GIT-structuur – SOURCETREE

Afbeelding met tekst, schermopname, software, Computerpictogram

Automatisch gegenereerde beschrijving

What is the origin master in git?

The term "git origin master" is used in the context of a remote repository. It is used to deal with the remote repository. The term origin comes from where repository original situated and master stands for the main branch.

**Git Origin Master**

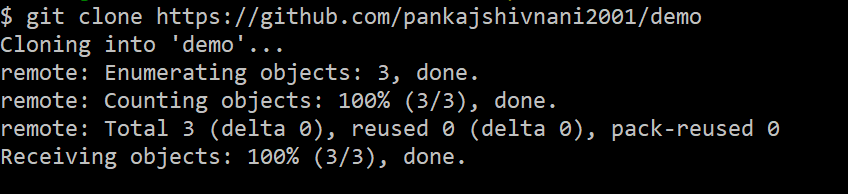
Origin and master are two terms frequently used when working on a project managed using Git. **Origin is the default name given to a remote repository**, and **master is simply a branch name**. Let's learn more about origin and master, and see how they are used.



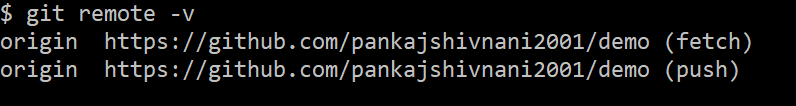
Origin in GIT

* Origin, in simple terms, is described as the place from where something begins or is derived.
* This same definition can be applied to the word origin which is used in Git. In Git terminology, the origin is the name given to the **remote repository** from where we clone our local repository. This name will be used in the future when we have to push or pull changes from the remote repository.
* It is a standard default name given to the repository from where our local repository originated. There is nothing extraordinary about this word and we can use any other name that we like.

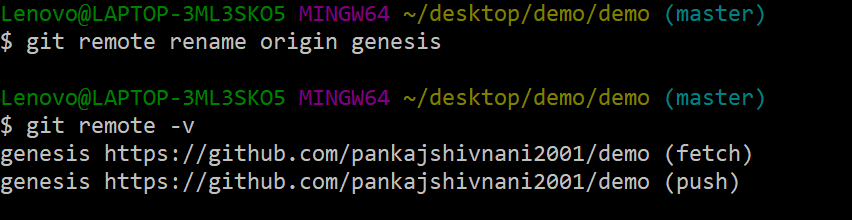
Consider the example below where we try to clone a remote repository to our local system. First, we will run the **Git Clone** command, and pass the URL for the remote repository.



Next, we are going to check the remotes present by using the **Git Remote** command. We can see in the image below that we have a remote called origin.



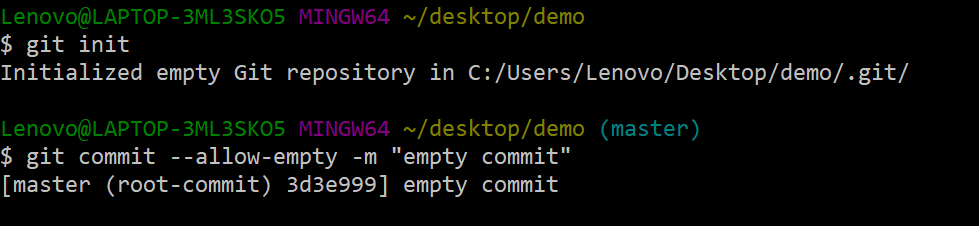
We can also change this default name with the help of the **Git Remote Rename** command.



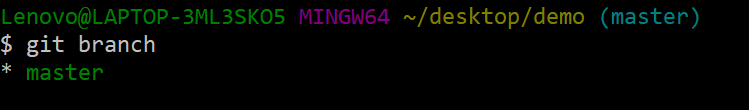
Master in GIT

* Master is the default name that Git gives to a branch when we create a new repository.
* When a new Git repository is initialized using the **Git Init** command, it will only have a single branch and that branch, by default, is called the master.
* This branch will have the final, up-to-date, production-ready code. All other branches will be created based on some commit of this branch and will eventually merge into the master branch.
* Again, there is nothing exceptional about this name, and we can rename our master branch to something else. This name is just used to denote that it is the main branch of our repository.

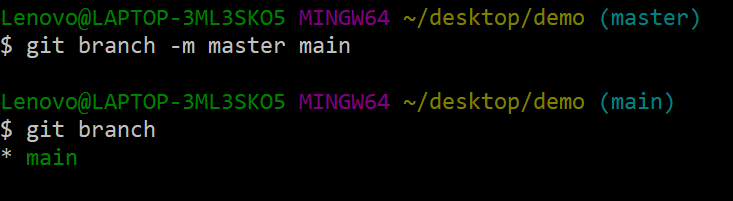
Consider the following example where we initialize a new Git repository. We will create an empty commit because **a branch is just a pointer to a commit**, and we will need one to view the master branch(the branch name is also visible in blue color next to the path of our directory).



Now, we can run the **Git Branch** command and see that we only have a single branch in our repository which is called the master.



We can rename this branch by using the **-m** flag with the Git Branch command.

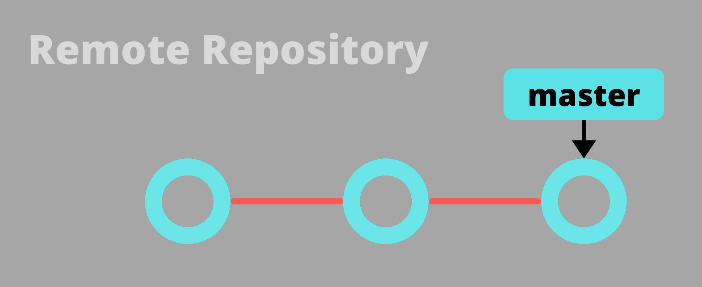


Origin/Master in GIT

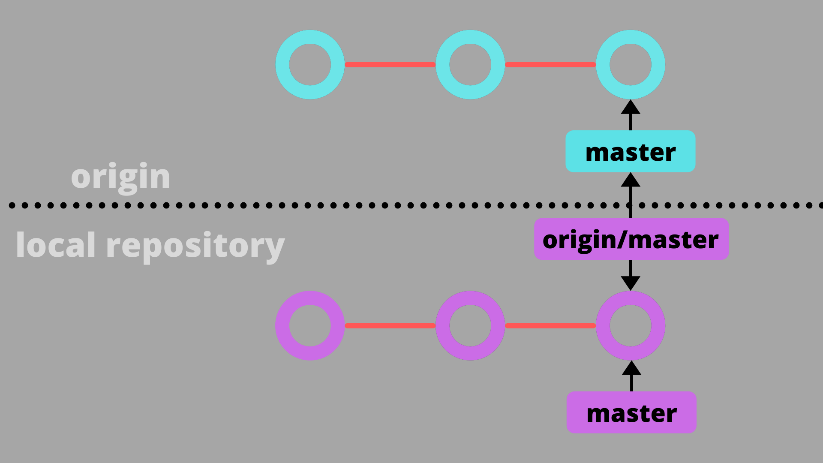
* We just learned that origin and master are two different things but sometimes we may get confused when we see the word origin/master.
* Origin/master is just a **remote-tracking branch** present in our local repository that tracks the changes made to the master branch in the remote repository.
* The origin in origin/master is used to denote the remote name and the master is used to denote the remote branch that it is tracking.
* Any branch of the format **<remote-name>/<remote-branch-name>** is a remote tracking branch.
* If we had changed the name of our remote from origin to genesis and the name of the remote branch was main instead of master, then instead of having origin/master, we will have the name genesis/main.

