updating your local to the remote.
7. When your changes are done, and are ready, you can “merge” your branch into Master via a Pull Request

1.3 Basic Git Example

A basic example of the steps to create a branch, write a readme file, and push it to the branch. These commands are explained in more detail on the next page.

`git branch mybranch`	- Create a new branch named “mybranch”
`git checkout mybranch`	- Switch the working directory to the new branch
`vim README.rmd`	- Create and edit `README.rmd` (vim is a text editor)
`git add README.rmd`	- Stage `README.rmd` for commit
`git commit -m ‘Add README.rmd’`	- Commit the change with the message ‘Add README.rmd’
`git pull`	- Fetch any commits, to prevent conflicts when you push
`git push`	- Push your commit to the remote

While there are commands to create and merge Pull Requests, it is much easier and preferred that it is done via the GitHub website.

1.4 Basic Commands

`git pull`
: "Pull" retrieves changes from the remote and applies them to the local
`git pull`

`git push`
: "Push" sends changes on the local to the remote
<u>ALWAYS PULL BEFORE PUSHING TO AVOID CONFLICTS</u>
`git push`

`git add`
: Stage changes for a commit.
`git add <filename>`

`git commit`
: Save staged changes into commit
`git commit [-m <commit message>]`

`git branch`
: List, create, and delete branches.
`git branch -a` will list all branches (and show an asterisk next to the current branch)
`git branch <name>` will create a new branch with the specified `name`
`git branch -D <name>` will delete the branch with the specified `name`

`git checkout`
: Switch to a branch.
`git checkout <branch name>`

`git cherry-pick`
: Apply changes from an existing commit.
`git cherry-pick <commit hash>`

`git clone`
: Clone a remote repository to the local machine.
`git clone <url> [target directory]`

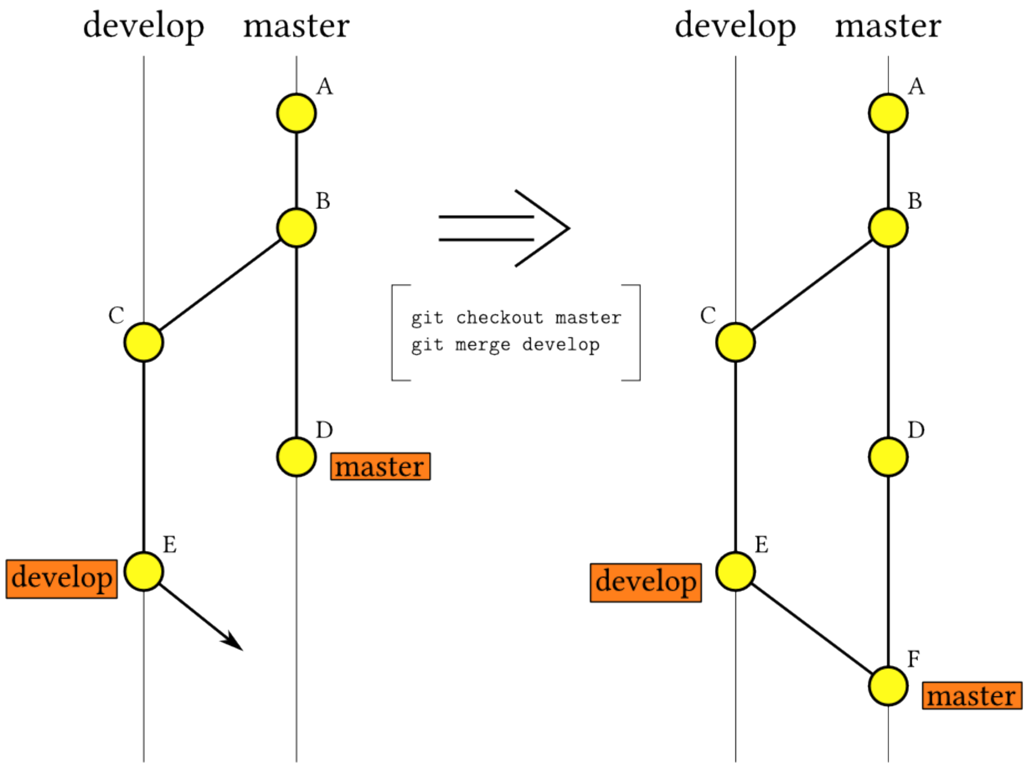
`git diff`
: Show the differences between two commits or a commit and the working tree
`git diff <commit> [commit]`
If the second `commit` is not specified, it will compare against the most recent

`git status`
: Show working tree status (staged and unstaged changes, etc...).
`git status`

`git log`
: Show a log of the working tree. Zack prefers it with these settings:
`git log --oneline --graph`

`git reset`
: Resets the working tree to the specified state. Two most useful ways:
`git reset --soft [commit]` will reset you to `commit` and leave any changes ready to be staged
`git reset --hard [commit]` will discard any changes since `commit`
If no `commit` is specified, the most recent will be assumed

1.5 Graphical representation



Events in the image above:

1. Commit A is created and pushed
2. The branch “develop” is created off Master at Point B
3. Commit C is created and pushed on develop
4. Commit D is created and pushed on Master
   Note that a commit on one branch does not affect any other branch
5. Commit E is created and pushed on develop
6. Develop is merged into Master at point F

As of point F, all commits on develop are on Master. Master has gained the new features, bugfixes, etc, that were created in develop without having to experience any bugs or errors during the development of commits C and F. This way, only functioning code is on Master.

