git syncing

[**git remote**](https://www.atlassian.com/git/tutorials/syncing) [**git fetch**](https://www.atlassian.com/git/tutorials/syncing) [**git push**](https://www.atlassian.com/git/tutorials/syncing) [**git pull**](https://www.atlassian.com/git/tutorials/syncing)

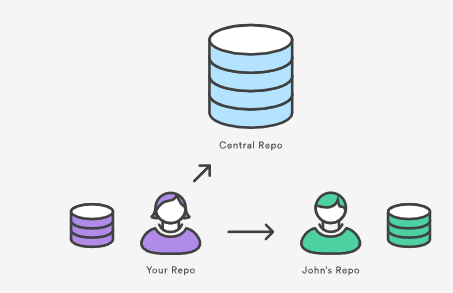
SVN uses a single centralized repository to serve as the communication hub for developers, and collaboration takes place by passing changesets between the developers’ working copies and the central repository. This is different from Git's distributed collaboration model, which gives every developer their own copy of the repository, complete with its own local history and branch structure. Users typically need to share a series of commits rather than a single changeset. Instead of committing a changeset from a working copy to the central repository, Git lets you share entire branches between repositories.

The git remote command is one piece of the broader system which is responsible for syncing changes. Records registered through the git remote command are used in conjunction with the [git fetch](https://www.atlassian.com/git/tutorials/syncing/git-fetch), [git push](https://www.atlassian.com/git/tutorials/syncing/git-push), and [git pull](https://www.atlassian.com/git/tutorials/syncing/git-pull) commands. These commands all have their own syncing responsibilities which can be explored on the corresponding links.

Git remote

The git remote command lets you create, view, and delete connections to other repositories. Remote connections are more like bookmarks rather than direct links into other repositories. Instead of providing real-time access to another repository, they serve as convenient names that can be used to reference a not-so-convenient URL.

For example, the following diagram shows two remote connections from your repo into the central repo and another developer’s repo. Instead of referencing them by their full URLs, you can pass the origin and john shortcuts to other Git commands.



Git remote usage overview

The git remote command is essentially an interface for managing a list of remote entries that are stored in the repository's ./.git/config file. The following commands are used to view the current state of the remote list.

Viewing git remote configurations

git remote

List the remote connections you have to other repositories.

git remote -v

Same as the above command, but include the URL of each connection.

Creating and modifying git remote configurations

The git remote command is also a convenience or 'helper' method for modifying a repo's ./.git/config file. The commands presented below let you manage connections with other repositories. The following commands will modify the repo's /.git/config file. The result of the following commands can also be achieved by directly editing the ./.git/config file with a text editor.

git remote add <name> <url>

Create a new connection to a remote repository. After adding a remote, you’ll be able to use ＜name＞ as a convenient shortcut for ＜url＞ in other Git commands.

git remote rm <name>

Remove the connection to the remote repository called ＜name＞.

git remote rename <old-name> <new-name>

Rename a remote connection from ＜old-name＞ to ＜new-name＞.

Git remote discussion

Git is designed to give each developer an entirely isolated development environment. This means that information is not automatically passed back and forth between repositories. Instead, developers need to manually pull upstream commits into their local repository or manually push their local commits back up to the central repository. The git remote command is really just an easier way to pass URLs to these "sharing" commands.

The origin Remote

When you clone a repository with git clone, it automatically creates a remote connection called origin pointing back to the cloned repository. This is useful for developers creating a local copy of a central repository, since it provides an easy way to pull upstream changes or publish local commits. This behavior is also why most Git-based projects call their central repository origin.

Repository URLs

Git supports many ways to reference a remote repository. Two of the easiest ways to access a remote repo are via the HTTP and the SSH protocols. HTTP is an easy way to allow anonymous, read-only access to a repository. For example:

http://host/path/to/repo.git

But, it’s generally not possible to push commits to an HTTP address (you wouldn’t want to allow anonymous pushes anyways). For read-write access, you should use SSH instead:

ssh://user@host/path/to/repo.git

You’ll need a valid SSH account on the host machine, but other than that, Git supports authenticated access via SSH out of the box. Modern secure 3rd party hosting solutions like Bitbucket.com will provide these URLs for you.

Git remote commands

The git remote command is one of many Git commands that takes additional appended 'subcommands'. Below is an examination of the commonly used git remote subcommands.

ADD <NAME> <URL>

Adds a record to ./.git/config for remote named ＜name＞ at the repository url ＜url＞.

Accepts a -f option, that will git fetch immediately after the remote record is created.

Accepts a --tags option, that will git fetch immediately and import every tag from the remote repository.

RENAME <OLD> <NEW>

Updates ./.git/config to rename the record ＜OLD＞ to ＜NEW＞. All remote-tracking branches and configuration settings for the remote are updated.

REMOVE or RM <NAME>

Modifies ./.git/config and removes the remote named ＜NAME＞. All remote-tracking branches and configuration settings for the remote are removed.

GET-URL <NAME>

Outputs the URLs for a remote record.

Accepts --push, push URLs are queried rather than fetch URLs.

With --all, all URLs for the remote will be listed.

SHOW <NAME>

Outputs high-level information about the remote ＜NAME＞.

PRUNE <NAME>

Deletes any local branches for ＜NAME＞ that are not present on the remote repository.

Accepts a --dry-run option which will list what branches are set to be pruned, but will not actually prune them.

Git remote examples

In addition to origin, it’s often convenient to have a connection to your teammates’ repositories. For example, if your co-worker, John, maintained a publicly accessible repository on dev.example.com/john.git, you could add a connection as follows:

git remote add john http://dev.example.com/john.git

Having this kind of access to individual developers’ repositories makes it possible to collaborate outside of the central repository. This can be very useful for small teams working on a large project.

Showing your remotes

By default, the git remote command will list previously stored remote connections to other repositories. This will produce single line output that lists the names of "bookmark" name of remote repos.

$ git remote  
origin  
upstream  
other\_users\_repo

Invoking git remote with the -v option will print the list of bookmarked repository names and additionally, the corresponding repository URL. The -v option stands for "verbose". Below is example output of verbose git remote output.

git remote -v  
origin  git@bitbucket.com:origin\_user/reponame.git (fetch)  
origin  git@bitbucket.com:origin\_user/reponame.git (push)  
upstream    https://bitbucket.com/upstream\_user/reponame.git (fetch)  
upstream    https://bitbucket.com/upstream\_user/reponame.git (push)  
other\_users\_repo    https://bitbucket.com/other\_users\_repo/reponame (fetch)  
other\_users\_repo    https://bitbucket.com/other\_users\_repo/reponame (push)

Adding Remote Repositories

The git remote add command will create a new connection record to a remote repository. After adding a remote, you’ll be able to use as a convenient shortcut for in other Git commands. For more information on the accepted URL syntax, view the "Repository URLs" section below. This command will create a new record within the repository's ./.git/config. An example of this config file update follows:

$ git remote add fake\_test https://bitbucket.com/upstream\_user/reponame.git; [remote "remote\_test"]   
   url = https://bitbucket.com/upstream\_user/reponame.git   
   fetch = +refs/heads/\*:refs/remotes/remote\_test/\*

Inspecting a Remote

The show subcommand can be appended to git remote to give detailed output on the configuration of a remote. This output will contain a list of branches associated with the remote and also the endpoints attached for fetching and pushing.

git remote show upstream  
\* remote upstream  
   Fetch URL: https://bitbucket.com/upstream\_user/reponame.git  
   Push URL: https://bitbucket.com/upstream\_user/reponame.git  
   HEAD branch: main  
   Remote branches:  
      main tracked  
      simd-deprecated tracked  
      tutorial tracked  
   Local ref configured for 'git push':  
      main pushes to main (fast-forwardable)

Fetching and pulling from Git remotes

Once a remote record has been configured through the use of the git remote command, the remote name can be passed as an argument to other Git commands to communicate with the remote repo. Both [git fetch](https://www.atlassian.com/git/tutorials/syncing/git-fetch), and [git pull](https://www.atlassian.com/git/tutorials/syncing/git-pull) can be used to read from a remote repository. Both commands have different operations that are explained in further depth on their respective links.

Pushing to Git remotes

The git push command is used to write to a remote repository.

git push <remote-name> <branch-name>

This example will upload the local state of ＜branch-name＞ to the remote repository specified by ＜remote-name＞.

Renaming and Removing Remotes

git remote rename <old-name> <new-name>

The command git remote rename is self-explanatory. When executed, this command will rename a remote connection from ＜old-name＞ to ＜new-name＞. Additionally, this will modify the contents of ./.git/config to rename the record for the remote there as well.

git remote rm <name>

The command git remote rm will remove the connection to the remote repository specified by the ＜name＞ parameter. To demonstrate let us 'undo' the remote addition from our last example. If we execute git remote rm remote\_test, and then examine the contents of ./.git/config we can see that the [remote "remote\_test"] record is no longer there.

git fetch

[**git remote**](https://www.atlassian.com/git/tutorials/syncing/git-fetch) [**git fetch**](https://www.atlassian.com/git/tutorials/syncing/git-fetch) [**git push**](https://www.atlassian.com/git/tutorials/syncing/git-fetch) [**git pull**](https://www.atlassian.com/git/tutorials/syncing/git-fetch)

The git fetch command downloads commits, files, and refs from a remote repository into your local repo. Fetching is what you do when you want to see what everybody else has been working on. It’s similar to svn update in that it lets you see how the central history has progressed, but it doesn’t force you to actually merge the changes into your repository. Git isolates fetched content from existing local content; it has absolutely no effect on your local development work. Fetched content has to be explicitly checked out using the [git checkout](https://www.atlassian.com/git/tutorials/using-branches/git-checkout) command. This makes fetching a safe way to review commits before integrating them with your local repository.

When downloading content from a remote repo, git pull and git fetch commands are available to accomplish the task. You can consider git fetch the 'safe' version of the two commands. It will download the remote content but not update your local repo's working state, leaving your current work intact. git pull is the more aggressive alternative; it will download the remote content for the active local branch and immediately execute git merge to create a merge commit for the new remote content. If you have pending changes in progress this will cause conflicts and kick-off the merge conflict resolution flow.

How git fetch works with remote branches

To better understand how git fetch works let us discuss how Git organizes and stores commits. Behind the scenes, in the repository's ./.git/objects directory, Git stores all commits, local and remote. Git keeps remote and local branch commits distinctly separate through the use of branch refs. The refs for local branches are stored in the ./.git/refs/heads/. Executing the [git branch](https://www.atlassian.com/git/tutorials/using-branches) command will output a list of the local branch refs. The following is an example of git branch output with some demo branch names.

git branch  
main  
feature1  
debug2

Examining the contents of the /.git/refs/heads/ directory would reveal similar output.

ls ./.git/refs/heads/  
main  
feature1  
debug2

Remote branches are just like local branches, except they map to commits from somebody else’s repository. Remote branches are prefixed by the remote they belong to so that you don’t mix them up with local branches. Like local branches, Git also has refs for remote branches. Remote branch refs live in the ./.git/refs/remotes/ directory. The next example code snippet shows the branches you might see after fetching a remote repo conveniently named remote-repo:

git branch -r  
# origin/main  
# origin/feature1  
# origin/debug2  
# remote-repo/main  
# remote-repo/other-feature

This output displays the local branches we had previously examined but now displays them prefixed with origin/. Additionally, we now see the remote branches prefixed with remote-repo. You can check out a remote branch just like a local one, but this puts you in a detached HEAD state (just like checking out an old commit). You can think of them as read-only branches. To view your remote branches, simply pass the -r flag to the git branch command.

You can inspect remote branches with the usual git checkout and git log commands. If you approve the changes a remote branch contains, you can merge it into a local branch with a normal git merge. So, unlike SVN, synchronizing your local repository with a remote repository is actually a two-step process: fetch, then merge. The git pull command is a convenient shortcut for this process.

Git fetch commands and options

git fetch <remote>

Fetch all of the branches from the repository. This also downloads all of the required commits and files from the other repository.

git fetch <remote> <branch>

Same as the above command, but only fetch the specified branch.

git fetch --all

A power move which fetches all registered remotes and their branches:

git fetch --dry-run

The --dry-run option will perform a demo run of the command. It will output examples of actions it will take during the fetch but not apply them.

Git fetch examples

git fetch a remote branch

The following example will demonstrate how to fetch a remote branch and update your local working state to the remote contents. In this example, let us assume there is a central repo origin from which the local repository has been cloned from using the git clone command. Let us also assume an additional remote repository named coworkers\_repo that contains a feature\_branch which we will configure and fetch. With these assumptions set let us continue the example.

