All in one page

Thursday, December 2, 2021 4:10 AM

Git Cheat Sheet

Git: configurations

```
$ git config --global user.name "FirstName LastName"
$ git config --global user.email "your-email@email-provider.com"
$ git config --global color.ui true
$ git config --list
```

Git: starting a repository

```
$ git init
$ git status
```

Git: staging files

```
$ git add <file-name>
$ git add <file-name> <another-file-name> <yet-another-file-name>
$ git add .
$ git add --all
$ git add -A
$ git rm --cached <file-name>
$ git reset <file-name>
```

Git: committing to a repository

```
$ git commit -m "Add three files"
$ git reset --soft HEAD^
$ git commit --amend -m <enter your message>
```

Git: pulling and pushing from and to repositories

```
$ git remote add origin <link>
$ git push -u origin master
$ git clone <clone>
$ git pull
```

Git: branching

```
$ git branch
$ git branch <branch-name>
$ git checkout <branch-name>
$ git merge <branch-name>
$ git checkout -b <branch-name>
```

What's Git?

Git is a distributed version control system (DVCS). "Distributed" means that all developers within a team have a complete version of the project. A version control system is simply software that lets you effectively manage application versions. Thanks to Git, you'll be able to do the following:

- Keep track of all files in a project
- Record any changes to project files
- Restore previous versions of files
- Compare and analyze code
- Merge code from different computers and different team members.
- **Configuring Git**

Git Cheat Sheet

Git: configurations

```
$ git config --global user.name "FirstName LastName"
$ git config --global user.email "your-email@email-provider.com"
$ git config --global color.ui true
$ git config --list
```

When you come to a bank for the first time and ask to store your money there, they give you a bunch of paperwork to fill out. Before you can use banking services, you have to register with the bank. Git wants you to do the same (register with Git) before you start using a repository.

To tell Git who you are, run the following two commands:

```
$ git config --global user.name "King Kong"
$ git config --global user.email "king-kong@gmail.com"
$ git config --list
```

Output:

```
user.name=King Kong
user.email=king-kong@gmail.com
```

Tracking Files with Git

```
$ git status
On branch master
Initial commit
nothing to commit (create/copy files and use "git add"to track)
```

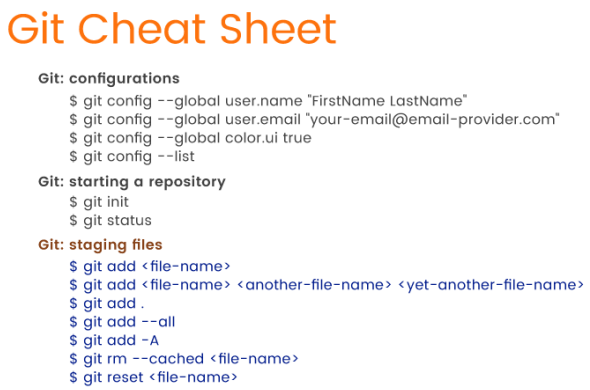
Since there are no files in the root directory yet, Git shows that there's nothing to commit.

When you run "git status" once more (assuming you've added a file to the project's root directory), you'll get a different output:

```
$ git status
On branch master
Initial commit
Untracked files:
(use "git add <file>..."to include inwhat will be committed)
my_new_file.txt
nothing added to commit but untracked files present (use "git add"to track)
```

Note the "Untracked files" message with the file "my new file.txt". Git conveniently informs us that we've added a new file to the project. But that isn't enough for Git. As Git tells us, we need to track "my new file.txt". In other words, we need to add "my new file.txt" to the staging area. The following section will uncover the basic Git commands for working with the staging area.

Staging Files with Git



Before we cover simple Git commands used for staging files, we need to explain what the staging area is.

Let's say you want to move some of your valuable effects to a lock box, but you don't know yet what things you'll put there. For now, you just gather things into a basket. You can take things out of the basket if you decide that they aren't valuable enough to store in a lock box, and you can add things to the basket as you wish. With Git, this basket is the *staging area*. When you move files to the staging area in Git, you actually *gather and prepare* files for Git before committing them to the local repository.

To let Git track files for a commit, we need to run the following in the terminal:

$ git add my_new_file.txt

That's it; you've added a file to the staging area with the "add" command. Don't forget to pass a filename to this command so Git knows which file to track.

