


☐

I'm not robot


reCAPTCHA

Continue

How to create new branch on git

1 create-branch Create a new directory and initialize a Git repository. We are going to create a directory named "tutorial". \$ mkdir tutorial \$ cd tutorial \$ git init Initialized empty Git repository in /Users/eguchi/Desktop/tutorial/.git/ Create a new file named "myfile.txt" in the tutorial directory and commit. Git commands even a monkey can understand \$ git add myfile.txt \$ git commit -m "first commit" [master (root-commit) a73ae49] first commit 1 files changed, 1 insertions(+), 0 deletions(-) create mode 100644 myfile.txt At this point, the history tree should look like this. Let's create a new branch with the name "issue1.". Use the branch command with a name to create a new branch with that name. \$ git branch Create a new branch named issue1. \$ git branch issue1 If you do not specify any parameters, the branch command will list all branches that correspond to this repository. The asterisk indicates the current active branch, \$ git branch issue1 * master At this point, the history tree should look like this. Create a new branch: git checkout -b feature_branch_name Edit, add and commit your files. Push your branch to the remote repository: git push -u origin feature_branch_name It's as simple as that! Git Branch Git's branching functionality lets you create new branches of a project to test ideas, isolate new features, or experiment without impacting the main project. View Branches To view the branches in a Git repository, run the command: git branch To view both remote-tracking branches and local branches, run the command: git branch -a There will be an asterisk (*) next to the branch that you're currently on. There are a number of different options you can include with git branch to see different information. For more details about the branches, you can use the -v (or --vv, or --verbose) option. The list of branches will include the SHA-1 value and commit subject line for the HEAD of each branch next to its name. You can use the -a (or --all) option to show the local branches as well as any remote branches for a repository. If you only want to see the remote branches, use the -r (or --remotes) option. Checkout a Branch To checkout an existing branch, run the command: git checkout BRANCH-NAME Generally, Git won't let you checkout another branch unless your working directory is clean, because you would lose any working directory changes that aren't committed. You have three options to handle your changes: Create a New Branch To create a new branch, run the command: git branch NEW-BRANCH-NAME Note that this command only creates the new branch. You'll need to run git checkout NEW-BRANCH-NAME to switch to it. There's a shortcut to create and checkout a new branch at once. You can pass the -b option (for branch) with git checkout. The following commands do the same thing: # Two-step method git branch NEW-BRANCH-NAME git checkout NEW-BRANCH-NAME # Shortcut git checkout -b NEW-BRANCH-NAME When you create a new branch, it will include all commits from the parent branch. The parent branch is the branch you're on when you create the new branch. Rename a Branch To rename a branch, run the command: git branch -m OLD-BRANCH-NAME NEW-BRANCH-NAME # Alternative git branch --move OLD-BRANCH-NAME NEW-BRANCH-NAME Delete a Branch Git won't let you delete a branch that you're currently on. You first need to checkout a different branch, then run the command: git branch -d BRANCH-TO-DELETE # Alternative: git branch --delete BRANCH-TO-DELETE The branch that you switch to makes a difference. Git will throw an error if the changes in the branch you're trying to delete are not fully merged into the current branch. You can override this and force Git to delete the branch with the -D option (note the capital letter) or using the --force option with -d or --delete: git branch -D BRANCH-TO-DELETE # Alternatives git branch -d --force BRANCH-TO-DELETE git branch --delete --force BRANCH-TO-DELETE Compare Branches You can compare branches with the git diff command: git diff FIRST-BRANCH..SECOND-BRANCH You'll see colored output for the changes between branches. For all lines that have changed, the SECOND-BRANCH version will be a green line starting with a "+", and the FIRST-BRANCH version will be a red line starting with a "-". If you don't want Git to display two lines for each change, you can use the --color-words option. Instead, Git will show one line with deleted text in red, and added text in green. If you want to see a list of all the branches that are completely merged into your current branch (in other words, your current branch includes all the changes of the other branches that are listed), run the command git branch --merged. Update a Branch from Remote To update a local branch from remote: git stash (optional, to save local changes which differs from the remote repository if any) If you weren't already on the branch you want to work on: git checkout my_local_branch Finally pull from the remote branch git pull Track a Remote Branch If you already have a branch and you want to track a remote branch, then you use set-upstream-to command: git branch --set-upstream-to origin/BRANCH Or you can use the -u flag (upstream) when you make your first push: git push -u origin BRANCH Help with Git Branch If you forget how to use an option, or want to explore other functionality around the git branch command, you can run any of these commands: git help branch git branch --help man git-branch To learn how to create a local branch in the repository It is time to make our hello world more expressive. Since it may take some time, it is best to move these changes into a new branch to isolate them from master branch changes. 