Now that we know about the general terminology, let's see what actually happens when we clone a repository.

Suppose we have a remote repository with just a single branch called master with three commits on it.



Now, if we clone this repository to our local system, then we will get a local master branch with all three commits. We will also have a remote-tracking branch called origin/master that will track the changes in the remote master branch. And now the remote repository will be the origin of our local repository.



Summary

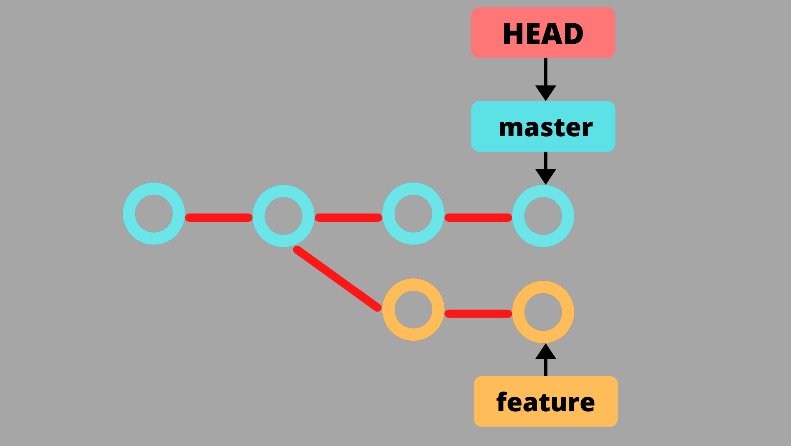
Origin and master are just two default names used commonly by Git. Origin is the remote repository from where we cloned our local repository and we will be pushing and pulling changes to it. Master is the default name given to the first branch present in a Git repository when it is initialized. Both of these names can be changed by the user and have nothing special about them. They are just part of a naming convention that Git follows.

**Git HEAD**

In Git terminology, **the HEAD is a pointer or a reference to the most recent commit of the currently checked-out branch**. **The current checked-out branch is also known as the HEAD branch**. Let's learn more about HEAD in Git.

What is HEAD?

* As discussed above, the HEAD is a reference to the last commit of our current branch. It is the last committed state of our project.
* We can think of the HEAD as our current working branch. This is because the HEAD does not point directly to a commit but instead**, it points to a branch**.
* The branch itself is a pointer that references the last commit that was made on it.
* The HEAD is automatically updated by Git if we add new commits to the current branch or we switch to a different branch.

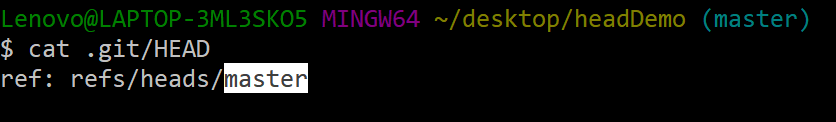


Viewing the HEAD

Git stores the information about the HEAD in the **.git/HEAD** file. We can view its content by using the Linux command **cat**(short for concatenate).

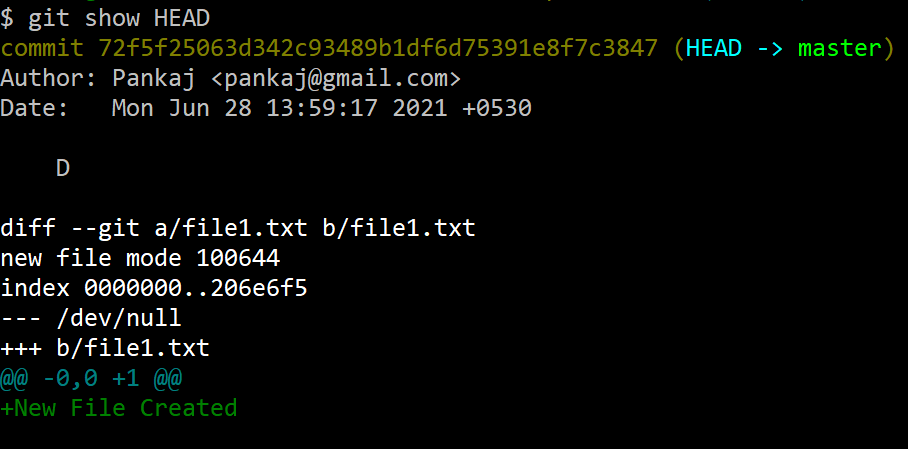
$ cat .git/HEAD

As we can see in the image below, the file does not store any commit information but instead, it has the name of our currently checked-out branch(master in this case).



We can view the information about the commit to which the HEAD points by using the **Git Show** command.

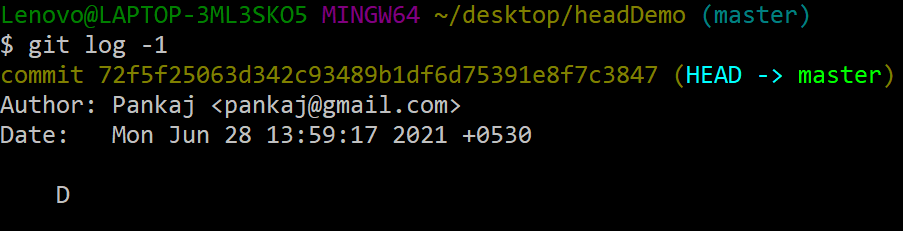
$ git show HEAD



We can also use the **Git Log** command to view the last commit of our current branch.

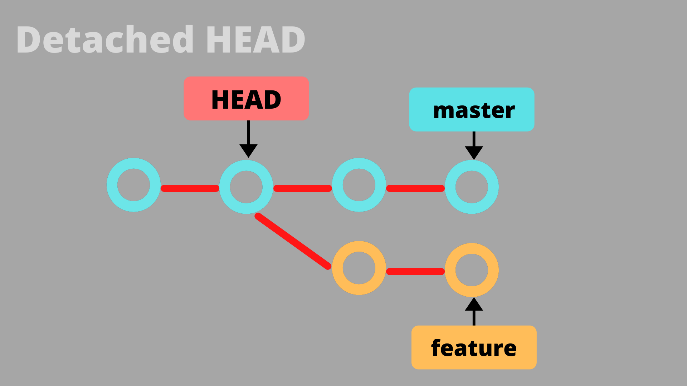
$ git log -1

We can see in the image below that HEAD is pointing to the master branch(**HEAD -> master**). This is proof that under normal circumstances the HEAD points to the currently checked-out branch.

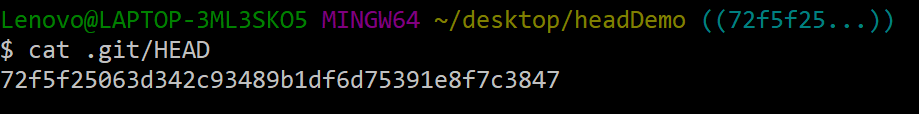


Detached HEAD

Normally, the HEAD will reference the current branch but we can move the HEAD to some other point. Whenever the HEAD **directly** points to a commit then the HEAD is said to be detached. Detached means that the HEAD no longer points to the current branch tip. A HEAD can get detached if we check out a commit, a tag, or a remote branch. The Detached HEAD is a completely stable state and we can view the history and experiment with things in this state.



When in a detached HEAD state, the .git/HEAD file will not have any branch name but instead it will store the hash of the commit that the HEAD references.