But what has this "add" command actually done? Let's view an updated status (we promised that you'll often run "git status", didn't we?):

```
$ git status
On branch master
Initial commit
Changes to be committed:
(use "git rm --cached <file>..."to unstage)
new file: my_new_file.txt
```

The status has changed! Git knows that there's a newly created file in your basket (the staging area), and is ready to commit the file. We'll get to committing files in the next section. For now, we want to talk more about the "git add" command.

What if you create or change several files? With a basket as your staging area, you have to put things into the basket one by one. Committing files to the repository individually isn't convenient. What Git can do is provide alternatives to the "git add <file-name>" command.

Let's assume you've added another three files to the root directory: my-file.ts, another-file.js, and new file.rb. Now you want to add all of them to the staging area. Instead of adding these files separately, we can add them all together:

$ git add my-file.ts another-file.js new_file.rb

All you need to do is type file names separated by spaces following the "add" command. When you run "git status" once more to see what has changed, Git will output a new message listing all the files you've added:

```
$ git status
Changes to be committed:
(use "git rm --cached <file>..."to unstage)
new file: another_file.js
new file: my_file.ts
new file: my_new_file.txt
new file: new_file.rb
```

Adding several files to the staging area in one go is much more convenient! But hold on a second. What if the project grows enormously and you have to add more than three files? How can we add a dozen files (or dozens of files) in one go? Git accepts the challenge and offers the following solution:

$ git add .

Instead of listing file names one by one, you can use a period – yes, a simple dot – to select all files under the current directory. Git will grab all new or changed files and shove them into the basket (the staging area) all at once. That's even more convenient, isn't it? But Git can do even better.

There's a problem with the "git add ." command. Since we're currently working in the root directory, "git add ." will only add files located in the root directory. But the root directory may contain many *other directories* with files. How can we add files from those other directories plus the files in the root directory to the staging area? Git offers the command below:

$ git add --all

The option "--all" tells Git: "Find all new and updated files everywhere throughout the project and add them to the staging area." Note that you can also use the option "-A" instead of "--all". Thanks to this simple option, "-A" or "--all", the workflow is greatly simplified.

To remove files from the staging area, use the following command:

$ git rm --cached my-file.ts

In our example, we specified the command "rm", which stands for remove. The "--cached" option indicates files in the staging area. Finally, we pass a file that we want to unstage. Git will output the following message for us:

```
$ rm 'my_file.ts'
```

run "git status" again:

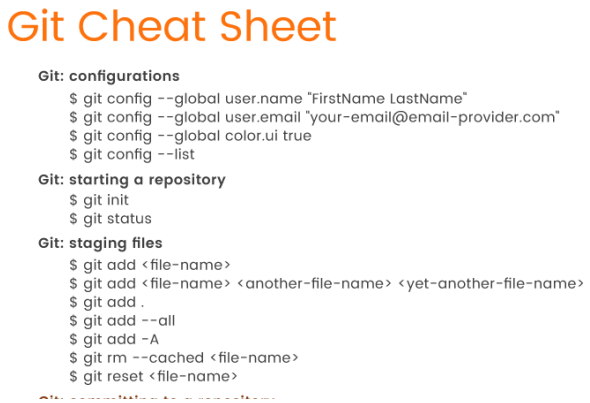
```
$ git status
Changes to be committed:
(use "git rm --cached <file>..."to unstage)
new file: another-file.js
new file: my_new_file.txt
new file: new_file.rb
Untracked files:
(use "git add <file>..."to include inwhat will be committed)
my-file.ts
```

Git is no longer tracking my-file.ts. In this simple way, you can untrack files if necessary. As an alternative to "rm --cached <filename>", you can use the "reset" command:

$ git reset another-file.js

You can consider "reset" as the opposite of "add".

Committing Changes to Git



Git: committing to a repository

```
$ git commit -m "Add three files"
$ git reset --soft HEAD^
$ git commit --amend -m <enter your message>
```

Let's start with a quick overview of committing to the Git repository. By now, you should have at least one file tracked by Git (we have three). As we mentioned, tracked files aren't located in the repository yet. We have to commit them: we need to carry our basket with stuff to the lock box. There are several useful Git commands to do (almost) the same: move (commit) files from the staging area (an imaginary basket) to the repository (a lock box).

There's nothing difficult about committing to a repository. Just run the following command:

$ git commit -m "Add three files"

To commit to a repository, use the "commit" command. Next, pass the "commit" command the "-m" option, which stands for "message". Lastly, type in your commit message. We wrote "Add three files" for our example, but it's recommended that you write more meaningful messages like "Add admin panel" or "Update admin panel". Note that we didn't use the past tense! A commit message must tell what your commit *does* – adds or removes files, updates app features, and so on.