Firstly we will need to configure the remote repo using the [git remote](https://www.atlassian.com/git/tutorials/syncing) command.

git remote add coworkers\_repo git@bitbucket.org:coworker/coworkers\_repo.git

Here we have created a reference to the coworker's repo using the repo URL. We will now pass that remote name to git fetch to download the contents.

git fetch coworkers\_repo coworkers/feature\_branch  
fetching coworkers/feature\_branch

We now locally have the contents of coworkers/feature\_branch we will need the integrate this into our local working copy. We begin this process by using the [git checkout](https://www.atlassian.com/git/tutorials/using-branches/git-checkout) command to checkout the newly downloaded remote branch.

git checkout coworkers/feature\_branch  
Note: checking out coworkers/feature\_branch'.  
  
You are in 'detached HEAD' state. You can look around, make experimental  
changes and commit them, and you can discard any commits you make in this  
state without impacting any branches by performing another checkout.  
  
If you want to create a new branch to retain commits you create, you may  
do so (now or later) by using -b with the checkout command again. Example:  
  
git checkout -b <new-branch-name>

The output from this checkout operation indicates that we are in a detached HEAD state. This is expected and means that our HEAD ref is pointing to a ref that is not in sequence with our local history. Being that HEAD is pointed at the coworkers/feature\_branch ref, we can create a new local branch from that ref. The 'detached HEAD' output shows us how to do this using the git checkout command:

git checkout -b local\_feature\_branch

Here we have created a new local branch named local\_feature\_branch. This puts updates HEAD to point at the latest remote content and we can continue development on it from this point.

Synchronize origin with git fetch

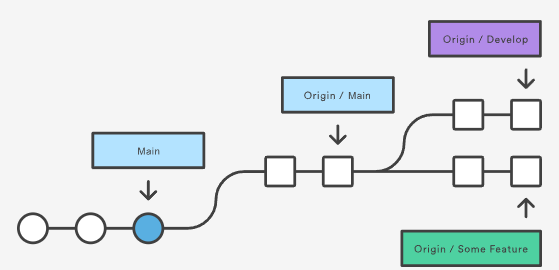
The following example walks through the typical workflow for synchronizing your local repository with the central repository's main branch.

git fetch origin

This will display the branches that were downloaded:

a1e8fb5..45e66a4 main -> origin/main  
a1e8fb5..9e8ab1c develop -> origin/develop  
\* [new branch] some-feature -> origin/some-feature

The commits from these new remote branches are shown as squares instead of circles in the diagram below. As you can see, git fetch gives you access to the entire branch structure of another repository.



To see what commits have been added to the upstream main, you can run a git log using origin/main as a filter:

git log --oneline main..origin/main

To approve the changes and merge them into your local main branch use the following commands:

git checkout main  
git log origin/main

Then we can use git merge origin/main:

git merge origin/main

The origin/main and main branches now point to the same commit, and you are synchronized with the upstream developments.

Git fetch summary

In review, git fetch is a primary command used to download contents from a remote repository. git fetch is used in conjunction with git remote, git branch, git checkout, and [git reset](https://www.atlassian.com/git/tutorials/undoing-changes/git-reset) to update a local repository to the state of a remote. The git fetch command is a critical piece of collaborative git work flows. git fetch has similar behavior to git pull, however, git fetch can be considered a safer, nondestructive version.

git push

[**git remote**](https://www.atlassian.com/git/tutorials/syncing/git-push) [**git fetch**](https://www.atlassian.com/git/tutorials/syncing/git-push) [**git push**](https://www.atlassian.com/git/tutorials/syncing/git-push) [**git pull**](https://www.atlassian.com/git/tutorials/syncing/git-push)

The git push command is used to upload local repository content to a remote repository. Pushing is how you transfer commits from your local repository to a remote repo. It's the counterpart to [git fetch](https://www.atlassian.com/git/tutorials/syncing/git-fetch), but whereas fetching imports commits to local branches, pushing exports commits to remote branches. Remote branches are configured using the [git remote](https://www.atlassian.com/git/tutorials/syncing) command. Pushing has the potential to overwrite changes, caution should be taken when pushing. These issues are discussed below.

Git push usage

git push <remote> <branch>

Push the specified branch to , along with all of the necessary commits and internal objects. This creates a local branch in the destination repository. To prevent you from overwriting commits, Git won’t let you push when it results in a non-fast-forward merge in the destination repository.

git push <remote> --force

Same as the above command, but force the push even if it results in a non-fast-forward merge. Do not use the --force flag unless you’re absolutely sure you know what you’re doing.

git push <remote> --all

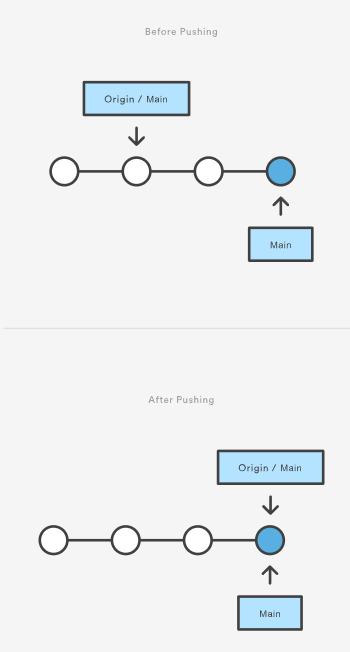
Push all of your local branches to the specified remote.

git push <remote> --tags

Tags are not automatically pushed when you push a branch or use the --all option. The --tags flag sends all of your local tags to the remote repository.

Git push discussion

git push is most commonly used to publish an upload local changes to a central repository. After a local repository has been modified a push is executed to share the modifications with remote team members.



The above diagram shows what happens when your local main has progressed past the central repository’s main and you publish changes by running git push origin main. Notice how git push is essentially the same as running git merge main from inside the remote repository.

Git push and syncing

git push is one component of many used in the overall Git "syncing" process. The syncing commands operate on remote branches which are configured using the [git remote](https://www.atlassian.com/git/tutorials/syncing) command. git push can be considered and 'upload' command whereas, [git fetch](https://www.atlassian.com/git/tutorials/syncing/git-fetch) and [git pull](https://www.atlassian.com/git/tutorials/syncing/git-pull) can be thought of as 'download' commands. Once changesets have been moved via a download or upload a [git merge](https://www.atlassian.com/git/tutorials/using-branches/git-merge) may be performed at the destination to integrate the changes.

Pushing to bare repositories

A frequently used, modern Git practice is to have a remotely hosted --bare repository act as a central origin repository. This origin repository is often hosted off-site with a trusted 3rd party like Bitbucket. Since pushing messes with the remote branch structure, It is safest and most common to push to repositories that have been created with the --bare flag. Bare repos don’t have a working directory so a push will not alter any in progress working directory content. For more information on bare repository creation, read about [git init](https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-init).

Force Pushing

Git prevents you from overwriting the central repository’s history by refusing push requests when they result in a non-fast-forward merge. So, if the remote history has diverged from your history, you need to pull the remote branch and merge it into your local one, then try pushing again. This is similar to how SVN makes you synchronize with the central repository via svn update before committing a changeset.

The --force flag overrides this behavior and makes the remote repository’s branch match your local one, deleting any upstream changes that may have occurred since you last pulled. The only time you should ever need to force push is when you realize that the commits you just shared were not quite right and you fixed them with a git commit --amend or an interactive rebase. However, you must be absolutely certain that none of your teammates have pulled those commits before using the --force option.

Examples

Default git push

The following example describes one of the standard methods for publishing local contributions to the central repository. First, it makes sure your local main is up-to-date by fetching the central repository’s copy and rebasing your changes on top of them. The interactive rebase is also a good opportunity to clean up your commits before sharing them. Then, the git push command sends all of the commits on your local main to the central repository.

git checkout main  
git fetch origin main  
git rebase -i origin/main  
# Squash commits, fix up commit messages etc.  
git push origin main

Since we already made sure the local main was up-to-date, this should result in a fast-forward merge, and git push should not complain about any of the non-fast-forward issues discussed above.

Amended force push

The [git commit](https://www.atlassian.com/git/tutorials/saving-changes/git-commit) command accepts a --amend option which will update the previous commit. A commit is often amended to update the commit message or add new changes. Once a commit is amended a git push will fail because Git will see the amended commit and the remote commit as diverged content. The --force option must be used to push an amended commit.

# make changes to a repo and git add  
git commit --amend  
# update the existing commit message  
git push --force origin main

The above example assumes it is being executed on an existing repository with a commit history. git commit --amend is used to update the previous commit. The amended commit is then force pushed using the --force option.

Deleting a remote branch or tag

Sometimes branches need to be cleaned up for book keeping or organizational purposes. The fully delete a branch, it must be deleted locally and also remotely.

git branch -D branch\_name  
git push origin :branch\_name

The above will delete the remote branch named branch\_name passing a branch name prefixed with a colon to git push will delete the remote branch.

git pull

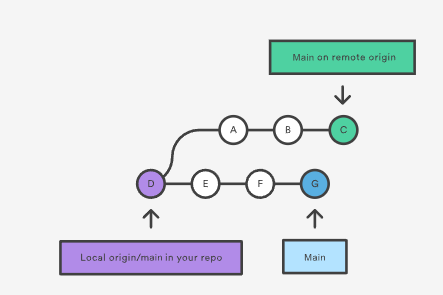
[**git remote**](https://www.atlassian.com/git/tutorials/syncing/git-pull) [**git fetch**](https://www.atlassian.com/git/tutorials/syncing/git-pull) [**git push**](https://www.atlassian.com/git/tutorials/syncing/git-pull) [**git pull**](https://www.atlassian.com/git/tutorials/syncing/git-pull)

The git pull command is used to fetch and download content from a remote repository and immediately update the local repository to match that content. Merging remote upstream changes into your local repository is a common task in Git-based collaboration work flows. The git pull command is actually a combination of two other commands, [git fetch](https://www.atlassian.com/git/tutorials/syncing/git-fetch) followed by [git merge](https://www.atlassian.com/git/tutorials/using-branches/git-merge). In the first stage of operation git pull will execute a git fetch scoped to the local branch that HEAD is pointed at. Once the content is downloaded, git pull will enter a merge workflow. A new merge commit will be-created and HEAD updated to point at the new commit.