2 Contribution

2.1 Contribution Procedure

1. Find an issue on the GitHub page, find something to add, etc.
2. Create a new branch from Master to address it
   Your branch should be named appropriately.
   For example, if you are adding documentation for an imaginary BHQQ system, your branch would be named `bhqq-documentation`
3. Commit your changes as you work
   You should commit changes as you make them, not everything in one large block. This makes it easier to reference specific changes, especially during review.
   Commit messages should be short and descriptive.
4. File a Pull Request
   The Pull Request title should only be a few words, *e.g.* `Add BHQQ Documentation`.
   In the Pull Request message (or in the comments), reference the issue (if any) that you are addressing.
   GitHub has an automatic system, where you only need to type the issue number (*e.g.* `#10`)
5. Assign reviewers
   A pull request into Master can only be accepted if the changes have been approved by at least one other person who has write access to Master.
   You should assign someone who is related to the content of your Pull Request (*e.g.* the person who wrote the BHQQ system that you are writing documentation for)
   More than one person can be assigned to review your Pull Request.
6. Make changes
   The reviewer may request changes to be made before they approve your Pull Request.
   Your Pull Request will also need to be re-approved if you push more commits to your branch before it is merged.
7. Pull Request Merged
   Once all reviewers have approved your Pull Request, it can be merged into Master.

2.2 Contribution Guidelines

- Create a separate branch for each major change/feature.
  This allows us to accept/reject on a feature-by-feature basis.
- The name of your PR should be short and descriptive.
  If you cannot fit everything into a few words, you can expand on additional lines or by further explaining in the PR comments.
- Your PR **must** be reviewed by at least one other person.
  This person should be someone related to the PR
  (*e.g.* whoever wrote whatever it is that you are modifying).
- Commit names should be short, descriptive, and unique.
  Pushing six commits in a row all named `code formatting` helps absolutely no one in understand what is being changed and where.
  Additionally, a commit name such as `Changed README.md` is equally unhelpful. Be sure to include what changes were made. If too many changes were made, include the gist of it (e.g. "Restructure Section 2") and include a more detailed description in the commit description.

2.3 How to Improve the Project

In order to continually improve the project, it is important to take full advantage of the features of GitHub. This means using features such as Issues, Labels, and Milestones. These features are completely contained inside the GitHub web interface (or in their graphical client, if you choose to use it).

2.3.1 Issues

Anyone can file an issue in the repo. An issue doesn't necessarily need to be an "issue," they are just used to keep track of any bugs, feature requests, suggested changes, and things of that sort that need to be addressed.

All issues, like all Pull Requests, have as associated number and name. For example, the most recent issue filed is titled `#9: Cross Platform Compatibility`. This issue can be referenced anywhere in the repo with just "#9" and GitHub will automatically create a hyperlink to that issue.

Issues have three other important fields besides the name and number:

Assignees
: You can assign specific people to work on any issue. It is best to assign the person most likely to resolve the issue. For example, an issue regarding a bug with the way GUI elements are being drawn on the client should be assigned to the person who wrote that section of the client code.

Labels - Described in section 2.3.2

Milestones
: You can associate an issue or Pull Request with a specific Milestone, such as "Convert from Sockets to MQTT" that would span many issues and Pull Requests, as a way to group them together.

It is important that there are always plenty of issues to be resolved. If you notice a bug, want some new feature, or have an idea for how something could be changed or improved, this is how you let other people know. This is we make progress – by letting people know what progress needs to be made.

2.3.2 Labels

Labels are markers that can be applied to an issue or a Pull Request in order to help categorize, sort, and filter them. Multiple labels can be assigned to any issue or Pull Request. Labels are useful because they allow you to quickly get an idea of what an issue or Pull Request is, and they allow you to filter the list of issues and Pull Requests to only show those with certain labels.

New labels can be created or removed at any time. Currently, these exist:

- Suggestion
- Bug
- Documentation
- Duplicate
- Feature Request
- Help Wanted
- Question

If you ever feel that there is no label suitable for your issue or Pull Request, you can always create a new one to fill the hole.

3 Documentation

3.1 Guidelines

- **EVERYTHING** must be documented
  - All new features
  - Any new dependencies
  - *Any* changes that would make current documentation incorrect
- General information (Dependencies, basic operation and descriptions) belong in README.rmd
- Specialized information (if any) about a file belong in <filename>.rmd
- All code must be thoroughly commented
  You will probably not be here in a few years, and it would be best if the next generation can understand how everything works.

3.2 Markdown and RMarkdown

Markdown (*.md) and RMarkdown (*.rmd) are simple markup languages designed to be compiled into HTML documents, and any Markdown/RMarkdown files inside the repository or in comments will be parsed and displayed as HTML. RMarkdown, unlike vanilla Markdown, is capable of much more, including arbitrary LaTeX commands, as well as running basic R and Python code *inside* the text document.

RMarkdown Cheatsheet
More comprehensive cheatsheets for RMarkdown and Vanilla Markdown available at these hyperlinks

Some basics of RMarkdown:

- **Headings** are done with pound signs:
  ```
  #Heading 1
  ##Heading 2
  ###Heading 3
  ```
  And so on, to Heading 6
- **Hyperlinks** are done with the format `[link](url)`
- **Unordered Lists** are done with asterisks, dashes, or pluses, interchangeably:
  ```
  - Bullet 1
  + Bullet 2
      - Sublists are made with indentation with a tab or multiples of 2 spaces
  * Bullet 3
  ```
- **Ordered Lists** are done same as unordered lists, just with numbers:
  ```
  1. Entry One
  5. Entry Two     < The number doesn’t matter, just that it’s a number
      1. You can also have ordered sublists, with their own numbering
  ```
- **Inline Code** has graves around it: `` `print(‘hello’)` ``
  ~~~~
  ```
  Block code is done with three graves above and below
  ```
  ~~~~
- **Emphasis** is done with *single asterisks* or with _single underlines_
- **Bold** is done with **double asterisks** or with __double underlines__