Once we've run "git commit -m 'Add three files'", we get the following output:

```
[master (root-commit) abfbdeb] Add three files
3 files changed, 0 insertions(+), 0 deletions(-)
create mode 100644 another_file.js
create mode 100644 my_new_file.txt
create mode 100644 new_file.rb
```

The message tells us that there have been three files added to the current branch, which in our example is the master or the main branch. The "create mode 100644" message tells us that these files are regular non-executable files. The "0 insertions(+)" and "0 deletions(-)" messages mean we haven't added any new code or removed any code from the files. We actually don't need this information; it only confirms that the commit was successful

So, what have we done so far? We added files to a project directory in the first section. Then we added files to the staging area, and now we've committed them. The basic Git flow looks like this:

- Create a new file in a root directory or in a subdirectory, or update an existing file.
- Add files to the staging area by using the "git add" command and passing necessary options.
- Commit files to the local repository using the "git commit -m <message>" command.
- Repeat.

That's enough to get the idea of Git's flow. But let's get back to committing files. We should mention a great alternative to the standard "git commit -m 'Does something'" command.

When working on a project, chances are you'll modify some files and commit them many times. Git's flow doesn't really change for adding modified files to a new commit. In other words, every time you make changes you'll need to add a modified file to the staging area and then commit. But this standard flow is tedious. And why should you have to ask Git to track a file that was tracked before?

The question is how can we add modified files to the staging area and commit them at the same time. Git provides the following super command:

$ git commit -a -m "Do something once more"

Note the "-a" option, which stands for "add". Git will react to this command like this: "I'll just commit the files immediately. Don't forget to write a commit message, though!" As we can see, this little trick lets us avoid running two commands.

There will be times when you'll regret committing to a repository. Let's say you've modified ten files, but committed only nine. How can you add that remaining file to the last commit? And how can you modify a file if you've already committed it? There are two ways out. First, you can undo the commit:

```
$ git reset --soft HEAD^
```

To understand what that "HEAD" thing represents, recall that we work in branches. Currently we're in the master branch, and HEAD *points* to this master branch. When we switch to a different branch later, HEAD will point to that different branch. HEAD is just a pointer to a branch:

```
$ git add file-i-forgot-to-add.html
$ git commit --amend -m "Add the remaining file"
```

The "--amend" option lets you *amend the last commit* by adding a new file (or multiple files). Using the "--amend" option, you can also overwrite the message of your last commit.

Push and Pull To and From a Remote Repository

Git Cheat Sheet

Git: configurations

```
$ git config --global user.name "FirstName LastName"
$ git config --global user.email "your-email@email-provider.com"
$ git config --global color.ui true
$ git config --list
```

Git: starting a repository

```
$ git init
$ git status
```

Git: staging files

```
$ git add <file-name>
$ git add <file-name> <another-file-name> <yet-another-file-name>
$ git add .
$ git add --all
$ git add -A
$ git rm --cached <file-name>
$ git reset <file-name>
```

Git: committing to a repository

```
$ git commit -m "Add three files"
$ git reset --soft HEAD^
$ git commit --amend -m <enter your message>
```

Git: pulling and pushing from and to repositories

```
$ git remote add origin <link>
$ git push -u origin master
```

```
$ git clone <clone>
$ git pull
```