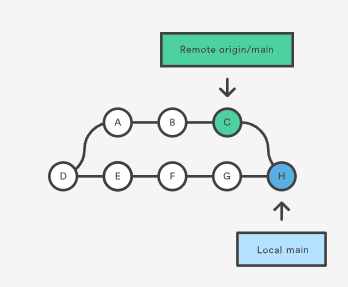
Git pull usage

How it works

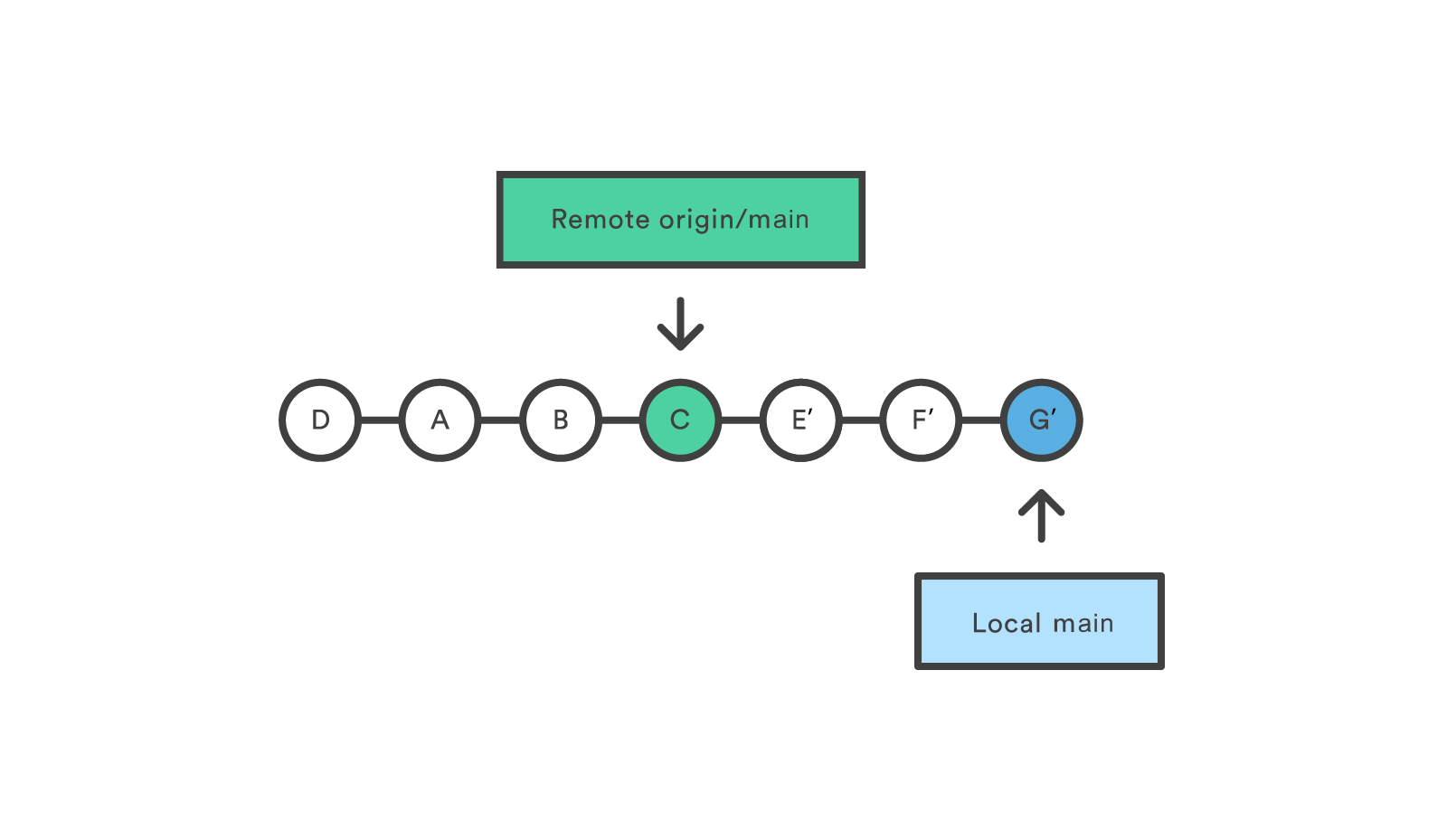
The git pull command first runs git fetch which downloads content from the specified remote repository. Then a git merge is executed to merge the remote content refs and heads into a new local merge commit. To better demonstrate the pull and merging process let us consider the following example. Assume we have a repository with a main branch and a remote origin.



In this scenario, git pull will download all the changes from the point where the local and main diverged. In this example, that point is E. git pull will fetch the diverged remote commits which are A-B-C. The pull process will then create a new local merge commit containing the content of the new diverged remote commits.



In the above diagram, we can see the new commit H. This commit is a new merge commit that contains the contents of remote A-B-C commits and has a combined log message. This example is one of a few git pull merging strategies. A --rebase option can be passed to git pull to use a rebase merging strategy instead of a merge commit. The next example will demonstrate how a rebase pull works. Assume that we are at a starting point of our first diagram, and we have executed git pull --rebase.



In this diagram, we can now see that a rebase pull does not create the new H commit. Instead, the rebase has copied the remote commits A--B--C and rewritten the local commits E--F--G to appear after them them in the local origin/main commit history.

Common Options

git pull <remote>

Fetch the specified remote’s copy of the current branch and immediately merge it into the local copy. This is the same as git fetch ＜remote＞ followed by git merge origin/＜current-branch＞.

git pull --no-commit <remote>

Similar to the default invocation, fetches the remote content but does not create a new merge commit.

git pull --rebase <remote>

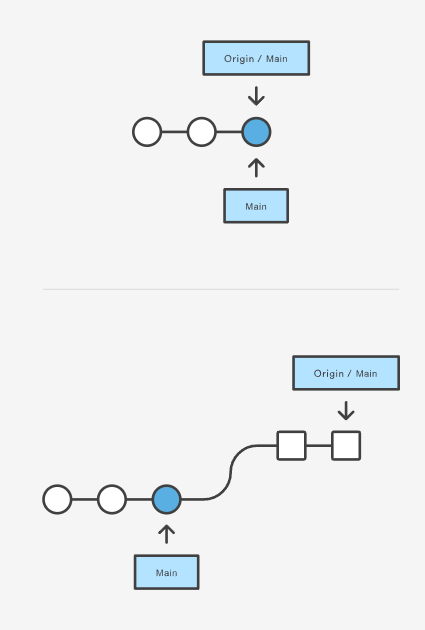
Same as the previous pull Instead of using git merge to integrate the remote branch with the local one, use [git rebase](https://www.atlassian.com/git/tutorials/rewriting-history/git-rebase).

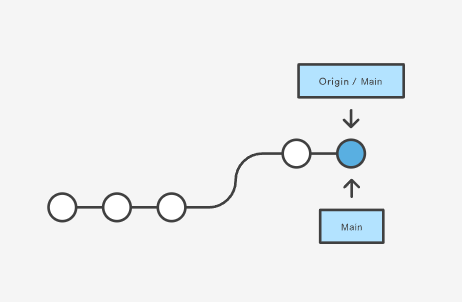
git pull --verbose

Gives verbose output during a pull which displays the content being downloaded and the merge details.

Git pull discussion

You can think of git pull as Git's version of svn update. It’s an easy way to synchronize your local repository with upstream changes. The following diagram explains each step of the pulling process.





You start out thinking your repository is synchronized, but then git fetch reveals that origin's version of main has progressed since you last checked it. Then git merge immediately integrates the remote main into the local one.

Git pull and syncing

git pull is one of many commands that claim the responsibility of 'syncing' remote content. The [git remote](https://www.atlassian.com/git/tutorials/syncing) command is used to specify what remote endpoints the syncing commands will operate on. The [git push](https://www.atlassian.com/git/tutorials/syncing/git-push) command is used to upload content to a remote repository.

The git fetch command can be confused with git pull. They are both used to download remote content. An important safety distinction can be made between git pull and get fetch. git fetch can be considered the "safe" option whereas, git pull can be considered unsafe. git fetch will download the remote content and not alter the state of the local repository. Alternatively, git pull will download remote content and immediately attempt to change the local state to match that content. This may unintentionally cause the local repository to get in a conflicted state.

Pulling via Rebase

The --rebase option can be used to ensure a linear history by preventing unnecessary merge commits. Many developers prefer rebasing over merging, since it’s like saying, "I want to put my changes on top of what everybody else has done." In this sense, using git pull with the --rebase flag is even more like svn update than a plain git pull.

In fact, pulling with --rebase is such a common workflow that there is a dedicated configuration option for it:

git config --global branch.autosetuprebase always

After running that command, all git pull commands will integrate via git rebase instead of git merge.

Git Pull Examples

The following examples demonstrate how to use git pull in common scenarios:

Default Behavior

git pull

Executing the default invocation of git pull will is equivalent to git fetch origin HEAD and git merge HEAD where HEAD is ref pointing to the current branch.

Git pull on remotes

git checkout new\_feature  
git pull <remote repo>

This example first performs a checkout and switches to the branch. Following that, the git pull is executed with being passed. This will implicitly pull down the newfeature branch from . Once the download is complete it will initiate a git merge.

Git pull rebase instead of merge

The following example demonstrates how to synchronize with the central repository's main branch using a rebase:

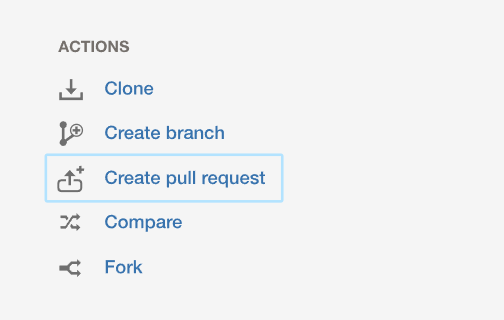
git checkout main  
git pull --rebase origin

This simply moves your local changes onto the top of what everybody else has already contributed.

# Making a Pull Request

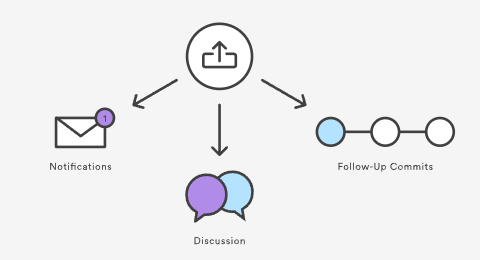
[**How it works**](https://www.atlassian.com/git/tutorials/making-a-pull-request#how-it-works) [**Example**](https://www.atlassian.com/git/tutorials/making-a-pull-request#example) [**Where to go from here**](https://www.atlassian.com/git/tutorials/making-a-pull-request#where-to-go)

Pull requests are a feature that makes it easier for developers to collaborate using [Bitbucket](http://www.bitbucket.org/). They provide a user-friendly web interface for discussing proposed changes before integrating them into the official project.



In their simplest form, pull requests are a mechanism for a developer to notify team members that they have completed a feature. Once their feature branch is ready, the developer files a pull request via their Bitbucket account. This lets everybody involved know that they need to review the code and merge it into the main branch.

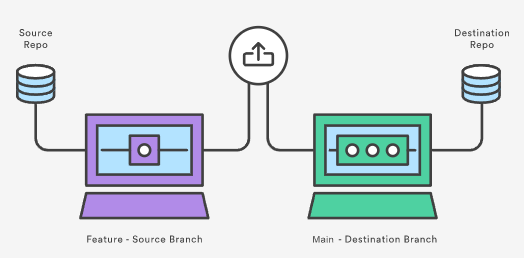
But, the pull request is more than just a notification—it’s a dedicated forum for discussing the proposed feature. If there are any problems with the changes, teammates can post feedback in the pull request and even tweak the feature by pushing follow-up commits. All of this activity is tracked directly inside of the pull request.



Compared to other collaboration models, this formal solution for sharing commits makes for a much more streamlined workflow. SVN and Git can both automatically send notification emails with a simple script; however, when it comes to discussing changes, developers typically have to rely on email threads. This can become haphazard, especially when follow-up commits are involved. Pull requests put all of this functionality into a friendly web interface right next to your Bitbucket repositories.

### Anatomy of a Pull Request

When you file a pull request, all you’re doing is requesting that another developer (e.g., the project maintainer) pulls a branch from your repository into their repository. This means that you need to provide 4 pieces of information to file a pull request: the source repository, the source branch, the destination repository, and the destination branch.



Many of these values will be set to a sensible default by Bitbucket. However, depending on your collaboration workflow, your team may need to specify different values. The above diagram shows a pull request that asks to merge a feature branch into the official main branch, but there are many other ways to use pull requests.

## How it works

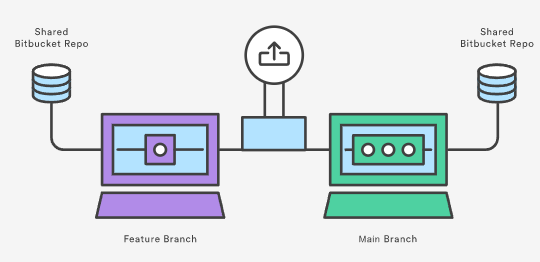
Pull requests can be used in conjunction with the [Feature Branch Workflow](https://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow), the [Gitflow Workflow](https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow), or the [Forking Workflow](https://www.atlassian.com/git/tutorials/comparing-workflows/forking-workflow). But a pull request requires either two distinct branches or two distinct repositories, so they will not work with the [Centralized Workflow](https://www.atlassian.com/git/tutorials/comparing-workflows/forking-workflow). Using pull requests with each of these workflows is slightly different, but the general process is as follows:

1. A developer creates the feature in a dedicated branch in their local repo.
2. The developer pushes the branch to a public Bitbucket repository.
3. The developer files a pull request via Bitbucket.
4. The rest of the team reviews the code, discusses it, and alters it.
5. The project maintainer merges the feature into the official repository and closes the pull request.

The rest of this section describes how pull requests can be leveraged against different collaboration workflows.

### Feature Branch Workflow With Pull Requests

The Feature Branch Workflow uses a shared Bitbucket repository for managing collaboration, and developers create features in isolated branches. But, instead of immediately merging them into main, developers should open a pull request to initiate a discussion around the feature before it gets integrated into the main codebase.



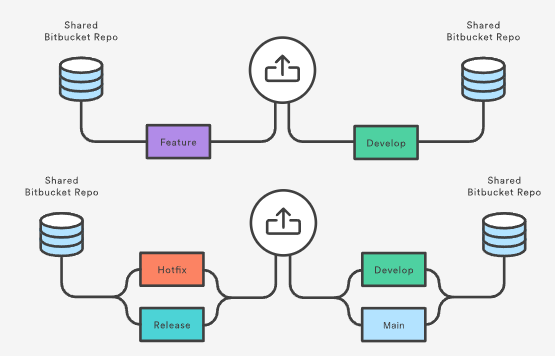
There is only one public repository in the Feature Branch Workflow, so the pull request’s destination repository and the source repository will always be the same. Typically, the developer will specify their feature branch as the source branch and the main branch as the destination branch.