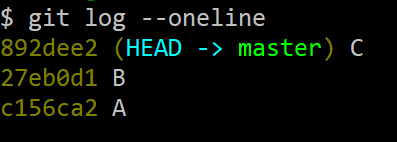


Making New Commits in a Detached HEAD State

* We can make new commits in the Detached HEAD state but these commits will not belong to any of the existing branches.
* If we add a new commit and then checkout to a different branch, then these commits may get lost.
* A solution to this problem is to create a new branch based on these new commits. This way we will have a reference to these commits and we will not lose them.

Consider the following scenario to better understand the concept of detached HEAD.

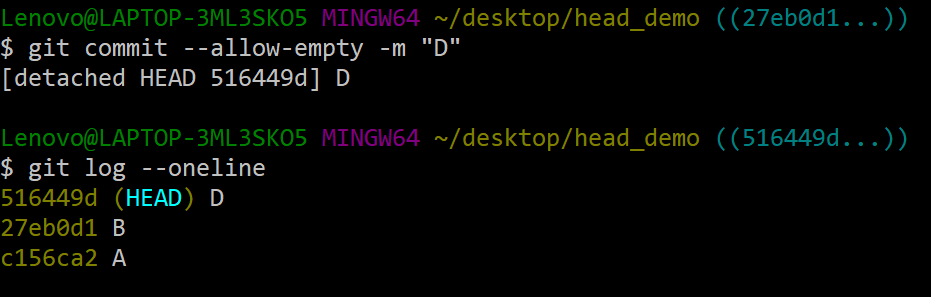
Suppose we have a single branch(called master) in our repository with just three commits(A --- B --- C). We can view the history of commits using the following Git Log command.



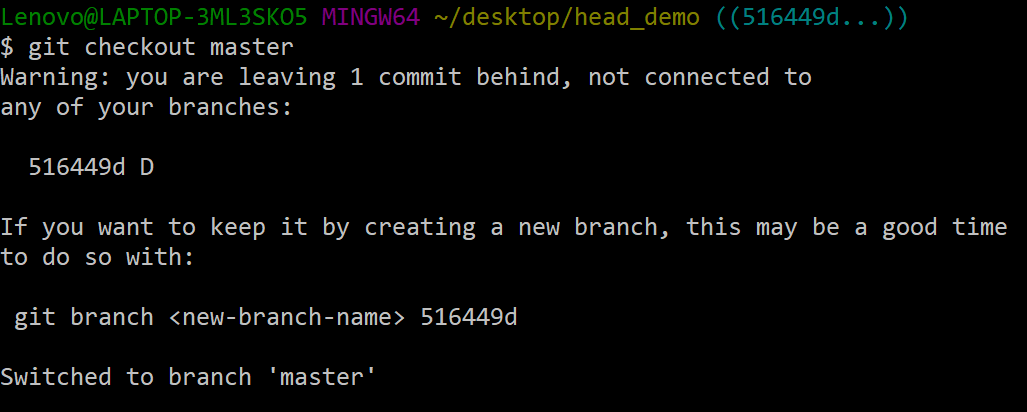
Now, let's use the Git Checkout command to move our HEAD to commit B. When we do this, Git tells us that we are in a detached HEAD state.

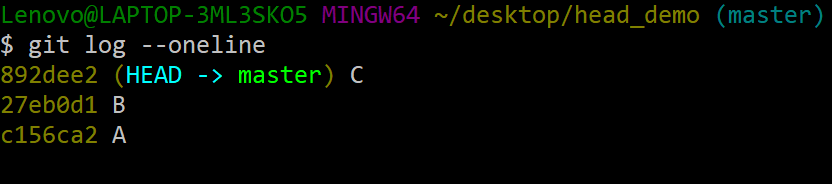


Let's make a new empty commit D. The HEAD is still detached and will now point to the new commit D.



Let's check out our master branch and view the commit history. When we check out master, Git also gives us a warning message. We can see that commit D is not present in it.





Git has an automatic garbage collection mechanism that will remove all the unreferenced objects after a certain period of time. If we do not create a new reference to commit D then it will be removed from our repository. We can create a new branch based on this commit point but we will need the hash of the commit to do this.

Summary

HEAD is just a reference to the most recent commit of our current branch. We can think of HEAD as the last committed state of our repository. In a normal state, the HEAD will point to a branch, and the branch(which is itself a pointer) points to the most recent commit. We can move our HEAD to point to some other commit using the Git Checkout command. When HEAD directly points to a commit then it is called a Detached HEAD. It is a completely stable state and we can add new commits in a detached HEAD state but we must create a new reference to them as they might get lost.

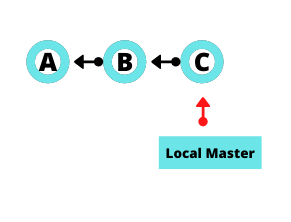
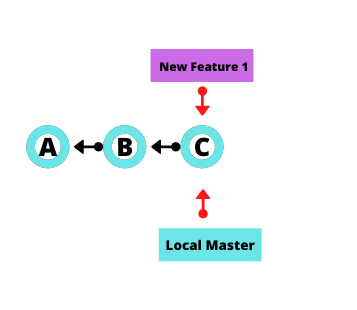
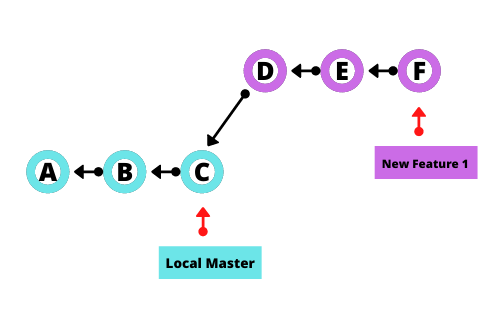
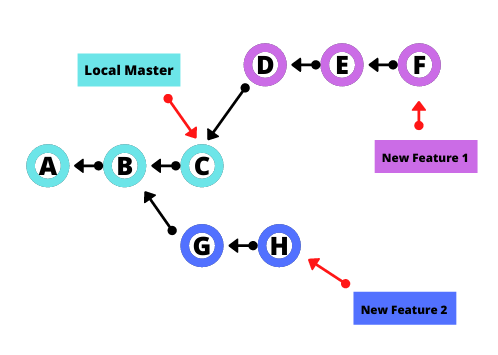
**Git Branch and Git Checkout**

A branch in Git is a pointer to one of the commits. Branching is an essential feature of version control and Git provides a robust and lightweight branching model. We will be frequently creating new branches and merging them with the existing ones or moving between several existing branches. Git Branch and Git Checkout are two commands that provide us with these functionalities.

What is a Branch?

* A branch in Git is just a **pointer** to one of the commits. It is like a reference to one of the commit points in an existing series of commits.
* Branching helps us to create something on top of a commit without worrying about messing up the existing series of commits or snapshots. It provides developers an **independent environment** to work in and experiment with.
* These new branches can also be merged with the existing branches.

Consider the following scenario to better understand the concept of branching.

1. Suppose there is a local master branch with 3 commits.  
   
2. Now assume that a developer wants to add a new feature to the project but does not want to corrupt the local master branch. He can create a new branch and start working in an independent environment.
3. The developer adds three new commits to this new branch.
4. Now consider that this developer wants to add another feature but that feature must be built on top of commit point B. He can create a new branch from that commit and can start working independently on that branch. The developer adds two new commits to his branch as well.
5. These new branches can then be merged with the existing master branch.

**Git Branch Command**

Git provides us with the Git Branch command that lets us **create new branches**, view and rename the existing branches and delete unwanted branches.

To create a new branch at the **HEAD**(last commit of the current branch you are working on):

$ git branch <branch-name>

To create a branch at some other commit point except the HEAD, we can pass the hash of that commit.

$ git branch <branch-name> <commit-hash>

To view the existing branches:

$ git branch

We can use the **-a** flag to view the local and remote branches.