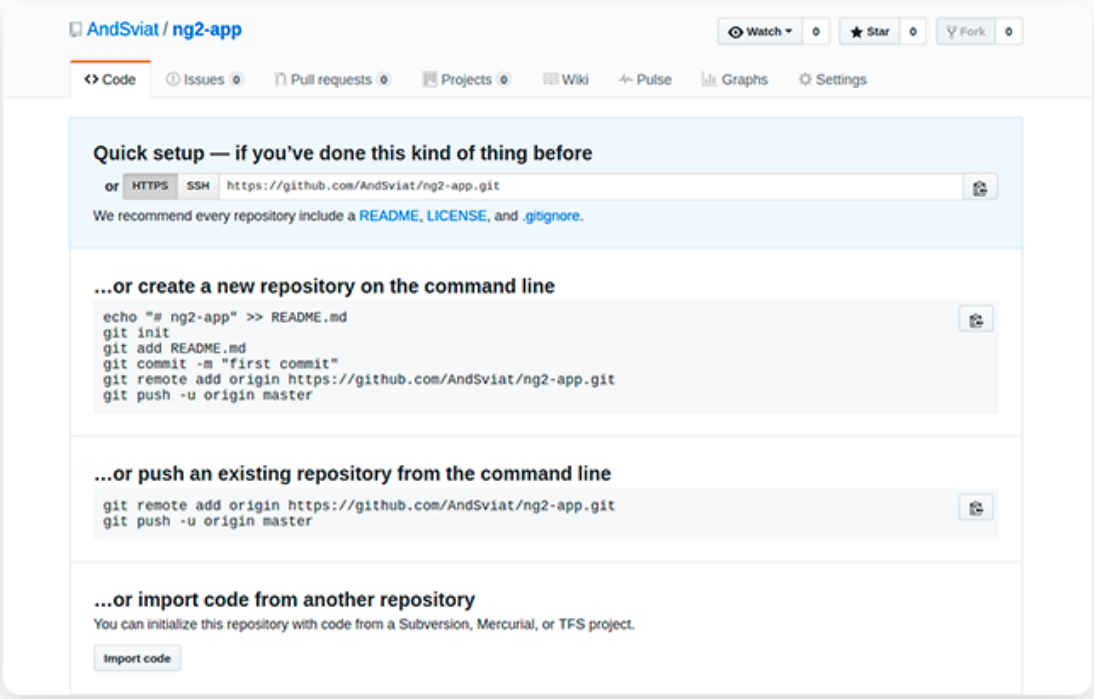
In real development, your workflow will look like this:

- You work on a feature and commit files to a branch (master or any other branch).
- You push commits to a remote repository.
- Other developers pull your commits to their computers to have the latest version of the project.

You'll use several important Git commands to move (push) your code from a local repository to a remote repository and to grab (pull) your team's collective code from a remote repository.

Git has a "remote" command to deal with remote repositories. But before we jump into an explanation of how the "remote" command works, we have to do a little bit of setup.

First things first, you need to create a remote repository. We'll use GitHub for this section. You can create an account on GitHub and create a new repository for your project. Just start a project and give it a name. You then need to grab the HTTPS link to this new repository. The link will look similar to this – https://github.com/YourUsername/some-small-app.git – where YourUsername will be your GitHub username and "some-small-app.git" will be the name of your app.



Now you need to bind this remote repository to your local repository:

```
$ git remote add origin https://github.com/YourUsername/some-small-app.git
```

view rawadding-remote-repository.sh hosted with ❤ by GitHub

We tell Git to "add" a repository. The "origin" option is the default name for the server on which your remote repository is located. Lastly, there's a link to your project on GitHub.

At RubyGarage, we use GitHub to release high-quality code. Learn how we manage this.

Once you run the command above, Git will connect your local and remote repositories. But what does this liaison actually mean? Can you already access your

code online? Unfortunately, not yet.

All you did for now is signed papers so that the remote lock box (GitHub) can accept various items (your code) from your home drawer (local repository). To actually copy your things to a remote lock box, you need to personally carry them to it. With Git, copying your code to a remote repository looks like this:

```
$ git push -u origin master
```