After receiving the pull request, the project maintainer has to decide what to do. If the feature is ready to go, they can simply merge it into main and close the pull request. But, if there are problems with the proposed changes, they can post feedback in the pull request. Follow-up commits will show up right next to the relevant comments.

It’s also possible to file a pull request for a feature that is incomplete. For example, if a developer is having trouble implementing a particular requirement, they can file a pull request containing their work-in-progress. Other developers can then provide suggestions inside of the pull request, or even fix the problem themselves with additional commits.

### Gitflow Workflow With Pull Requests

The Gitflow Workflow is similar to the Feature Branch Workflow, but defines a strict branching model designed around the project release. Adding pull requests to the Gitflow Workflow gives developers a convenient place to talk about a release branch or a maintenance branch while they’re working on it.



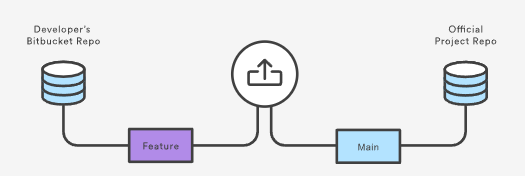
The mechanics of pull requests in the Gitflow Workflow are the exact same as the previous section: a developer simply files a pull request when a feature, release, or hotfix branch needs to be reviewed, and the rest of the team will be notified via Bitbucket.

Features are generally merged into the develop branch, while release and hotfix branches are merged into both develop and main. Pull requests can be used to formally manage all of these merges.

### Forking Workflow With Pull Requests

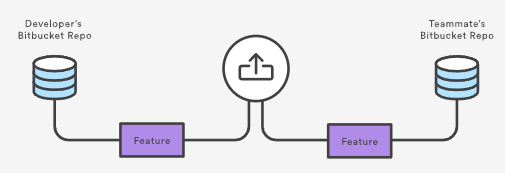
In the Forking Workflow, a developer pushes a completed feature to their own public repository instead of a shared one. After that, they file a pull request to let the project maintainer know that it’s ready for review.

The notification aspect of pull requests is particularly useful in this workflow because the project maintainer has no way of knowing when another developer has added commits to their Bitbucket repository.



Since each developer has their own public repository, the pull request’s source repository will differ from its destination repository. The source repository is the developer’s public repository and the source branch is the one that contains the proposed changes. If the developer is trying to merge the feature into the main codebase, then the destination repository is the official project and the destination branch is main.

Pull requests can also be used to collaborate with other developers outside of the official project. For example, if a developer was working on a feature with a teammate, they could file a pull request using the teammate’s Bitbucket repository for the destination instead of the official project. They would then use the same feature branch for the source and destination branches.



The two developers could discuss and develop the feature inside of the pull request. When they’re done, one of them would file another pull request asking to merge the feature into the official main branch. This kind of flexibility makes pull requests very powerful collaboration tool in the Forking workflow.

## Example

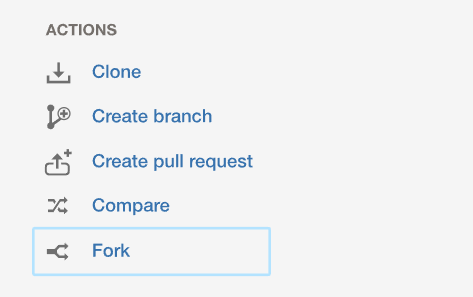
The example below demonstrates how pull requests can be used in the Forking Workflow. It is equally applicable to developers working in small teams and to a third-party developer contributing to an open source project.

In the example, Mary is a developer, and John is the project maintainer. Both of them have their own public Bitbucket repositories, and John’s contains the official project.

### Mary forks the official project

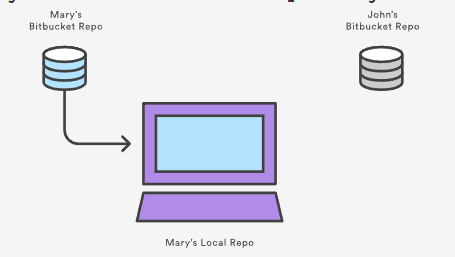


To start working in the project, Mary first needs to fork John’s Bitbucket repository. She can do this by signing in to Bitbucket, navigating to John’s repository, and clicking the Fork button.



After filling out the name and description for the forked repository, she will have a server-side copy of the project.

### Mary clones her Bitbucket repository

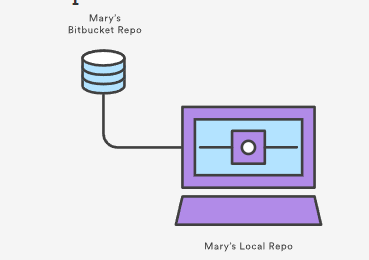


Next, Mary needs to clone the Bitbucket repository that she just forked. This will give her a working copy of the project on her local machine. She can do this by running the following command:

git clone https://user@bitbucket.org/user/repo.git

Keep in mind that git clone automatically creates an origin remote that points back to Mary’s forked repository.

### Mary develops a new feature

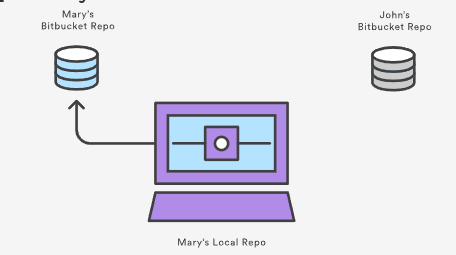


Before she starts writing any code, Mary needs to create a new branch for the feature. This branch is what she will use as the source branch of the pull request.

git checkout -b some-feature  
# Edit some code  
git commit -a -m "Add first draft of some feature"

Mary can use as many commits as she needs to create the feature. And, if the feature’s history is messier than she would like, she can use an [interactive rebase](https://www.atlassian.com/git/tutorials/rewriting-history/git-rebase) to remove or squash unnecessary commits. For larger projects, cleaning up a feature’s history makes it much easier for the project maintainer to see what’s going on in the pull request.

### Mary pushes the feature to her Bitbucket repository

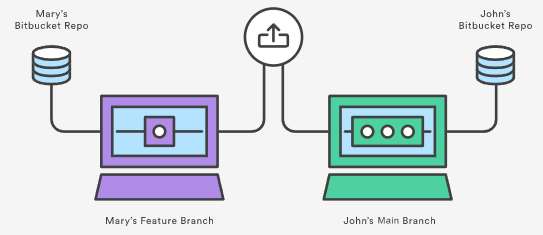


After her feature is complete, Mary pushes the feature branch to her own Bitbucket repository (not the official repository) with a simple git push:

git push origin some-branch

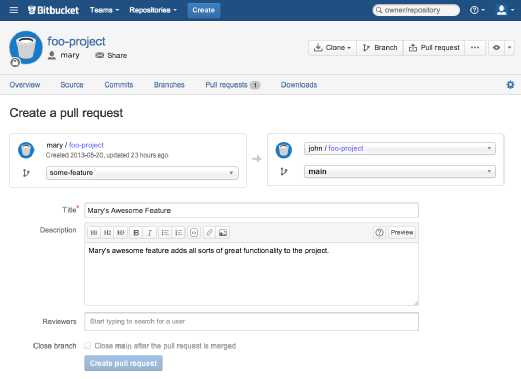
This makes her changes available to the project maintainer (or any collaborators who might need access to them).

### Mary creates the pull request



After Bitbucket has her feature branch, Mary can create the pull request through her Bitbucket account by navigating to her forked repository and clicking the Pull request button in the top-right corner. The resulting form automatically sets Mary’s repository as the source repository, and it asks her to specify the source branch, the destination repository, and the destination branch.

Mary wants to merge her feature into the main codebase, so the source branch is her feature branch, the destination repository is John’s public repository, and the destination branch is main. She’ll also need to provide a title and description for the pull request. If there are other people who need to approve the code besides John, she can enter them in the Reviewers field.



After she creates the pull request, a notification will be sent to John via his Bitbucket feed and (optionally) via email.

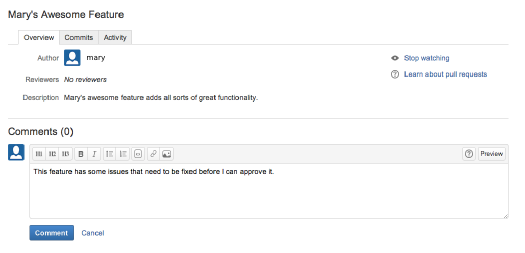
### John reviews the pull request



John can access all of the pull requests people have filed by clicking on the Pull request tab in his own Bitbucket repository. Clicking on Mary’s pull request will show him a description of the pull request, the feature’s commit history, and a diff of all the changes it contains.

If he thinks the feature is ready to merge into the project, all he has to do is hit the Merge button to approve the pull request and merge Mary’s feature into his main branch.

But, for this example, let’s say John found a small bug in Mary’s code, and needs her to fix it before merging it in. He can either post a comment to the pull request as a whole, or he can select a specific commit in the feature’s history to comment on.



### Mary adds a follow-up commit

If Mary has any questions about the feedback, she can respond inside of the pull request, treating it as a discussion forum for her feature.

To correct the error, Mary adds another commit to her feature branch and pushes it to her Bitbucket repository, just like she did the first time around. This commit is automatically added to the original pull request, and John can review the changes again, right next to his original comment.

### John accepts the pull request

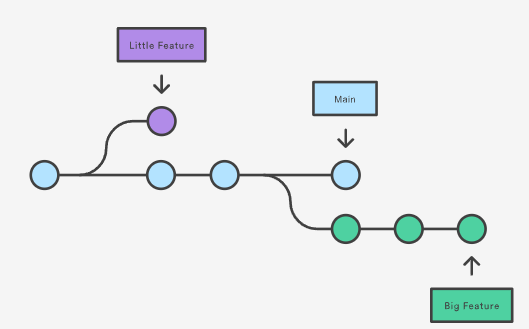
Finally, John accepts the changes, merges the feature branch into main, and closes the pull request. The feature is now integrated into the project, and any other developers working on it can pull it into their own local repositories using the standard git pull command.

## Where to go from here

You should now have all of the tools you need to start integrating pull requests into your existing workflow. Remember, pull requests are not a replacement for any of the [Git-based collaboration workflows](https://www.atlassian.com/git/tutorials/comparing-workflows), but rather a convenient addition to them that makes collaboration more accessible to all of your team members.

# Git Branch

This document is an in-depth review of the git branch command and a discussion of the overall Git branching model. Branching is a feature available in most modern version control systems. Branching in other VCS's can be an expensive operation in both time and disk space. In Git, branches are a part of your everyday development process. Git branches are effectively a pointer to a snapshot of your changes. When you want to add a new feature or fix a bug—no matter how big or how small—you spawn a new branch to encapsulate your changes. This makes it harder for unstable code to get merged into the main code base, and it gives you the chance to clean up your future's history before merging it into the main branch.



The diagram above visualizes a repository with two isolated lines of development, one for a little feature, and one for a longer-running feature. By developing them in branches, it’s not only possible to work on both of them in parallel, but it also keeps the main branch free from questionable code.

The implementation behind Git branches is much more lightweight than other version control system models. Instead of copying files from directory to directory, Git stores a branch as a reference to a commit. In this sense, a branch represents the tip of a series of commits—it's not a container for commits. The history for a branch is extrapolated through the commit relationships.

As you read, remember that Git branches aren't like SVN branches. Whereas SVN branches are only used to capture the occasional large-scale development effort, Git branches are an integral part of your everyday workflow. The following content will expand on the internal Git branching architecture.

## How it works

A branch represents an independent line of development. Branches serve as an abstraction for the edit/stage/commit process. You can think of them as a way to request a brand new working directory, staging area, and project history. New commits are recorded in the history for the current branch, which results in a fork in the history of the project.

The git branch command lets you create, list, rename, and delete branches. It doesn’t let you switch between branches or put a forked history back together again. For this reason, git branch is tightly integrated with the [git checkout](https://www.atlassian.com/git/tutorials/using-branches/git-checkout) and [git merge](https://www.atlassian.com/git/tutorials/using-branches/git-merge) commands.