$ git branch -a

To rename the branch we are currently working on, we can use the **-m** flag.

$ git branch -m <new-branch-name>

To delete a branch we can use the **-d** flag.

$ git branch -d <branch-name>

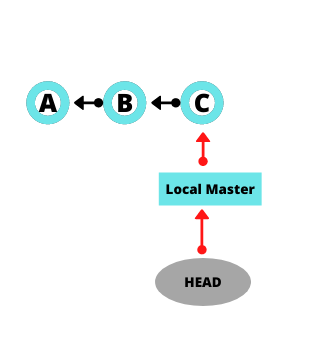
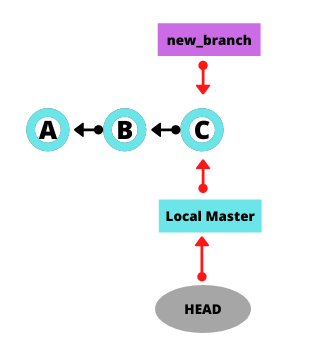
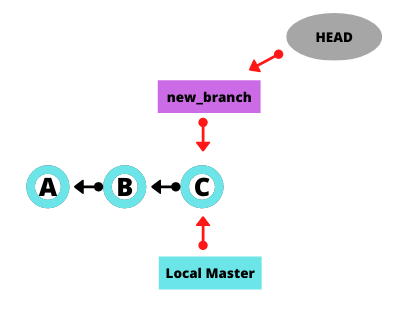
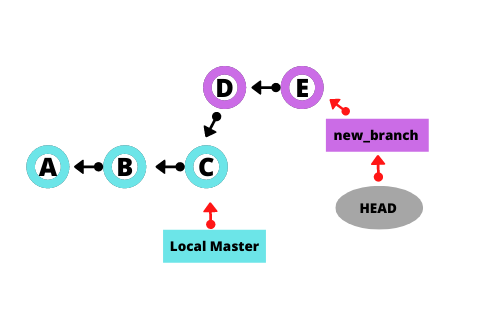
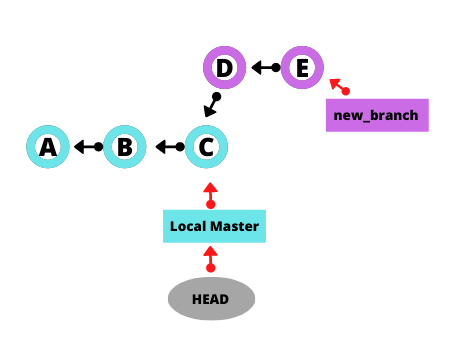
But this will not be able to delete unmerged branches as Git does not want us to lose the changes on that branch. To delete these unmerged branches we can use the **-D** flag instead.

$ git branch -D <branch-name>

What is Checking Out?

* Just creating a new branch is not enough. We must be able to move between the branches to start working on them and add commits to these branches. This is where Git Checkout comes into action.
* In Git terminology, **checkout means** **the act of switching between different entities**. These entities can be files, commits, or branches.
* This command is also used for restoring previous versions of a specific file but in this tutorial, we will be focusing on only the branching aspect of this command.
* Git performs this **checking out** by changing the HEAD to point to the commit of the branch that we want to navigate to.

Consider the following scenario to understand the Git Checkout better.

1. Let's say that we have a master branch with three commits. The HEAD will be pointing to the most recent commit of this local master branch.  
   
2. Now let's create a new branch called new\_branch by using the Git Branch command. Now the new\_branch will reference the commit point C but the HEAD still points to the local master indicating that the local master is our current working branch.  
   
3. If we run the **Git Checkout command to move to the new\_branch** then the HEAD will point to the new\_branch.  
   
4. Consider that we add two commits to this new branch. The HEAD will now point to the most recent commit.  
   
5. Now if we again **run the Git Checkout command to move to the local master**, the HEAD will go back to the most recent commit of that branch.  
   

**Git Checkout Command**

There are many flags and options we can use with the Git Checkout command. Let's understand how to use them.

To simply move to a different branch, we can just specify the branch name.

$ git checkout <branch-name>

Instead of creating a new branch using Git Branch and then move to that branch by using the Git Checkout command, we can directly create a new branch and move to that branch by using the **-b** flag with Git Checkout.

$ git checkout -b <new-branch-name>

By default Git Checkout with the **-b** flag will create a **new branch from the HEAD** of our current working branch. If we want to create a new branch from the commit point of some other existing branch we can pass the name of that branch.

$ git checkout -b <new-branch-name> <existing-branch-name>

We can also switch to remote branches by first fetching them and then running the git checkout command.

Summary

Branching is a very important and useful function of Git. It gives developers the freedom to try out new things without worrying about ruining previous progress. Git Branch is the function that lets us create, rename, view, and delete branches. We can use this command to start a new branch from any of the previous commit points. Git Checkout is another useful command for branching. It helps us to navigate between branches and allows us to create new branches and directly jump to them without using the Git Branch command.

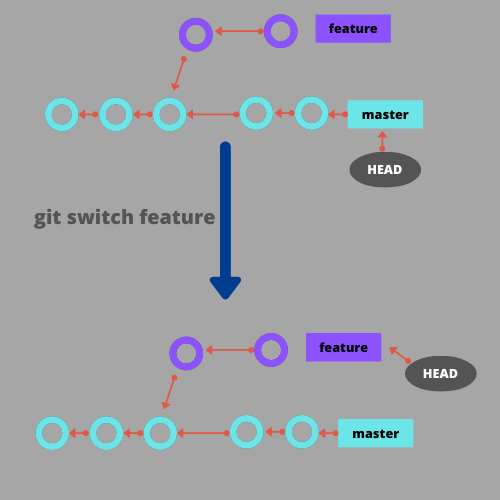
**Git Switch Branch**

When working on a project, it is good to work on separate branches to avoid corrupting the rest of the project. Git Switch is a command that helps us to create new branches and move from one branch to another. **It is very similar to Git Checkout but has lesser functionality when compared to the Checkout command**.

Switching Branches

Branching in Git is a feature that allows developers to work in an independent space without worrying about affecting the rest of the project. Moving from one branch to another is also a common operation.

* Git Checkout is the most used command for this purpose and is also more popular than Git Switch.
* Git Switch is a slightly newer command and was added in Git version 2.23.
* While Git Checkout is capable of a lot more than just switching branches, the Git Switch command was added solely to switch between branches.



Git Switch Command

The Git Switch command is used to switch between branches. There are a few additional options we can use with it for added functionalities. Let's take a look at how to use this command.

To simply switch to an existing branch, we just need the name of that branch.

$ git switch <branch-name>

We can create a new branch and switch to it by using the **-c** or the **--create**.

$ git switch -c <new-branch-name>

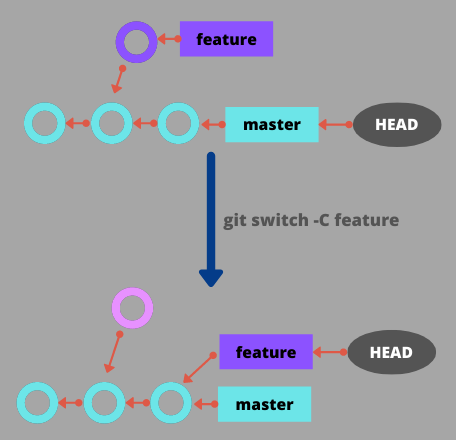
The above command will create a new branch that is based on the HEAD. To create a branch based on some other commit point we can pass that commits position relative to our current HEAD. For example, **the following command will create and switch to a branch that will be based on the n-th commit before the HEAD**.