view rawpushing-to-remote-master-branch.sh hosted with ❤ by GitHub

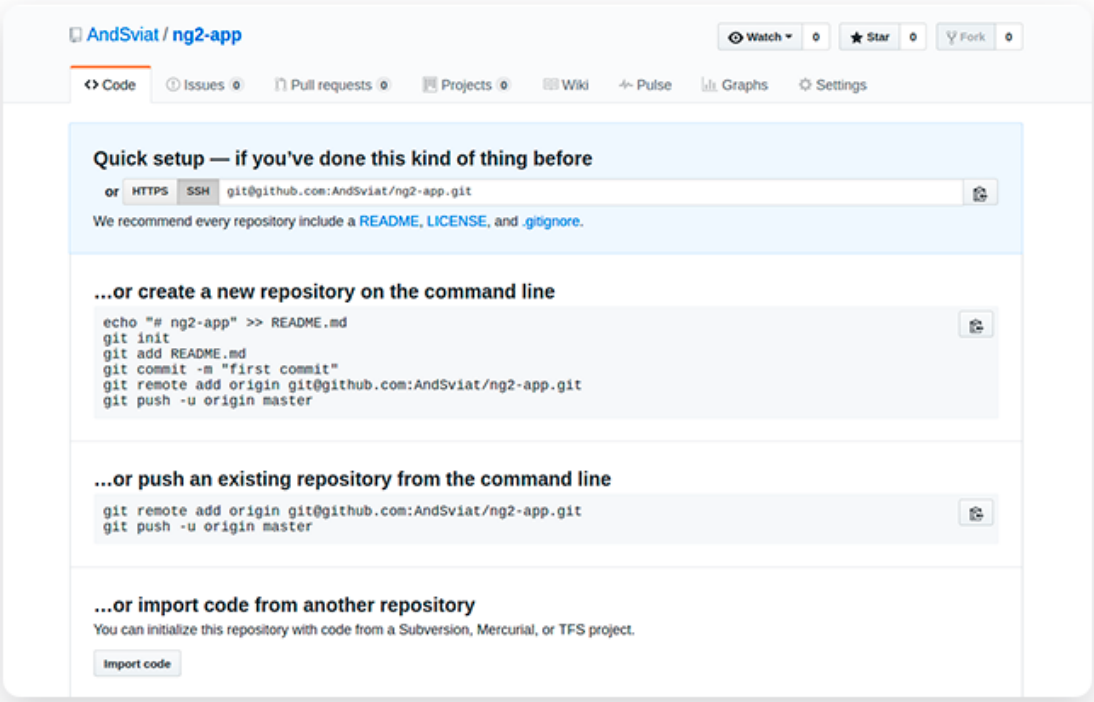
It's obvious that the command "push" tells Git to push your files to a remote repository. What we also specified is the server our local repo is connected to (origin) and the branch we're pushing, which is master. There's also that strange "-u" option. What it means is that we're lazy enough not to run a long "git push -u origin master" command each time we push code to the cloud. Thanks to "-u", we can run only "git push" next time!"

Once you've pushed changes to a remote repository, you can develop another feature and commit changes to the local repository. Then you can push all changes to the remote repository once again, but using only the "git push" command this time around. As we can see, Git tries to simplify things as much as possible.

Is this the happy ending? Not yet. Once you push code to a remote repository, you have to enter your username and password registered with that remote repository.

The current problem with "git push" is that you have to enter your credentials each time you push code to GitHub. That's not convenient. The root of this problem is the HTTPS link you used to connect repositories. Git offers a way out of this inconvenience, however.

Let's get back to the page where GitHub offered a link to our project:



As we can see, there's an SSH option that we can use instead of HTTPS. If you set up Git on your computer to work with SSH, then you won't have to enter GitHub credentials every time you push code to GitHub. You'll only need to add a remote origin with this SSH link, like this:

```
$ git remote add origin git@github.com:YourUsername/your-app.git
```

view rawadding-remote-repository-with-ssh-link.sh hosted with ❤ by GitHub

As you can see, to connect repositories via SSH we only changed the link. Keep in mind that HTTPS is the default protocol used for connecting with GitHub, and you

need to manually set up SSH.

Now that you've added a remote repository, you can view the list of repositories by running the following command:

```
$ git remote -v
```

view rawlisting-remote-repositories-in-console.sh hosted with ❤ by GitHub

The "-v" option will list all remote repositories you've connected to. This is what we've got:

```
origin https://github.com/YourName/some-app.git (fetch)
origin https://github.com/YourName/some-app.git (push)
```

view rawlist-of-remote-repositories.sh hosted with ❤ by GitHub

So far we've talked about how to move your things to a remote lock box. Let's say your drawer with all your valuables has disappeared from your home. Now you want to get your things back from the lock box, kind of like cloning them. Git can clone an entire project from a remote repository. That's what the "clone" command does:

```
$ git clone git@github.com:YourUsername/your-app.git
```

view rawcloning-remote-repository.sh hosted with ❤ by GitHub

"clone" is a simple command. We only need to pass it a link to the GitHub project. We've used an SSH link, but you can use the HTTPS link with the same command.

What "git clone" does is it copies the entire project to a directory on your computer. The directory will be created automatically and will have the same project name as the remote repository.

If you don't like the name of the repository you're cloning, just pass your preferred name to the command:

```
$ git clone git@github.com:YourUsername/your-app.git this-name-is-much-better
```

view rawcloning-remote-repository-and-changing-its-name.sh hosted with ❤ by GitHub

So far, you've pushed your changes from a local repository to a remote repository and cloned a remote repository. We haven't said anything about the "pull" command, though. Pushing changes to GitHub or BitBucket is great. But when other developers push their changes to a remote repository, you'll want to see their changes on your computer. That is, you'll want to *pull* their code to your local repository. To do so, run the following command:

```
$ git pull
```

view rawpulling-from-remote-repository.sh hosted with ❤ by GitHub

Running "git pull" is enough to update your local repository.