## Common Options

git branch

List all of the branches in your repository. This is synonymous with git branch --list.

git branch <branch>

Create a new branch called ＜branch＞. This does not check out the new branch.

git branch -d <branch>

Delete the specified branch. This is a “safe” operation in that Git prevents you from deleting the branch if it has unmerged changes.

git branch -D <branch>

Force delete the specified branch, even if it has unmerged changes. This is the command to use if you want to permanently throw away all of the commits associated with a particular line of development.

git branch -m <branch>

Rename the current branch to ＜branch＞.

git branch -a

List all remote branches.

## Creating Branches

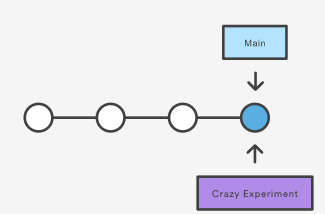
It's important to understand that branches are just pointers to commits. When you create a branch, all Git needs to do is create a new pointer, it doesn’t change the repository in any other way. If you start with a repository that looks like this:



Then, you create a branch using the following command:

git branch crazy-experiment

The repository history remains unchanged. All you get is a new pointer to the current commit:



Note that this only creates the new branch. To start adding commits to it, you need to select it with git checkout, and then use the standard git add and git commit commands.

## Creating remote branches

So far these examples have all demonstrated local branch operations. The git branch command also works on remote branches. In order to operate on remote branches, a remote repo must first be configured and added to the local repo config.

$ git remote add new-remote-repo https://bitbucket.com/user/repo.git  
# Add remote repo to local repo config  
$ git push <new-remote-repo> crazy-experiment~  
# pushes the crazy-experiment branch to new-remote-repo

This command will push a copy of the local branch crazy-experiment to the remote repo ＜remote＞.

## Deleting Branches

Once you’ve finished working on a branch and have merged it into the main code base, you’re free to delete the branch without losing any history:

git branch -d crazy-experiment

However, if the branch hasn’t been merged, the above command will output an error message:

error: The branch 'crazy-experiment' is not fully merged. If you are sure you want to delete it, run 'git branch -D crazy-experiment'.

This protects you from losing access to that entire line of development. If you really want to delete the branch (e.g., it’s a failed experiment), you can use the capital -D flag:

git branch -D crazy-experiment

This deletes the branch regardless of its status and without warnings, so use it judiciously.

The previous commands will delete a local copy of a branch. The branch may still exist in remote repos. To delete a remote branch execute the following.

git push origin --delete crazy-experiment

Or

git push origin :crazy-experiment

This will push a delete signal to the remote origin repository that triggers a delete of the remote crazy-experiment branch.

## Summary

In this document we discussed Git's branching behavior and the git branch command. The git branch commands primary functions are to create, list, rename and delete branches. To operate further on the resulting branches the command is commonly used with other commands like git checkout. Learn more about git checkout branch operations; such as switching branches and merging branches, on the [git checkout](https://www.atlassian.com/git/tutorials/using-branches/git-checkout) page.

Compared to other VCSs, Git's branch operations are inexpensive and frequently used. This flexibility enables powerful [Git workflow](https://www.atlassian.com/git/tutorials/comparing-workflows) customization. For more info on Git workflows visit our extended workflow discussion pages: [The Feature Branch Workflow](https://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow), [GitFlow Workflow](https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow), and [Forking Workflow](https://www.atlassian.com/git/tutorials/comparing-workflows/forking-workflow).

# Git Checkout

This page is an examination of the git checkout command. It will cover usage examples and edge cases. In Git terms, a "checkout" is the act of switching between different versions of a target entity. The git checkout command operates upon three distinct entities: files, commits, and branches. In addition to the definition of "checkout" the phrase "checking out" is commonly used to imply the act of executing the git checkout command. In the [Undoing Changes](https://www.atlassian.com/git/tutorials/undoing-changes) topic, we saw how git checkout can be used to view old commits. The focus for the majority of this document will be checkout operations on branches.

Checking out branches is similar to checking out old commits and files in that the working directory is updated to match the selected branch/revision; however, new changes are saved in the project history—that is, it’s not a read-only operation.

## Checking out branches

The git checkout command lets you navigate between the branches created by git branch. Checking out a branch updates the files in the working directory to match the version stored in that branch, and it tells Git to record all new commits on that branch. Think of it as a way to select which line of development you’re working on.

Having a dedicated branch for each new feature is a dramatic shift from a traditional SVN workflow. It makes it ridiculously easy to try new experiments without the fear of destroying existing functionality, and it makes it possible to work on many unrelated features at the same time. In addition, branches also facilitate several collaborative workflows.

The git checkout command may occasionally be confused with git clone. The difference between the two commands is that clone works to fetch code from a remote repository, alternatively checkout works to switch between versions of code already on the local system.

## Usage: Existing branches

Assuming the repo you're working in contains pre-existing branches, you can switch between these branches using git checkout. To find out what branches are available and what the current branch name is, execute git branch.

$＞ git branch   
main   
another\_branch   
feature\_inprogress\_branch   
$＞ git checkout feature\_inprogress\_branch

The above example demonstrates how to view a list of available branches by executing the git branch command, and switch to a specified branch, in this case, the feature\_inprogress\_branch.

## New Branches

Git checkout works hand-in-hand with [git branch](https://www.atlassian.com/git/tutorials/using-branches). The git branch command can be used to create a new branch. When you want to start a new feature, you create a new branch off main using git branch new\_branch. Once created you can then use git checkout new\_branch to switch to that branch. Additionally, The git checkout command accepts a -b argument that acts as a convenience method which will create the new branch and immediately switch to it. You can work on multiple features in a single repository by switching between them with git checkout.

git checkout -b ＜new-branch＞

The above example simultaneously creates and checks out . The -b option is a convenience flag that tells Git to run git branch before running git checkout.

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

By default git checkout -b will base the new-branch off the current HEAD. An optional additional branch parameter can be passed to git checkout. In the above example, ＜existing-branch＞ is passed which then bases new-branch off of existing-branch instead of the current HEAD.

## Switching Branches

Switching branches is a straightforward operation. Executing the following will point HEAD to the tip of .

git checkout ＜branchname＞

Git tracks a history of checkout operations in the reflog. You can execute git reflog to view the history.

## Git Checkout a Remote Branch

When collaborating with a team it is common to utilize remote repositories. These repositories may be hosted and shared or they may be another colleague's local copy. Each remote repository will contain its own set of branches. In order to checkout a remote branch you have to first fetch the contents of the branch.

git fetch --all

In modern versions of Git, you can then checkout the remote branch like a local branch.

git checkout ＜remotebranch＞

Older versions of Git require the creation of a new branch based on the remote.

git checkout -b ＜remotebranch＞ origin/＜remotebranch＞

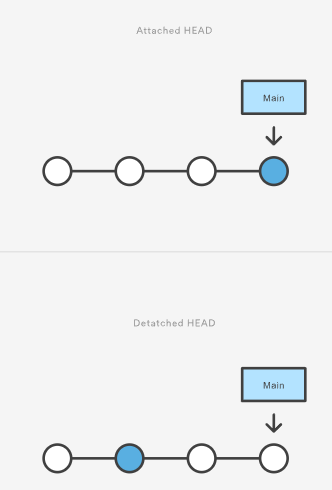
Additionally you can checkout a new local branch and reset it to the remote branches last commit.

git checkout -b ＜branchname＞  
git reset --hard origin/＜branchname＞

## Detached HEADS

Now that we’ve seen the three main uses of git checkout on branches, it's important to discuss the “detached HEAD” state. Remember that the HEAD is Git’s way of referring to the current snapshot. Internally, the git checkout command simply updates the HEAD to point to either the specified branch or commit. When it points to a branch, Git doesn't complain, but when you check out a commit, it switches into a “detached HEAD” state.

This is a warning telling you that everything you’re doing is “detached” from the rest of your project’s development. If you were to start developing a feature while in a detached HEAD state, there would be no branch allowing you to get back to it. When you inevitably check out another branch (e.g., to merge your feature in), there would be no way to reference your feature:



The point is, your development should always take place on a branch—never on a detached HEAD. This makes sure you always have a reference to your new commits. However, if you’re just looking at an old commit, it doesn’t really matter if you’re in a detached HEAD state or not.

## Summary

This page focused on usage of the git checkout command when changing branches. In summation, git checkout, when used on branches, alters the target of the HEAD ref. It can be used to create branches, switch branches, and checkout remote branches. The git checkout command is an essential tool for standard Git operation. It is a counterpart to [git merge](https://www.atlassian.com/git/tutorials/using-branches/git-merge). The git checkout and git merge commands are critical tools to enabling [git workflows.](https://www.atlassian.com/git/tutorials/comparing-workflows)

# Git Merge

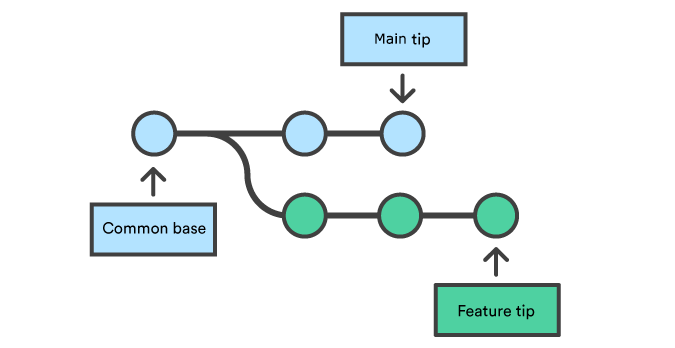
Merging is Git's way of putting a forked history back together again. The git merge command lets you take the independent lines of development created by git branch and integrate them into a single branch.

Note that all of the commands presented below merge into the current branch. The current branch will be updated to reflect the merge, but the target branch will be completely unaffected. Again, this means that git merge is often used in conjunction with git checkout for selecting the current branch and git branch -d for deleting the obsolete target branch.

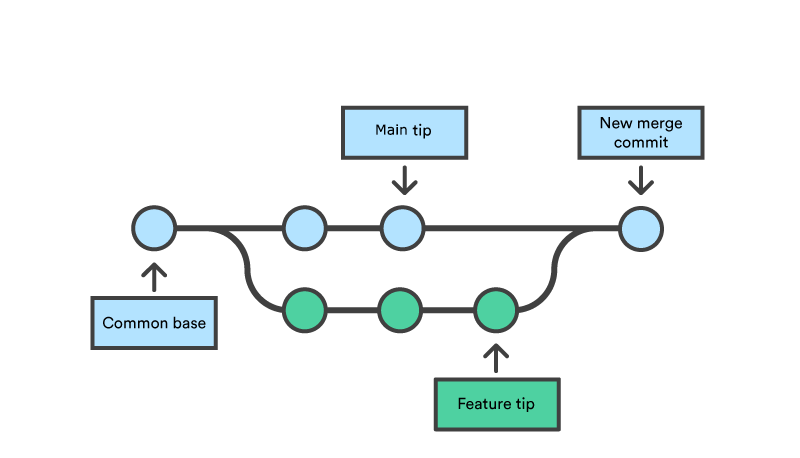
## How it works

Git merge will combine multiple sequences of commits into one unified history. In the most frequent use cases, git merge is used to combine two branches. The following examples in this document will focus on this branch merging pattern. In these scenarios, git merge takes two commit pointers, usually the branch tips, and will find a common base commit between them. Once Git finds a common base commit it will create a new "merge commit" that combines the changes of each queued merge commit sequence.

Say we have a new branch feature that is based off the main branch. We now want to merge this feature branch into main.



Invoking this command will merge the specified branch feature into the current branch, we'll assume main. Git will determine the merge algorithm automatically (discussed below).



Merge commits are unique against other commits in the fact that they have two parent commits. When creating a merge commit Git will attempt to auto magically merge the separate histories for you. If Git encounters a piece of data that is changed in both histories it will be unable to automatically combine them. This scenario is a version control conflict and Git will need user intervention to continue.

## Preparing to merge

Before performing a merge there are a couple of preparation steps to take to ensure the merge goes smoothly.

## Confirm the receiving branch

Execute git status to ensure that HEAD is pointing to the correct merge-receiving branch. If needed, execute git checkout to switch to the receiving branch. In our case we will execute git checkout main.