$ git switch -c <new-branch-name> HEAD~N

Git Switch provides us a quicker way to switch the previously checked-out branch(the branch we were on before switching to our current branch). We can do this by using **-**(hyphen) instead of entering the entire branch name.

$ git switch -

We can also forcefully reset an existing branch to a new start point by using the **-C** or the **--force-create**. One thing to note here is that we will be losing the commits on that branch. Consider the following illustration in which we have already have a feature branch with one commit. Now when we use the **-C** flag to reset the feature branch then that commit will be lost because we won't have any reference to it. To get this commit back we can use the **-d** option to detach our HEAD to that commit and then create another branch based on that commit. But we must know the commit hash to be able to do that.



$ git switch -C <branch-name>

We can use the **--discard-changes** option if we want to clear our working tree and staging area of any changes before switching to another branch. Git will by default try to block a switch if it causes a conflict in the files. The **--discard-changes** are used to bypass this security feature.

Git Switch can also take us to a previous commit point with the help of **-d** or **--detach**. This will be a Detached HEAD state.

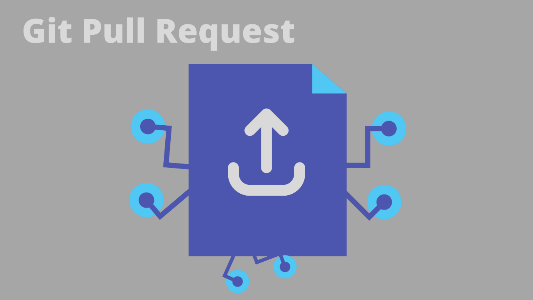
$ git switch -d <commit-hash>

Summary

Creating and Switching between branches is something that we will often do when working on any project that is version controlled. Git Switch is a command that enables us to do all this. It is quite similar to the existing Git Checkout command but is sole dedicated to switching between branches whereas the Checkout command has a few additional features like undoing and discarding changes. **Git Switch has also made it very easy to switch to the previously checked-out branch by just using the -(hyphen) instead of entering the entire branch name**. We learned about a few additional options we can use with this command that makes working with branches a lot easier.

Git Pull Request

When working in a team and collaborating on a project, we will be pushing changes to a central remote repository hosted on platforms like GitHub or Bitbucket. Pull request is a way of informing other team members about the changes that you have made. It is a way of requesting other team members to pull your changes and analyze them. Let's learn more about how pull requests work.



What are Pull Requests?

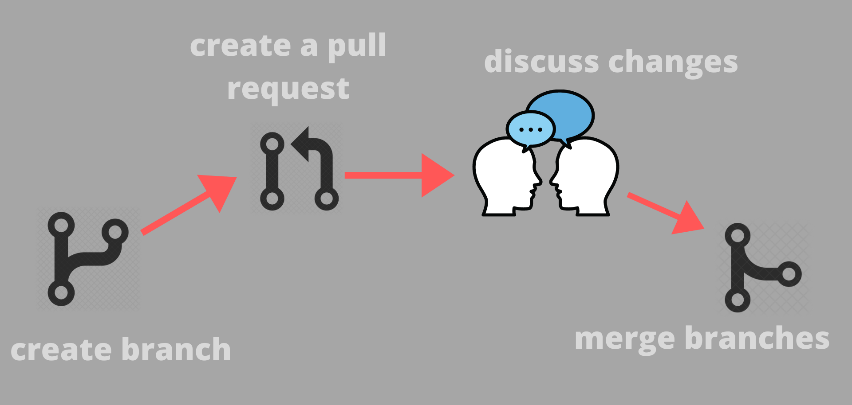
* A pull request is used to inform other team members and collaborators of the changes that you have pushed to the remote repository.
* When a pull request is created, all the developers can view and discuss your changes and can add new follow-up commits if needed. It is a convenient way of getting your code reviewed by the project manager.
* We can also push incomplete branches and start a pull request to ask for help from other developers.
* Once the changes are verified and approved by the team leader or the project manager, they can be merged with the master branch. This is why they are also known as **Merge Requests**.
* A pull request serves a three-fold purpose. **First,** it informs other developers that you have pushed changes and started a pull request. **Second**, it provides a platform to discuss the proposed changes, and **third,** it allows developers to add **follow-up commits** if they feel like something is missing.

How does a Pull Request work?

As discussed above, a pull request is made by a team member when he has pushed his/her code to the remote and wants to get it reviewed by others. A pull request can be made in two different scenarios. One, when the developer is a team member and working on a shared repository, and second when the developer wants to suggest changes to the authors of some other public repository. Let's discuss how to create a pull request in each of the above cases.

Pull Request for a Shared Repository

When working and collaborating on a shared repository, we will first clone the repository to our local system, and create a new branch for our work. We can then directly push changes to the remote and create a pull request to inform other developers. If our work is approved by the project manager then the branch that we created can be merged into the main branch of the repository.



Pull Request for a Forked Repository

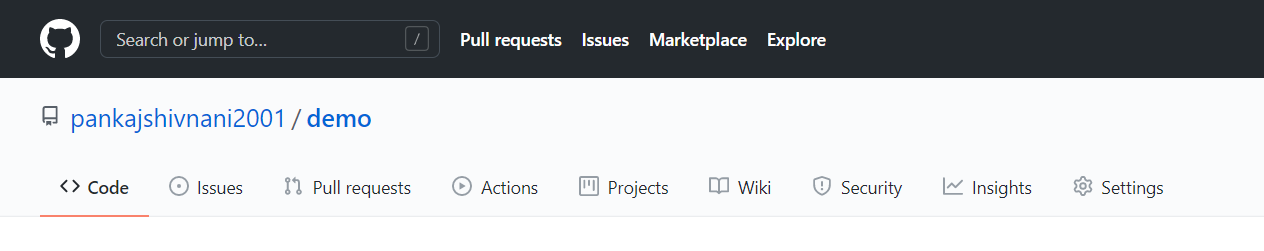
There are probably millions of public remote repositories out there. We can suggest changes to the authors of these repositories by using pull requests. First, we will fork the public repository to our GitHub account and then clone it to our local machine. We can then create new commits or new branches with the changes that we want to suggest. Then, we can push these changes to our public forked repository. Finally, we will create a pull request to let the author of the original public repository see the proposed changes. We can discuss the changes with these authors and if they like our work, they can add it to their public repository.

Pull Requests on GitHub

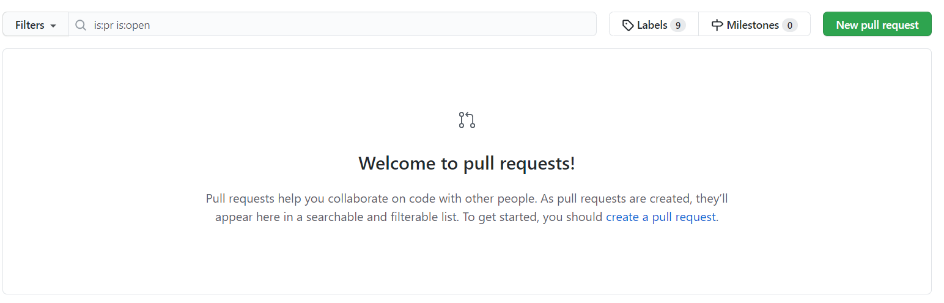
GitHub is probably the most widely used Git hosting service. Let's learn how to create pull requests on GitHub.