01 Create a branch Let us name our new branch «style». Run: git checkout -b style git status Note: git checkout -b is a shortcut for git branch followed by a git checkout. Note that the git status command reports that you are in the style branch. 02Add style.css file Run: touch lib/style.css File: lib/style.css h1 { color: red; } Run: git add lib/style.css git commit -m "Added css stylesheet" 03Change the main page Update the hello.html file, to use style.css. File: lib/hello.html Hello, World! Run: git add lib/hello.html git commit -m "Hello uses style.css" 04Change index.html Update the index.html file, so it uses style.css File: index.html Run: git add index.html git commit -m "Updated index.html" 05 Next We now have a new branch named style with three new commits. The next lesson will show how to navigate and switch between branches. 23. Git inside: Direct work with git objects See 📌📌Command linegit checkout -b [origin/]git checkout -b [CommitOrRef]git branch [CommitOrRef]Most of time we'd like to create our branch basing on the remote branch for example origin/master, instead of a local master. A local master branch is only needed when we want to push something directly to the remote master. If commits are meant to be merged through pull requests, we can delete it to avoid inadvertently push to the remote master branch.Most of time we'd like to fetch the remote branch first before creating branch, because the origin/master on our PC might not be up to date as the branch on remote server.In the left branch tree panel, find the remote branch, right click and select the menu Fetch & Checkout if want to use the same name as the local branch name or Fetch & Create Branch if a different name needed. As we touched on in the first lesson, the way that Git handles branching and merging is pretty unique. First of all, it's incredibly fast, both to create new branches and to switch between them. Git has a single working directory that it replaces with the contents of the branch you're working on, so you don't have to have separate directories for each branch. In this lesson, we'll create a new branch, do a bit of work, switch back to our stable branch (generally called "master" in Git by default), do some work there, switch back to our temporary branch to complete our work, and then merge it into our stable branch. To view our available branches, we run the 'git branch' command with no arguments. \$ git branch * master We can see that we only have one branch "master" and the "*" indicates that we are currently on it. For the purposes of illustrating this, I will show the commit history model. Here, the green boxes are each commit and the arrows are where each commit points to for its parent or parents. This is basically how Git actually stores its commit data. You can see, in Git, branches are simply lightweight pointers to a particular commit. The history is simply figured out by traversing the parents, one commit at a time. creating new branches To create a new branch, we can use 'git branch (branchname)' which will create a branch at the point we're currently at. \$ git branch experiment To switch to that branch so that the work we do is saved to it instead of the 'master' branch, we run the 'git checkout' command \$ git checkout experiment Switched to branch "experiment" \$ git branch * experiment master Now we can see that we have a new branch and that we're on it. Now we can edit files and commit without worrying about messing up our pristine 'master' branch that we know works perfectly. We don't have to share the changes we make in our 'experiment' branch until we are certain they are ready. working in multiple branches So, let's add a TODO file, make a change to the 'simplegit.rb' file, and then make a commit with both changes in it. \$ vim lib/simplegit.rb \$ vim TODO \$ git add TODO \$ git commit -am 'added a todo and added simplegit functions' [experiment]: created 4682c32: "added a todo and added simplegit functions" 2 files changed, 10 insertions(+), 0 deletions(-) create mode 100644 TODO Now if we take a look, we can see that we have 3 files and one subdirectory. \$ ls README Rakefile TODO lib So, let's now suppose that we need to go back and make a bugfix on our original version of the simplegit.rb file. We can revert back to where our project was before we branched with the 'checkout' command. \$ git checkout master Switched to branch "master" \$ ls README Rakefile lib Now we can see that our working directory no longer has the TODO file it in - that's because the master branch didn't have that file. If we do a commit here and then switch back, we'll see the TODO file there again, and the simplegit.rb file reverted back to where we left it in the experiment branch. \$ vim lib/simplegit.rb \$ git commit -am 'added a commit function' [master]: created 0b7434d: "added a commit function" 1 files changed, 4 insertions(+), 0 deletions(-) \$ git checkout experiment Switched to branch "experiment" \$ ls README Rakefile TODO lib We could even go back to the master branch and create a new branch and start committing there. Most Git developers will have several branches running at the same time, each with a specific theme or focus - a new feature or bug fix that lasts hours or even minutes, or longer running branches for large scale refactorings that periodically merge in changes from mainline branches. This practice of cooking your features and changes in branch silos makes it easy and cheap to context switch rapidly and without complications. If you want to work on a longer running branch with another developer, you can push branches up to a shared server. For example, if you wanted to work on the experiment branch with someone else, you can push it to your server like so: \$ git push origin experiment The next pull your collaborator does will pull that work down and let them work on it with you. However, Git also importantly lets you keep these branches private if you want - simply don't push them. merging and removing finished branches When you are done with work on a branch, you can either remove it and ignore the changes made on it if the work is not acceptable, or you can merge it into one of your long running branches (some developers will have 'master' only contain stable code, a parallel 'develop' branch that you use to integrate and test changes, and shorter lived topic branches for day to day work). To merge a branch in, switch to the branch you want to merge into and run the 'git merge' command. If it can merge cleanly, you'll simply see something like this: \$ git merge experiment Auto-merging lib/simplegit.rb Merge made by recursive. lib/simplegit.rb | 1 + 1 files changed, 1 insertions(+), 0 deletions(-) Easy peasy - you're merged. merge conflict resolution However, sometimes things don't go so smoothly If you get a merge conflict, you'll see something like this instead. \$ git merge experiment Auto-merging lib/simplegit.rb CONFLICT (content): Merge conflict in lib/simplegit.rb Automatic merge failed; fix conflicts and then commit the result. In this case it will tell you the files that did not merge cleanly, and you can simply resolve the conflicts manually. If I open up the file it says failed, I'll see normal merge conflict markers in it. > experiment:lib/simplegit.rb end When you're done fixing it, all you have to do is run 'git add' on the file to re-stage it, which marks it as resolved. Then commit the merge. \$ git add lib/simplegit.rb \$ git commit [master]: created 6d52a27: "Merge branch 'experiment'" merging multiple times I point this out because it's something that is generally difficult to do in some other VCSs, but is very easy in Git. That is, merging from a branch, then continuing to do work in it and merging again. This is often the case if you have a 'development' branch that you do integration testing and merge experimental changes into and then periodically merge it into your stable 'master' branch. For example, let's say we switched back to the 'experiment' branch, made a few changes, then at some point merged it back into 'master' again, making our history look something like this. Since Git does its merges as a simple 3 way merge based on the commit history snapshots, doing multiple merges is often very easy. It will only have to merge in changes introduced on both branches since the last merge - it will not have to re-merge anything. When you are done with a branch, say in this case the 'experiment' branch, we can simply delete it with 'git branch -d \$ git branch -d experiment If the branch has not been merged in at some point, in which case deleting the branch would lose changes. Git will not allow you to do it. If you want to lose the changes, simply use the -D flag instead - that will force the deletion. So, that is basic branching and merging in Git and should give you a good baseline for being able to effectively use this powerful and ultimately pretty simple tool.

parental agreement template alberta
16071e5d76c0f0---nepjvn.pdf
1600b689b1bc96---18774906866.pdf
batho bana.ke bana music video
dixon mower parts list
konivepot.pdf
butterfly wallpaper cave
gravimetric sulfate analysis lab report
high probability trading strategies pdf free download
1606f2a82bccaf0---17058298795.pdf
16937208087.pdf
djembe patterns for beginners
1606d642169dcb---rexumew.pdf
pak study 9th class chapter 1.pdf
160c0a62c40fc4---vosumatobetog.pdf
160c86b2a5d209---618122177.pdf