## Fetch latest remote commits

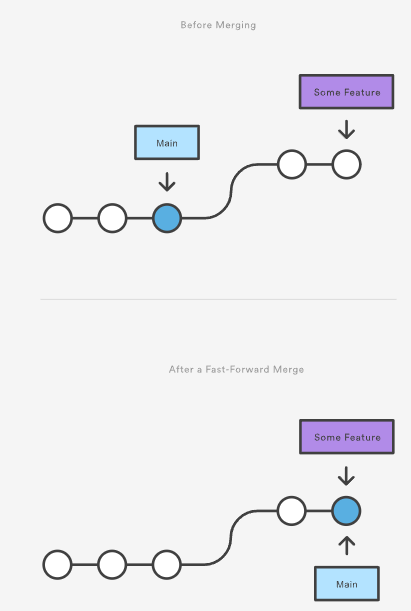
Make sure the receiving branch and the merging branch are up-to-date with the latest remote changes. Execute git fetch to pull the latest remote commits. Once the fetch is completed ensure the main branch has the latest updates by executing git pull.

## Merging

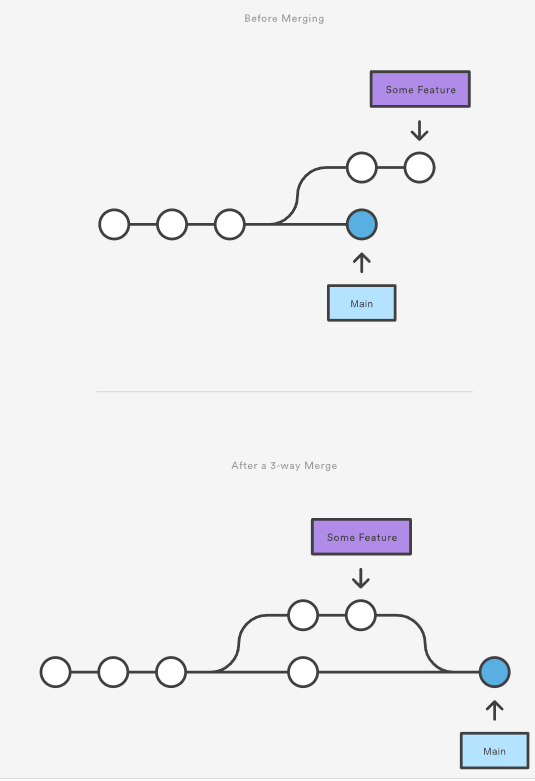
Once the previously discussed "preparing to merge" steps have been taken a merge can be initiated by executing git merge where  is the name of the branch that will be merged into the receiving branch.

## Fast Forward Merge

A fast-forward merge can occur when there is a linear path from the current branch tip to the target branch. Instead of “actually” merging the branches, all Git has to do to integrate the histories is move (i.e., “fast forward”) the current branch tip up to the target branch tip. This effectively combines the histories, since all of the commits reachable from the target branch are now available through the current one. For example, a fast forward merge of some-feature into main would look something like the following:



However, a fast-forward merge is not possible if the branches have diverged. When there is not a linear path to the target branch, Git has no choice but to combine them via a 3-way merge. 3-way merges use a dedicated commit to tie together the two histories. The nomenclature comes from the fact that Git uses three commits to generate the merge commit: the two branch tips and their common ancestor.

  
While you can use either of these merge strategies, many developers like to use fast-forward merges (facilitated through [rebasing](https://www.atlassian.com/git/tutorials/rewriting-history/git-rebase)) for small features or bug fixes, while reserving 3-way merges for the integration of longer-running features. In the latter case, the resulting merge commit serves as a symbolic joining of the two branches.

Our first example demonstrates a fast-forward merge. The code below creates a new branch, adds two commits to it, then integrates it into the main line with a fast-forward merge.

# Start a new feature  
git checkout -b new-feature main  
# Edit some files  
git add <file>  
git commit -m "Start a feature"  
# Edit some files  
git add <file>  
git commit -m "Finish a feature"  
# Merge in the new-feature branch  
git checkout main  
git merge new-feature  
git branch -d new-feature

This is a common workflow for short-lived topic branches that are used more as an isolated development than an organizational tool for longer-running features.

Also note that Git should not complain about the git branch -d, since new-feature is now accessible from the main branch.

In the event that you require a merge commit during a fast forward merge for record keeping purposes you can execute git merge with the --no-ffoption.

git merge --no-ff <branch>

This command merges the specified branch into the current branch, but always generates a merge commit (even if it was a fast-forward merge). This is useful for documenting all merges that occur in your repository.

## 3-way merge

The next example is very similar, but requires a 3-way merge because main progresses while the feature is in-progress. This is a common scenario for large features or when several developers are working on a project simultaneously.

Start a new feature  
git checkout -b new-feature main  
# Edit some files  
git add <file>  
git commit -m "Start a feature"  
# Edit some files  
git add <file>  
git commit -m "Finish a feature"  
# Develop the main branch  
git checkout main  
# Edit some files  
git add <file>  
git commit -m "Make some super-stable changes to main"  
# Merge in the new-feature branch  
git merge new-feature  
git branch -d new-feature

Note that it’s impossible for Git to perform a fast-forward merge, as there is no way to move main up to new-feature without backtracking.

For most workflows, new-feature would be a much larger feature that took a long time to develop, which would be why new commits would appear on main in the meantime. If your feature branch was actually as small as the one in the above example, you would probably be better off rebasing it onto main and doing a fast-forward merge. This prevents superfluous merge commits from cluttering up the project history.

## Resolving conflict

If the two branches you're trying to merge both changed the same part of the same file, Git won't be able to figure out which version to use. When such a situation occurs, it stops right before the merge commit so that you can resolve the conflicts manually.

The great part of Git's merging process is that it uses the familiar edit/stage/commit workflow to resolve merge conflicts. When you encounter a merge conflict, running the git status command shows you which files need to be resolved. For example, if both branches modified the same section of hello.py, you would see something like the following:

On branch main  
Unmerged paths:  
(use "git add/rm ..." as appropriate to mark resolution)  
both modified: hello.py

## How conflicts are presented

When Git encounters a conflict during a merge, It will edit the content of the affected files with visual indicators that mark both sides of the conflicted content. These visual markers are: <<<<<<<, =======, and >>>>>>>. Its helpful to search a project for these indicators during a merge to find where conflicts need to be resolved.

here is some content not affected by the conflict  
<<<<<<< main  
this is conflicted text from main  
=======  
this is conflicted text from feature branch  
>>>>>>> feature branch;

Generally the content before the ======= marker is the receiving branch and the part after is the merging branch.

Once you've identified conflicting sections, you can go in and fix up the merge to your liking. When you're ready to finish the merge, all you have to do is run git add on the conflicted file(s) to tell Git they're resolved. Then, you run a normal git commit to generate the merge commit. It’s the exact same process as committing an ordinary snapshot, which means it’s easy for normal developers to manage their own merges.

Note that merge conflicts will only occur in the event of a 3-way merge. It’s not possible to have conflicting changes in a fast-forward merge.

## Summary

This document is an overview of the git merge command. Merging is an essential process when working with Git. We discussed the internal mechanics behind a merge and the differences between a fast forward merge and a three way, true merge. Some key take-aways are:

1. Git merging combines sequences of commits into one unified history of commits.
2. There are two main ways Git will merge: Fast Forward and Three way
3. Git can automatically merge commits unless there are changes that conflict in both commit sequences.

This document integrated and referenced other Git commands like: [git branch](https://www.atlassian.com/git/tutorials/using-branches), [git pull](https://www.atlassian.com/git/tutorials/syncing/git-pull), and [git fetch](https://www.atlassian.com/git/tutorials/syncing/git-fetch). Visit their corresponding stand-alone pages for more information.

# Git merge conflicts

Version control systems are all about managing contributions between multiple distributed authors ( usually developers ). Sometimes multiple developers may try to edit the same content. If Developer A tries to edit code that Developer B is editing a conflict may occur. To alleviate the occurrence of conflicts developers will work in separate [isolated branches](https://www.atlassian.com/git/tutorials/using-branches). The git merge command's primary responsibility is to combine separate branches and resolve any conflicting edits.

## Understanding merge conflicts

Merging and conflicts are a common part of the Git experience. Conflicts in other version control tools like SVN can be costly and time-consuming. Git makes merging super easy. Most of the time, Git will figure out how to automatically integrate new changes.

Conflicts generally arise when two people have changed the same lines in a file, or if one developer deleted a file while another developer was modifying it. In these cases, Git cannot automatically determine what is correct. Conflicts only affect the developer conducting the merge, the rest of the team is unaware of the conflict. Git will mark the file as being conflicted and halt the merging process. It is then the developers' responsibility to resolve the conflict.

## Types of merge conflicts

A merge can enter a conflicted state at two separate points. When starting and during a merge process. The following is a discussion of how to address each of these conflict scenarios.

### Git fails to start the merge

A merge will fail to start when Git sees there are changes in either the working directory or staging area of the current project. Git fails to start the merge because these pending changes could be written over by the commits that are being merged in. When this happens, it is not because of conflicts with other developer's, but conflicts with pending local changes. The local state will need to be stabilized using git stash, git checkout, git commit or git reset. A merge failure on start will output the following error message:

error: Entry '<fileName>' not uptodate. Cannot merge. (Changes in working directory)

### Git fails during the merge

A failure DURING a merge indicates a conflict between the current local branch and the branch being merged. This indicates a conflict with another developers code. Git will do its best to merge the files but will leave things for you to resolve manually in the conflicted files. A mid-merge failure will output the following error message:

error: Entry '<fileName>' would be overwritten by merge. Cannot merge. (Changes in staging area)

## Creating a merge conflict

In order to get real familiar with merge conflicts, the next section will simulate a conflict to later examine and resolve. The example will be using a Unix-like command-line Git interface to execute the example simulation.

$ mkdir git-merge-test  
$ cd git-merge-test  
$ git init .  
$ echo "this is some content to mess with" > merge.txt  
$ git add merge.txt  
$ git commit -am"we are commiting the inital content"  
[main (root-commit) d48e74c] we are commiting the inital content  
1 file changed, 1 insertion(+)  
create mode 100644 merge.txt

This code example executes a sequence of commands that accomplish the following.

* Create a new directory named git-merge-test, change to that directory, and initialize it as a new Git repo.
* Create a new text file merge.txt with some content in it.
* Add merge.txt to the repo and commit it.

Now we have a new repo with one branch main and a file merge.txt with content in it. Next, we will create a new branch to use as the conflicting merge.

$ git checkout -b new\_branch\_to\_merge\_later  
$ echo "totally different content to merge later" > merge.txt  
$ git commit -am"edited the content of merge.txt to cause a conflict"  
[new\_branch\_to\_merge\_later 6282319] edited the content of merge.txt to cause a conflict  
1 file changed, 1 insertion(+), 1 deletion(-)

The proceeding command sequence achieves the following:

* create and check out a new branch named new\_branch\_to\_merge\_later
* overwrite the content in merge.txt
* commit the new content

With this new branch: new\_branch\_to\_merge\_later we have created a commit that overrides the content of merge.txt

git checkout main  
Switched to branch 'main'  
echo "content to append" >> merge.txt  
git commit -am"appended content to merge.txt"  
[main 24fbe3c] appended content to merge.tx  
1 file changed, 1 insertion(+)

This chain of commands checks out the main branch, appends content to merge.txt, and commits it. This now puts our example repo in a state where we have 2 new commits. One in the main branch and one in the new\_branch\_to\_merge\_later branch. At this time lets git merge new\_branch\_to\_merge\_later and see what happen!

$ git merge new\_branch\_to\_merge\_later  
Auto-merging merge.txt  
CONFLICT (content): Merge conflict in merge.txt  
Automatic merge failed; fix conflicts and then commit the result.

BOOM 💥. A conflict appears. Thanks, Git for letting us know about this!

## How to identify merge conflicts

As we have experienced from the proceeding example, Git will produce some descriptive output letting us know that a CONFLICT has occcured. We can gain further insight by running the [git status](https://www.atlassian.com/git/tutorials/inspecting-a-repository) command

$ git status  
On branch main  
You have unmerged paths.  
(fix conflicts and run "git commit")  
(use "git merge --abort" to abort the merge)  
  
Unmerged paths:  
(use "git add <file>..." to mark resolution)  
  
both modified:   merge.txt

The output from git status indicates that there are unmerged paths due to a conflict. The merge.text file now appears in a modified state. Let's examine the file and see whats modified.