For Shared Repository

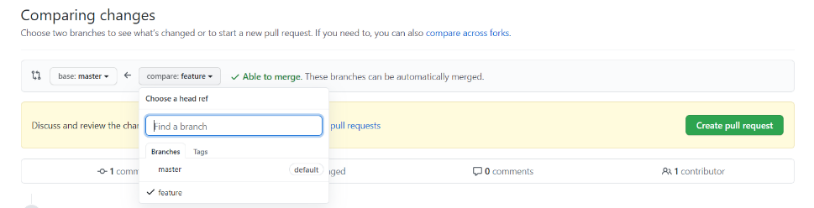
* Suppose we are working on a team project and have a shared remote repository on GitHub with just a single master branch. First, let's clone this repository to our local system. Next, we will create a new branch called feature and add some commits to this branch. Push this new feature branch to the remote repository.
* Now, head over to the GitHub repo page and click on the "pull requests" tab.



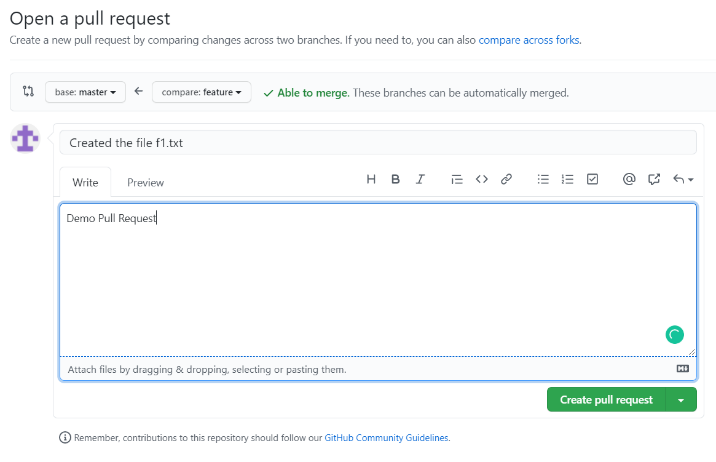
* Click on the New Pull Request button.



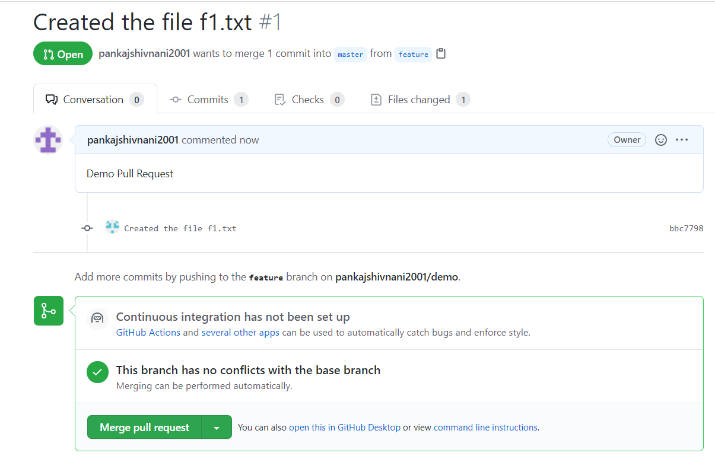
Next, we need to choose the base branch and the branch that we are trying to compare. The base branch, in most cases, will be the main branch where you want to merge the changes. The compare branch will the one that we created(feature in this case).



Click on the Create Pull Request button and add a suitable description for your feature branch.

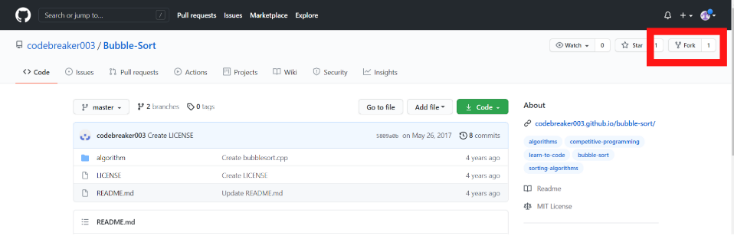


Other developers can now view and suggest changes. If everything seems fine then the project leader can merge these changes with the main branch.



For Forked Repository

First head over to the public repository to which you want to suggest changes and fork it to your GitHub account.

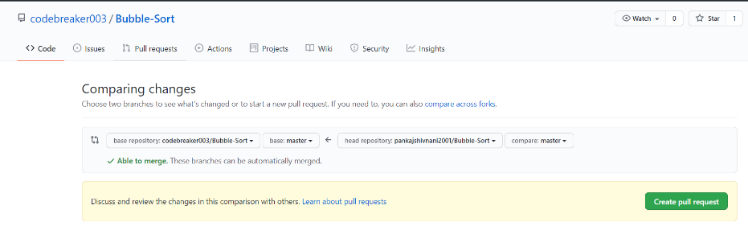


Clone this forked repository to your local system and make the commits for the changes that you want to include. Push the changes to your forked repository.

Now, when you will head over to your repository page on GitHub, you will be able to see a "compare and pull request" button. Click on it and add a suitable description and create a pull request.



You can also head over to the original public repository on GitHub and create a pull request from there. Make sure you click on the "compare across forks" links and choose the appropriate repositories and branches.



Summary

Pull requests are an efficient way of proposing changes to a repository. The collaborators can discuss these changes and add new commits to the branch. If everything works fine, then the new branch can be merged with the main workflow of the repository. We can also create pull requests for public repositories to suggest changes to other developers.

<https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging>

# **3.1 Git Branching - Branches in a Nutshell**

Nearly every VCS has some form of branching support. Branching means you diverge from the main line of development and continue to do work without messing with that main line. In many VCS tools, this is a somewhat expensive process, often requiring you to create a new copy of your source code directory, which can take a long time for large projects.

Some people refer to Git’s branching model as its “killer feature,” and it certainly sets Git apart in the VCS community. Why is it so special? The way Git branches is incredibly lightweight, making branching operations nearly instantaneous, and switching back and forth between branches generally just as fast. Unlike many other VCSs, Git encourages workflows that branch and merge often, even multiple times in a day. Understanding and mastering this feature gives you a powerful and unique tool and can entirely change the way that you develop.

## Branches in a Nutshell

To really understand the way Git does branching, we need to take a step back and examine how Git stores its data.

As you may remember from [What is Git?](https://git-scm.com/book/en/v2/ch00/what_is_git_section), Git doesn’t store data as a series of changesets or differences, but instead as a series of **snapshots**.

When you make a commit, Git stores a commit object that contains a pointer to the snapshot of the content you staged. This object also contains the author’s name and email address, the message that you typed, and pointers to the commit or commits that directly came before this commit (its parent or parents): zero parents for the initial commit, one parent for a normal commit, and multiple parents for a commit that results from a merge of two or more branches.

To visualize this, let’s assume that you have a directory containing three files, and you stage them all and commit. Staging the files computes a checksum for each one (the SHA-1 hash we mentioned in [What is Git?](https://git-scm.com/book/en/v2/ch00/what_is_git_section)), stores that version of the file in the Git repository (Git refers to them as **blobs**), and adds that checksum to the staging area:

$ git add README test.rb LICENSE  
$ git commit -m 'Initial commit'

When you create the commit by running git commit, Git checksums each subdirectory (in this case, just the root project directory) and stores them as a tree object in the Git repository. Git then creates a commit object that has the metadata and a pointer to the root project tree so it can re-create that snapshot when needed.

Your Git repository now contains **five** objects: three **blobs** (each representing the contents of one of the three files), one **tree** that lists the contents of the directory and specifies which file names are stored as which blobs, and one **commit** with the pointer to that root tree and all the commit metadata.



Figure 9. A commit and its tree

If you make some changes and commit again, the next commit stores a pointer to the commit that came immediately before it.