Wait, but you said I could clone a repository. Why do I have to pull something? Anonymous Developer

That's a valid question, so let's clarify.

Cloning a repository is very different from *pulling* from a repository. If you clone a remote repository, Git will:

- Download the entire project into a specified directory; and
- Create a remote repository called origin and point it to the URL you pass.

The last item simply means that you don't need to run "git remote add origin git@github.com:YourUsername/your-app.git" after cloning a repository. The "clone" command will add a remote origin automatically, and you can simply run "git push" from the repository.

Grab this a step-by-step guide to the RubyGarage git and release management to set the efficient workflow.

When you run the "pull" command, Git will:

- Pull changes in the current branch made by other developers; and
- Synchronize your local repository with the remote repository.

The "pull" command doesn't create a new directory with the project name. Git will only pull updates to make sure that your the local repository is up to date.

Basically, that's all you need to know about pushing, pulling, and cloning with Git. One set of basic Git commands is left, though. Remember how we mentioned branches in the beginning of the article? Why do we ever need branches? And what are the basic Git commands to view, add, and delete branches? Let's find out.

The Git cheat sheet continues to grow:

Working with Branches

You'll use multiple branches for your projects. Branches are, arguably, the greatest feature of Git, and they're very helpful. Thanks to branches, you can actively work on different versions of you projects simultaneously.

The reason why we use branches lies on the surface. If you have a stable, working application, you don't want to break it when developing a new feature. Therefore, it's best to have two branches: one branch with a stable app and another one for developing features. Then again, when you complete a feature and it seems to be working, some bug may still be there. And bugs must not appear in a production-ready version. Thus, you'll want to have another branch for testing.

Managing branches in Git is simple. Let's first see our current branches:

```
$ git branch
*master
```

That's it: one command, "branch", will ask Git to list all branches. In our app, we have only one branch – master. But an application under development is far from being complete, and we need to develop new features. Let's say we want to add a user profile feature. To create this feature, we need to create a new branch:

```
$ git branch user-profile
```

Again, it's very simple: the "branch" command creates a new branch with the name we gave it: "user-profile". Can we write code for our new feature right away? Not yet!

Let's run the "git branch" command once more:

```
$ git branch
```

The output will be the following:

```
*master
user-profile
```

Note the asterisk to the left of "master." This asterisk marks the current branch you're in. In other words, if you create a branch and start changing code right away, you'll still be editing the previous branch, not the new one. After you've created a new branch to develop a feature, you need to switch to the new branch *before* you get to work on a feature.

For switching branches in Git, you won't use a "switch" command, as you might think. Instead, you'll need to use "checkout":

```
$ git checkout user-profile
Switched to branch 'user-profile'
```

Git also notifies you that you've switched to a different branch: "Switched to branch 'user-profile'". Let's run "git branch" once more to prove that:

master

*user-profile

1. Create a new branch to develop a new feature using "git branch <branch-name>".

2. Switch to the new branch from the main branch using "git checkout <branch-name>".

3. Develop the new feature.

You're stuck on the third step. What should you do next? The answer is simple: you need to use the "merge" command. To merge a secondary branch into the main branch (which can be a master, development, or feature branch), first switch back to the main branch. In our case, we should checkout the master branch:

$ git checkout master

*master

user-profile

The current branch is now changed to master, and we can merge the user-profile branch using the command "merge":

$ git merge user-profile

Keep in mind that you're in the main branch and you're merging another branch into the main – not vice versa! Now that the user-profile feature is in the master branch, we don't need the user-profile branch anymore. So let's run the following command:

$ git branch -d user-profile

With the "-d" option, we can delete the now unnecessary "user-profile". By the way, if you try to remove the branch you're in, Git won't let you:

error: Cannot delete the branch 'user-profile'which you are currently on.

Let's mention a simpler command for creating new branches than "git branch <name>". Given that you're in the main branch and you need to create a new branch, you can just do this:

$ git branch my-new-branch

$ git checkout my-new-branch

But instead of running two commands you can run only one:

$ git checkout -b admin-panel

This one command will let you create a new "admin-panel" branch and switch to that branch right away. Git earns another point for improving the workflow.

Rename a file using git's mv command:

$ git mv file_from file_to

Code:

git mv helo.txt hello.txt

•