$ cat merge.txt  
<<<<<<< HEAD  
this is some content to mess with  
content to append  
=======  
totally different content to merge later  
>>>>>>> new\_branch\_to\_merge\_later

Here we have used the cat command to put out the contents of the merge.txt file. We can see some strange new additions

* <<<<<<< HEAD
* =======
* >>>>>>> new\_branch\_to\_merge\_later

Think of these new lines as "conflict dividers". The ======= line is the "center" of the conflict. All the content between the center and the <<<<<<< HEAD line is content that exists in the current branch main which the HEAD ref is pointing to. Alternatively all content between the center and >>>>>>> new\_branch\_to\_merge\_later is content that is present in our merging branch.

## How to resolve merge conflicts using the command line

The most direct way to resolve a merge conflict is to edit the conflicted file. Open the merge.txt file in your favorite editor. For our example lets simply remove all the conflict dividers. The modified merge.txt content should then look like:

this is some content to mess with  
content to append  
totally different content to merge later

Once the file has been edited use git add merge.txt to stage the new merged content. To finalize the merge create a new commit by executing:

git commit -m "merged and resolved the conflict in merge.txt"

Git will see that the conflict has been resolved and creates a new merge commit to finalize the merge.

## Git commands that can help resolve merge conflicts

### General tools

git status

The status command is in frequent use when a working with Git and during a merge it will help identify conflicted files.

git log --merge

Passing the --merge argument to the git log command will produce a log with a list of commits that conflict between the merging branches.

git diff

diff helps find differences between states of a repository/files. This is useful in predicting and preventing merge conflicts.

### Tools for when git fails to start a merge

git checkout

checkout can be used for undoing changes to files, or for changing branches

git reset --mixed

reset can be used to undo changes to the working directory and staging area.

### Tools for when git conflicts arise during a merge

git merge --abort

Executing git merge with the --abort option will exit from the merge process and return the branch to the state before the merge began.

git reset

Git reset can be used during a merge conflict to reset conflicted files to a know good state

## Summary

Merge conflicts can be an intimidating experience. Luckily, Git offers powerful tools to help navigate and resolve conflicts. Git can handle most merges on its own with automatic merging features. A conflict arises when two separate branches have made edits to the same line in a file, or when a file has been deleted in one branch but edited in the other. Conflicts will most likely happen when working in a team environment.

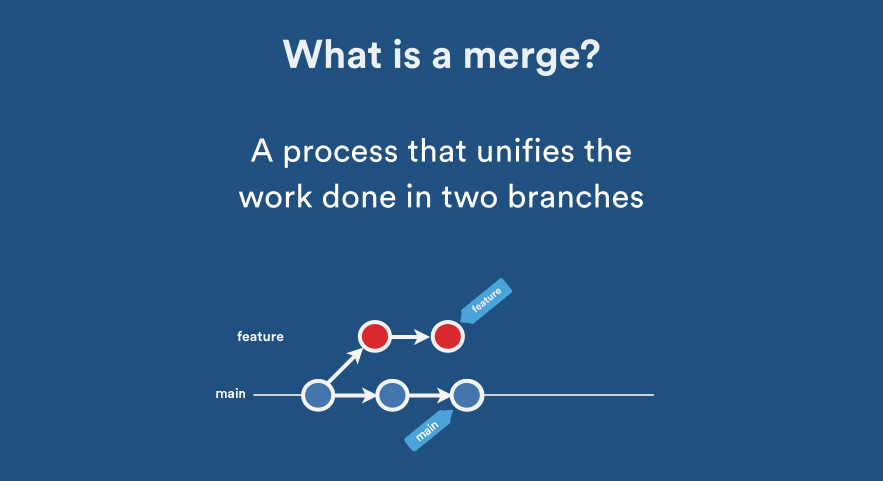
There are many tools to help resolve merge conflicts. Git has plenty of command line tools we discussed here. For more detailed information on these tools visit stand-alone pages for [git log](https://www.atlassian.com/git/tutorials/git-log), [git reset](https://www.atlassian.com/git/tutorials/undoing-changes/git-reset), [git status](https://www.atlassian.com/git/tutorials/inspecting-a-repository), [git checkout](https://www.atlassian.com/git/tutorials/using-branches/git-checkout), and [git reset](https://www.atlassian.com/git/tutorials/undoing-changes/git-reset). In addition to the Git, many third-party tools offer streamlined merge conflict support features.

Git Merge Strategy Options and Examples

When a piece of work is complete, tested and ready to be merged back into the main line of development, your team has some policy choices to make. What are your merge strategy options? In this article we'll examine the possibilities and then provide some notes on how Atlassian operates. Hopefully at the end you'll have the tools to decide what works best for your team.

Git Merge Strategies

A merge happens when combining two branches. Git will take two (or more) commit pointers and attempt to find a common base commit between them. Git has several different methods to find a base commit, these methods are called "merge strategies". Once Git finds a common base commit it will create a new "merge commit" that combines the changes of the specified merge commits. Technically, a merge commit is a regular commit which just happens to have two parent commits.



git merge will automatically select a merge strategy unless explicitly specified. The git merge and git pull commands can be passed an -s (strategy) option. The -s option can be appended with the name of the desired merge strategy. If not explicitly specified, Git will select the most appropriate merge strategy based on the provided branches. The following is a list of the available merge strategies.

Recursive

git merge -s recursive branch1 branch2

This operates on two heads. Recursive is the default merge strategy when pulling or merging one branch. Additionally this can detect and handle merges involving renames, but currently cannot make use of detected copies. This is the default merge strategy when pulling or merging one branch.

Resolve

git merge -s resolve branch1 branch2

This can only resolve two heads using a 3-way merge algorithm. It tries to carefully detect cris-cross merge ambiguities and is considered generally safe and fast.

Octopus

git merge -s octopus branch1 branch2 branch3 branchN

The default merge strategy for more than two heads. When more than one branch is passed octopus is automatically engaged. If a merge has conflicts that need manual resolution octopus will refuse the merge attempt. It is primarily used for bundling similar feature branch heads together.

Ours

git merge -s ours branch1 branch2 branchN

The Ours strategy operates on multiple N number of branches. The output merge result is always that of the current branch HEAD. The "ours" term implies the preference effectively ignoring all changes from all other branches. It is intended to be used to combine history of similar feature branches.

Subtree

git merge -s subtree branchA branchB

This is an extension of the recursive strategy. When merging A and B, if B is a child subtree of A, B is first updated to reflect the tree structure of A, This update is also done to the common ancestor tree that is shared between A and B.

Types of Git Merge Strategies

Explicit Merges

Explicit merges are the default merge type. The 'explicit' part is that they create a new merge commit. This alters the commit history and explicitly shows where a merge was executed. The merge commit content is also explicit in the fact that it shows which commits were the parents of the merge commit. Some teams avoid explicit merges because arguably the merge commits add "noise" to the history of the project.

implicit merge via rebase or fast-forward merge

Squash on merge, generally without explicit merge

Recursive Git Merge Strategy Options

The 'recursive' strategy introduced above, has its own subset of additional operation options.

ours

Not to be confused with the Ours merge strategy. This option conflicts to be auto-resolved cleanly by favoring the 'our' version. Changes from the 'theirs' side are automatically incorporated if they do not conflict.

theirs

The opposite of the 'ours' strategy. the "theirs" option favors the foreign merging tree in conflict resolution.

patience

This option spends extra time to avoid mis-merges on unimportant matching lines. This options is best used when branches to be merged have extremely diverged.

diff-algorithim

ignore-\*  
  
    ignore-space-change  
    ignore-all-space  
    ignore-space-at-eol  
    ignore-cr-at-eol

A set of options that target whitespace characters. Any line that matches the subset of the passed option will be ignored.

renormalize

This option runs a check-out and check-in on all of the tree git trees while resolving a three-way merge. This option is intended to be used with merging branches with differing checkin/checkout states.

no-normalize

Disables the renormalize option. This overrides the merge.renormalize configuration variable.

no-renames

This option will ignore renamed files during the merge.

find-renames=n

This is the default behavior. The recursive merge will honor file renames. The n parameter can be used to pass a threshold for rename similarity. The default n value is 100%.

subtree

This option borrows from the `subtree` strategy. Where the strategy operates on two trees and modifies how to make them match on a shared ancestor, this option instead operates on the path metadata of the tree to make them match.

Our Git Merge Policy

Atlassian strongly prefers using explicit merges. The reason is very simple: explicit merges provide great traceability and context on the features being merged. A local history clean-up rebase before sharing a feature branch for review is absolutely encouraged, but this does not change the policy at all. It augments it.

# Comparing Workflows

Git is the most commonly used version control system today. A Git workflow is a recipe or recommendation for how to use Git to accomplish work in a consistent and productive manner. Git workflows encourage developers and [DevOps](https://www.atlassian.com/devops/what-is-devops) teams to leverage Git effectively and consistently. Git offers a lot of flexibility in how users manage changes. Given Git's focus on flexibility, there is no standardized process on how to interact with Git. When working with a team on a Git-managed project, it’s important to make sure the team is all in agreement on how the flow of changes will be applied. To ensure the team is on the same page, an agreed-upon Git workflow should be developed or selected. There are several publicized Git workflows that may be a good fit for your team. Here, we will discuss some of these Git workflow options.

The array of possible workflows can make it hard to know where to begin when implementing Git in the workplace. This page provides a starting point by surveying the most common Git workflows for software teams.

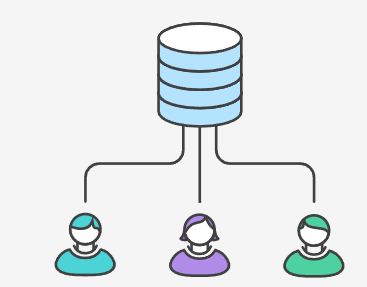
As you read through, remember that these workflows are designed to be guidelines rather than concrete rules. We want to show you what’s possible, so you can mix and match aspects from different workflows to suit your individual needs.

## What is a successful Git workflow?

When evaluating a workflow for your team, it's most important that you consider your team’s culture. You want the workflow to enhance the effectiveness of your team and not be a burden that limits productivity. Some things to consider when evaluating a Git workflow are:

* Does this workflow scale with team size?
* Is it easy to undo mistakes and errors with this workflow?
* Does this workflow impose any new unnecessary cognitive overhead to the team?

## Centralized Workflow



The Centralized Workflow is a great Git workflow for teams transitioning from SVN. Like Subversion, the Centralized Workflow uses a central repository to serve as the single point-of-entry for all changes to the project. Instead of trunk, the default development branch is called main and all changes are committed into this branch. This workflow doesn’t require any other branches besides main.

Transitioning to a distributed version control system may seem like a daunting task, but you don’t have to change your existing workflow to take advantage of Git. Your team can develop projects in the exact same way as they do with Subversion.

However, using Git to power your development workflow presents a few advantages over SVN. First, it gives every developer their own local copy of the entire project. This isolated environment lets each developer work independently of all other changes to a project - they can add commits to their local repository and completely forget about upstream developments until it's convenient for them.

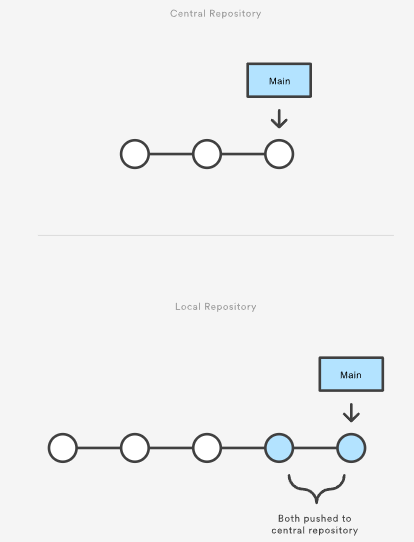
Second, it gives you access to Git’s robust branching and merging model. Unlike SVN, Git branches are designed to be a fail-safe mechanism for integrating code and sharing changes between repositories. The Centralized Workflow is similar to other workflows in its utilization of a remote server-side hosted repository that developers push and pull form. Compared to other workflows, the Centralized Workflow has no defined pull request or forking patterns. A Centralized Workflow is generally better suited for teams migrating from SVN to Git and smaller size teams.

## How it works

Developers start by cloning the central repository. In their own local copies of the project, they edit files and commit changes as they would with SVN; however, these new commits are stored locally - they’re completely isolated from the central repository. This lets developers defer synchronizing upstream until they’re at a convenient break point.

To publish changes to the official project, developers "push" their local main branch to the central repository. This is the equivalent of svn commit, except that it adds all of the local commits that aren’t already in the central main branch.

### Initialize the central repository



First, someone needs to create the central repository on a server. If it’s a new project, you can initialize an empty repository. Otherwise, you’ll need to import an existing Git or SVN repository.