Figure 10. Commits and their parents

A branch in Git is simply a lightweight movable pointer to one of these commits. The default branch name in Git is master. As you start making commits, you’re given a master branch that points to the last commit you made. Every time you commit, the master branch pointer moves forward automatically.

|  |  |
| --- | --- |
| **Note** | The “master” branch in Git is not a special branch. It is exactly like any other branch. The only reason nearly every repository has one is that the git init command creates it by default and most people don’t bother to change it. |



Figure 11. A branch and its commit history

### Creating a New Branch

What happens when you create a new branch? Well, doing so creates a new pointer for you to move around. Let’s say you want to create a new branch called testing. You do this with the git branch command:

$ git branch testing

This creates a new pointer to the same commit you’re currently on.



Figure 12. Two branches pointing into the same series of commits

How does Git know what branch you’re currently on? It keeps a special pointer called HEAD. Note that this is a lot different than the concept of HEAD in other VCSs you may be used to, such as Subversion or CVS. In Git, this is a pointer to the local branch you’re currently on. In this case, you’re still on master. The git branch command only **created** a new branch — it didn’t switch to that branch.



Figure 13. HEAD pointing to a branch

You can easily see this by running a simple git log command that shows you where the branch pointers are pointing. This option is called --decorate.

$ git log --oneline –decorate  
f30ab (HEAD -> master, testing) Add feature #32 - ability to add new formats to the central interface  
34ac2 Fix bug #1328 - stack overflow under certain conditions  
98ca9 Initial commit

You can see the master and testing branches that are right there next to the f30ab commit.

### Switching Branches

To switch to an existing branch, you run the git checkout command. Let’s switch to the new testing branch:

$ git checkout testing

This moves HEAD to point to the testing branch.



Figure 14. HEAD points to the current branch

What is the significance of that? Well, let’s do another commit:

$ vim test.rb  
$ git commit -a -m 'Make a change'



Figure 15. The HEAD branch moves forward when a commit is made

This is interesting, because now your testing branch has moved forward, but your master branch still points to the commit you were on when you ran git checkout to switch branches. Let’s switch back to the master branch:

$ git checkout master

|  |  |
| --- | --- |
| **Note** | **git log doesn’t show all the branches all the time**  If you were to run git log right now, you might wonder where the "testing" branch you just created went, as it would not appear in the output.  The branch hasn’t disappeared; Git just doesn’t know that you’re interested in that branch and it is trying to show you what it thinks you’re interested in. In other words, by default, git log will only show commit history below the branch you’ve checked out.  To show commit history for the desired branch you have to explicitly specify it: git log testing. To show all of the branches, add --all to your git log command. |



Figure 16. HEAD moves when you checkout

That command did two things. It moved the HEAD pointer back to point to the master branch, and it reverted the files in your working directory back to the snapshot that master points to. This also means the changes you make from this point forward will diverge from an older version of the project. It essentially rewinds the work you’ve done in your testing branch so you can go in a different direction.

|  |  |
| --- | --- |
| **Note** | **Switching branches changes files in your working directory**  It’s important to note that when you switch branches in Git, files in your working directory will change. If you switch to an older branch, your working directory will be reverted to look like it did the last time you committed on that branch. If Git cannot do it cleanly, it will not let you switch at all. |

Let’s make a few changes and commit again:

$ vim test.rb  
$ git commit -a -m 'Make other changes'

Now your project history has diverged (see [Divergent history](https://git-scm.com/book/en/v2/ch00/divergent_history)). You created and switched to a branch, did some work on it, and then switched back to your main branch and did other work. Both of those changes are isolated in separate branches: you can switch back and forth between the branches and merge them together when you’re ready. And you did all that with simple branch, checkout, and commit commands.



Figure 17. Divergent history

You can also see this easily with the git log command. If you run git log --oneline --decorate --graph --all it will print out the history of your commits, showing where your branch pointers are and how your history has diverged.

Afbeelding met tekst, elektronica, schermopname, software

Automatisch gegenereerde beschrijving

$ git log --oneline --decorate --graph –all  
\* c2b9e (HEAD, master) Make other changes  
| \* 87ab2 (testing) Make a change  
|/  
\* f30ab Add feature #32 - ability to add new formats to the central interface  
\* 34ac2 Fix bug #1328 - stack overflow under certain conditions  
\* 98ca9 Initial commit of my project

Because a branch in Git is actually a simple file that contains the 40 character SHA-1 checksum of the commit it points to, branches are cheap to create and destroy. Creating a new branch is as quick and simple as writing 41 bytes to a file (40 characters and a newline).

This is in sharp contrast to the way most older VCS tools branch, which involves copying all of the project’s files into a second directory. This can take several seconds or even minutes, depending on the size of the project, whereas in Git the process is always instantaneous. Also, because we’re recording the parents when we commit, finding a proper merge base for merging is automatically done for us and is generally very easy to do. These features help encourage developers to create and use branches often.

Let’s see why you should do so.

|  |  |
| --- | --- |
| **Note** | **Creating a new branch and switching to it at the same time**  It’s typical to create a new branch and want to switch to that new branch at the same time — this can be done in one operation with git checkout -b <newbranchname>. |
| **Note** | From Git version 2.23 onwards you can use git switch instead of git checkout to:   * Switch to an existing branch: git switch testing-branch. * Create a new branch and switch to it: git switch -c new-branch. The -c flag stands for create, you can also use the full flag: --create. * Return to your previously checked out branch: git switch -. |

# **3.2 Git Branching - Basic Branching and Merging**

## Basic Branching and Merging

Let’s go through a simple example of branching and merging with a workflow that you might use in the real world. You’ll follow these steps:

1. Do some work on a website.
2. Create a branch for a new user story you’re working on.
3. Do some work in that branch.

At this stage, you’ll receive a call that another issue is critical and you need a hotfix. You’ll do the following:

1. Switch to your production branch.
2. Create a branch to add the hotfix.
3. After it’s tested, merge the hotfix branch, and push to production.
4. Switch back to your original user story and continue working.

### Basic Branching

First, let’s say you’re working on your project and have a couple of commits already on the master branch.



Figure 18. A simple commit history

You’ve decided that you’re going to work on issue #53 in whatever issue-tracking system your company uses. To create a new branch and switch to it at the same time, you can run the git checkout command with the -b switch:

$ git checkout -b iss53  
Switched to a new branch "iss53"

This is shorthand for:

$ git branch iss53  
$ git checkout iss53



Figure 19. Creating a new branch pointer

You work on your website and do some commits. Doing so moves the iss53 branch forward, because you have it checked out (that is, your HEAD is pointing to it):

$ vim index.html  
$ git commit -a -m 'Create new footer [issue 53]'



Figure 20. The iss53 branch has moved forward with your work

Now you get the call that there is an issue with the website, and you need to fix it immediately. With Git, you don’t have to deploy your fix along with the iss53 changes you’ve made, and you don’t have to put a lot of effort into reverting those changes before you can work on applying your fix to what is in production. All you have to do is switch back to your master branch.

However, before you do that, note that if your working directory or staging area has uncommitted changes that conflict with the branch you’re checking out, Git won’t let you switch branches. It’s best to have a clean working state when you switch branches. There are ways to get around this (namely, stashing and commit amending) that we’ll cover later on, in [Stashing and Cleaning](https://git-scm.com/book/en/v2/ch00/_git_stashing). For now, let’s assume you’ve committed all your changes, so you can switch back to your master branch:

$ git checkout master  
Switched to branch 'master'

At this point, your project working directory is exactly the way it was before you started working on issue #53, and you can concentrate on your hotfix. This is an important point to remember: when you switch branches, Git resets your working directory to look like it did the last time you committed on that branch. It adds, removes, and modifies files automatically to make sure your working copy is what the branch looked like on your last commit to it.

Next, you have a hotfix to make. Let’s create a hotfix branch on which to work until it’s completed:

$ git checkout -b hotfix  
Switched to a new branch 'hotfix'  
$ vim index.html  
$ git commit -a -m 'Fix broken email address'  
[hotfix 1fb7853] Fix broken email address  
 1 file changed, 2 insertions(+)



Figure 21. Hotfix branch based on master

You can run your tests, make sure the hotfix is what you want, and finally merge the hotfix branch back into your master branch to deploy to production. You do this with the git merge command:

$ git checkout master  
$ git merge hotfix  
Updating f42c576..3a0874c  
Fast-forward  
 index.html | 2 ++  
 1 file changed, 2 insertions(+)

You’ll notice the phrase “fast-forward” in that merge. Because the commit C4 pointed to by the branch hotfix you merged in was directly ahead of the commit C2 you’re on, Git simply moves the pointer forward. To phrase that another way, when you try to merge one commit with a commit that can be reached by following the first commit’s history, Git simplifies things by moving the pointer forward because there is no divergent work to merge together — this is called a “fast-forward.”

Your change is now in the snapshot of the commit pointed to by the master branch, and you can deploy the fix.



Figure 22. master is fast-forwarded to hotfix

After your super-important fix is deployed, you’re ready to switch back to the work you were doing before you were interrupted. However, first you’ll delete the hotfix branch, because you no longer need it — the master branch points at the same place. You can delete it with the -d option to git branch:

$ **git branch -d hotfix**  
Deleted branch hotfix (3a0874c).

Now you can switch back to your work-in-progress branch on issue #53 and continue working on it.

$ git checkout iss53  
Switched to branch "iss53"  
$ vim index.html  
$ git commit -a -m 'Finish the new footer [issue 53]'  
[iss53 ad82d7a] Finish the new footer [issue 53]  
1 file changed, 1 insertion(+)



Figure 23. Work continues on iss53

It’s worth noting here that the work you did in your hotfix branch is not contained in the files in your iss53 branch. If you need to pull it in, you can merge your master branch into your iss53 branch by running git merge master, or you can wait to integrate those changes until you decide to pull the iss53 branch back into master later.

### Basic Merging

Suppose you’ve decided that your issue #53 work is complete and ready to be merged into your master branch. In order to do that, you’ll merge your iss53 branch into master, much like you merged your hotfix branch earlier. All you have to do is check out the branch you wish to merge into and then run the git merge command:

$ git checkout master  
Switched to branch 'master'  
$ git merge iss53  
Merge made by the 'recursive' strategy.  
index.html | 1 +  
1 file changed, 1 insertion(+)

This looks a bit different than the hotfix merge you did earlier. In this case, your development history has diverged from some older point. Because the commit on the branch you’re on isn’t a direct ancestor of the branch you’re merging in, Git has to do some work. In this case, Git does a simple three-way merge, using the two snapshots pointed to by the branch tips and the common ancestor of the two.



Figure 24. Three snapshots used in a typical merge

Instead of just moving the branch pointer forward, Git creates a new snapshot that results from this three-way merge and automatically creates a new commit that points to it. This is referred to as a merge commit, and is special in that it has more than one parent.



Figure 25. A merge commit

Now that your work is merged in, you have no further need for the iss53 branch. You can close the issue in your issue-tracking system, and delete the branch:

$ git branch -d iss53

### Basic Merge Conflicts

Occasionally, this process doesn’t go smoothly. If you changed the same part of the same file differently in the two branches you’re merging, Git won’t be able to merge them cleanly. If your fix for issue #53 modified the same part of a file as the hotfix branch, you’ll get a merge conflict that looks something like this:

$ git merge iss53  
Auto-merging index.html  
CONFLICT (content): Merge conflict in index.html  
Automatic merge failed; fix conflicts and then commit the result.

Git hasn’t automatically created a new merge commit. It has paused the process while you resolve the conflict. If you want to see which files are unmerged at any point after a merge conflict, you can run git status:

$ git status  
On branch master  
You have unmerged paths.  
 (fix conflicts and run "git commit")  
Unmerged paths:  
 (use "git add <file>..." to mark resolution)  
 both modified: index.html  
no changes added to commit (use "git add" and/or "git commit -a")

Anything that has merge conflicts and hasn’t been resolved is listed as unmerged. Git adds standard conflict-resolution markers to the files that have conflicts, so you can open them manually and resolve those conflicts. Your file contains a section that looks something like this:

<<<<<<< HEAD:index.html  
<div id="footer">contact : email.support@github.com</div>  
=======  
<div id="footer">  
 please contact us at [support@github.com](mailto:support@github.com)  
</div>  
>>>>>>> iss53:index.html

This means the version in HEAD (your master branch, because that was what you had checked out when you ran your merge command) is the top part of that block (everything above the =======), while the version in your iss53 branch looks like everything in the bottom part. In order to resolve the conflict, you have to either choose one side or the other or merge the contents yourself. For instance, you might resolve this conflict by replacing the entire block with this:

<div id="footer">  
please contact us at [email.support@github.com](mailto:email.support@github.com)  
</div>

This resolution has a little of each section, and the <<<<<<<, =======, and >>>>>>> lines have been completely removed. After you’ve resolved each of these sections in each conflicted file, run git add on each file to mark it as resolved. Staging the file marks it as resolved in Git.

If you want to use a graphical tool to resolve these issues, you can run git mergetool, which fires up an appropriate visual merge tool and walks you through the conflicts:

$ git mergetool  
This message is displayed because 'merge.tool' is not configured.  
See 'git mergetool --tool-help' or 'git help config' for more details.  
'git mergetool' will now attempt to use one of the following tools:  
opendiff kdiff3 tkdiff xxdiff meld tortoisemerge gvimdiff diffuse diffmerge ecmerge p4merge araxis bc3 codecompare vimdiff emerge  
Merging:  
index.html  
Normal merge conflict for 'index.html':  
 {local}: modified file  
 {remote}: modified file  
Hit return to start merge resolution tool (opendiff):

If you want to use a merge tool other than the default (Git chose opendiff in this case because the command was run on macOS), you can see all the supported tools listed at the top after “one of the following tools.” Just type the name of the tool you’d rather use.

|  |  |
| --- | --- |
| **Note** | If you need more advanced tools for resolving tricky merge conflicts, we cover more on merging in [Advanced Merging](https://git-scm.com/book/en/v2/ch00/_advanced_merging). |

After you exit the merge tool, Git asks you if the merge was successful. If you tell the script that it was, it stages the file to mark it as resolved for you. You can run git status again to verify that all conflicts have been resolved:

$ git status  
On branch master  
All conflicts fixed but you are still merging.  
 (use "git commit" to conclude merge)  
Changes to be committed:  
 modified: index.html

If you’re happy with that, and you verify that everything that had conflicts has been staged, you can type git commit to finalize the merge commit. The commit message by default looks something like this:

Merge branch 'iss53'  
Conflicts:  
 index.html  
# It looks like you may be committing a merge.  
# If this is not correct, please remove the file  
# .git/MERGE\_HEAD  
# and try again.  
# Please enter the commit message for your changes. Lines starting  
# with '#' will be ignored, and an empty message aborts the commit.  
# On branch master  
# All conflicts fixed but you are still merging.  
# Changes to be committed:  
# modified: index.html#

If you think it would be helpful to others looking at this merge in the future, you can modify this commit message with details about how you resolved the merge and explain why you did the changes you made if these are not obvious.