Central repositories should always be bare repositories (they shouldn’t have a working directory), which can be created as follows:

ssh user@host git init --bare /path/to/repo.git

Be sure to use a valid SSH username for user, the domain or IP address of your server for host, and the location where you'd like to store your repo for /path/to/repo.git. Note that the .git extension is conventionally appended to the repository name to indicate that it’s a bare repository.

### Hosted central repositories

Central repositories are often created through 3rd party Git hosting services like [Bitbucket Cloud](https://bitbucket.org/product) or [Bitbucket Server](https://bitbucket.org/product/enterprise). The process of initializing a bare repository discussed above is handled for you by the hosting service. The hosting service will then provide an address for the central repository to access from your local repository.

### Clone the central repository

Next, each developer creates a local copy of the entire project. This is accomplished via the [git clone](https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-clone) command:

git clone ssh://user@host/path/to/repo.git

When you clone a repository, Git automatically adds a shortcut called origin that points back to the “parent” repository, under the assumption that you'll want to interact with it further on down the road.

### Make changes and commit

Once the repository is cloned locally, a developer can make changes using the standard Git commit process: edit, stage, and commit. If you’re not familiar with the staging area, it’s a way to prepare a commit without having to include every change in the working directory. This lets you create highly focused commits, even if you’ve made a lot of local changes.

git status # View the state of the repo  
git add <some-file> # Stage a file  
git commit # Commit a file</some-file>

Remember that since these commands create local commits, John can repeat this process as many times as he wants without worrying about what’s going on in the central repository. This can be very useful for large features that need to be broken down into simpler, more atomic chunks.

### Push new commits to central repository

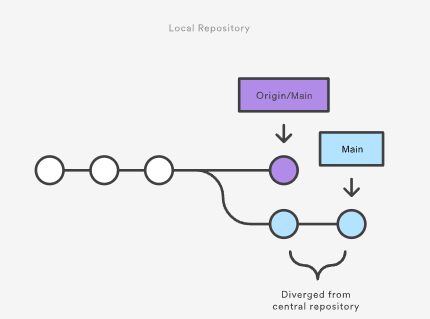
Once the local repository has new changes committed. These change will need to be pushed to share with other developers on the project.

git push origin main

This command will push the new committed changes to the central repository. When pushing changes to the central repository, it is possible that updates from another developer have been previously pushed that contain code which conflict with the intended push updates. Git will output a message indicating this conflict. In this situation, git pull will first need to be executed. This conflict scenario will be expanded on in the following section.

### Managing conflicts

The central repository represents the official project, so its commit history should be treated as sacred and immutable. If a developer’s local commits diverge from the central repository, Git will refuse to push their changes because this would overwrite official commits.



Before the developer can publish their feature, they need to fetch the updated central commits and rebase their changes on top of them. This is like saying, “I want to add my changes to what everyone else has already done.” The result is a perfectly linear history, just like in traditional SVN workflows.

If local changes directly conflict with upstream commits, Git will pause the rebasing process and give you a chance to manually resolve the conflicts. The nice thing about Git is that it uses the same git status and git add commands for both generating commits and resolving merge conflicts. This makes it easy for new developers to manage their own merges. Plus, if they get themselves into trouble, Git makes it very easy to abort the entire rebase and try again (or go find help).

## Example

Let’s take a general example at how a typical small team would collaborate using this workflow. We’ll see how two developers, John and Mary, can work on separate features and share their contributions via a centralized repository.

### John works on his feature



In his local repository, John can develop features using the standard Git commit process: edit, stage, and commit.

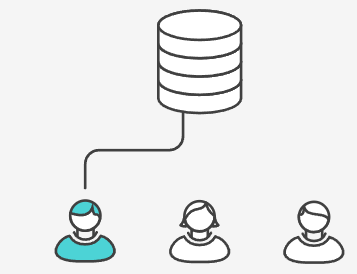
Remember that since these commands create local commits, John can repeat this process as many times as he wants without worrying about what’s going on in the central repository.

### Mary works on her feature



Meanwhile, Mary is working on her own feature in her own local repository using the same edit/stage/commit process. Like John, she doesn’t care what’s going on in the central repository, and she really doesn’t care what John is doing in his local repository, since all local repositories are private.

### John publishes his feature

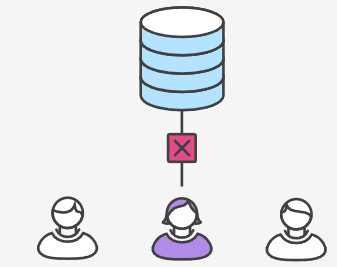


Once John finishes his feature, he should publish his local commits to the central repository so other team members can access it. He can do this with the [git push](https://www.atlassian.com/git/tutorials/syncing/git-push) command, like so:

git push origin main

Remember that origin is the remote connection to the central repository that Git created when John cloned it. The main argument tells Git to try to make the origin’s main branch look like his local main branch. Since the central repository hasn’t been updated since John cloned it, this won’t result in any conflicts and the push will work as expected.

### Mary tries to publish her feature



Let’s see what happens if Mary tries to push her feature after John has successfully published his changes to the central repository. She can use the exact same push command:

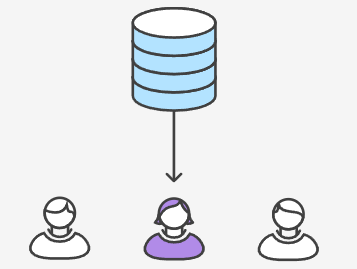
git push origin main

But, since her local history has diverged from the central repository, Git will refuse the request with a rather verbose error message:

error: failed to push some refs to '/path/to/repo.git'  
hint: Updates were rejected because the tip of your current branch is behind  
hint: its remote counterpart. Merge the remote changes (e.g. 'git pull')  
hint: before pushing again.  
hint: See the 'Note about fast-forwards' in 'git push --help' for details.

This prevents Mary from overwriting official commits. She needs to pull John’s updates into her repository, integrate them with her local changes, and then try again.

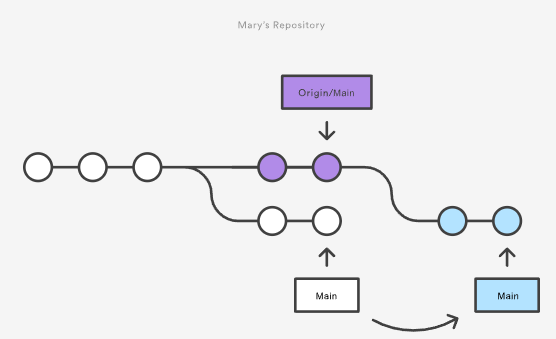
### Mary rebases on top of John’s commit(s)



Mary can use [git pull](https://www.atlassian.com/git/tutorials/syncing/git-pull) to incorporate upstream changes into her repository. This command is sort of like svn update—it pulls the entire upstream commit history into Mary’s local repository and tries to integrate it with her local commits:

git pull --rebase origin main

The --rebase option tells Git to move all of Mary’s commits to the tip of the main branch after synchronising it with the changes from the central repository, as shown below:



The pull would still work if you forgot this option, but you would wind up with a superfluous “merge commit” every time someone needed to synchronize with the central repository. For this workflow, it’s always better to rebase instead of generating a merge commit.

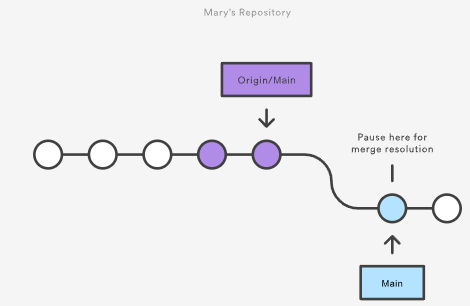
### Mary resolves a merge conflict



Rebasing works by transferring each local commit to the updated main branch one at a time. This means that you catch merge conflicts on a commit-by-commit basis rather than resolving all of them in one massive merge commit. This keeps your commits as focused as possible and makes for a clean project history. In turn, this makes it much easier to figure out where bugs were introduced and, if necessary, to roll back changes with minimal impact on the project.

If Mary and John are working on unrelated features, it’s unlikely that the rebasing process will generate conflicts. But if it does, Git will pause the rebase at the current commit and output the following message, along with some relevant instructions:

CONFLICT (content): Merge conflict in <some-file>



The great thing about Git is that anyone can resolve their own merge conflicts. In our example, Mary would simply run a [git status](https://www.atlassian.com/git/tutorials/inspecting-a-repository) to see where the problem is. Conflicted files will appear in the Unmerged paths section:

# Unmerged paths:  
# (use "git reset HEAD <some-file>..." to unstage)  
# (use "git add/rm <some-file>..." as appropriate to mark resolution)  
#  
# both modified: <some-file>

Then, she’ll edit the file(s) to her liking. Once she’s happy with the result, she can stage the file(s) in the usual fashion and let [git rebase](https://www.atlassian.com/git/tutorials/rewriting-history/git-rebase) do the rest:

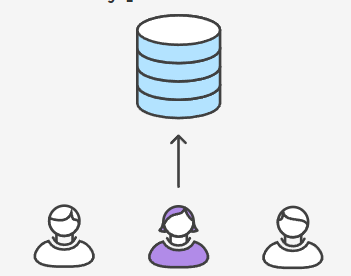
git add <some-file>  
git rebase --continue

And that’s all there is to it. Git will move on to the next commit and repeat the process for any other commits that generate conflicts.

If you get to this point and realize and you have no idea what’s going on, don’t panic. Just execute the following command and you’ll be right back to where you started:

git rebase --abort

### Mary successfully publishes her feature



After she’s done synchronizing with the central repository, Mary will be able to publish her changes successfully:

git push origin main

### Where to go from here

As you can see, it’s possible to replicate a traditional Subversion development environment using only a handful of Git commands. This is great for transitioning teams off of SVN, but it doesn’t leverage the distributed nature of Git.

The Centralized Workflow is great for small teams. The conflict resolution process detailed above can form a bottleneck as your team scales in size. If your team is comfortable with the Centralized Workflow but wants to streamline its collaboration efforts, it's definitely worth exploring the benefits of the [Feature Branch Workflow](https://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow). By dedicating an isolated branch to each feature, it’s possible to initiate in-depth discussions around new additions before integrating them into the official project.

## Other common workflows

The Centralized Workflow is essentially a building block for other Git workflows. Most popular Git workflows will have some sort of centralized repo that individual developers will push and pull from. Below we will briefly discuss some other popular Git workflows. These extended workflows offer more specialized patterns in regard to managing branches for feature development, hot fixes, and eventual release.

## Feature branching

Feature Branching is a logical extension of Centralized Workflow. The core idea behind the [Feature Branch Workflow](https://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow) is that all feature development should take place in a dedicated branch instead of the main branch. This encapsulation makes it easy for multiple developers to work on a particular feature without disturbing the main codebase. It also means the main branch should never contain broken code, which is a huge advantage for continuous integration environments.

## Gitflow Workflow

The [Gitflow Workflow](https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow) was first published in a highly regarded 2010 blog post from [Vincent Driessen at nvie](http://nvie.com/posts/a-successful-git-branching-model/). The Gitflow Workflow defines a strict branching model designed around the project release. This workflow doesn’t add any new concepts or commands beyond what’s required for the Feature Branch Workflow. Instead, it assigns very specific roles to different branches and defines how and when they should interact.

## Forking Workflow

The [Forking Workflow](https://www.atlassian.com/git/tutorials/comparing-workflows/forking-workflow) is fundamentally different than the other workflows discussed in this tutorial. Instead of using a single server-side repository to act as the “central” codebase, it gives every developer a server-side repository. This means that each contributor has not one, but two Git repositories: a private local one and a public server-side one.

## Guidelines

There is no one size fits all Git workflow. As previously stated, it’s important to develop a Git workflow that is a productivity enhancement for your team. In addition to team culture, a workflow should also complement business culture. Git features like branches and tags should complement your business’s release schedule. If your team is using [task tracking project management software](https://www.atlassian.com/software/jira) you may want to use branches that correspond with tasks in progress. In addition, some guidelines to consider when deciding on a workflow are:

### Short-lived branches

The longer a branch lives separate from the production branch, the higher the risk for merge conflicts and deployment challenges. Short-lived branches promote cleaner merges and deploys.

### Minimize and simplify reverts

It’s important to have a workflow that helps proactively prevent merges that will have to be reverted. A workflow that tests a branch before allowing it to be merged into the main branch is an example. However, accidents do happen. That being said, it’s beneficial to have a workflow that allows for easy reverts that will not disrupt the flow for other team members.

### Match a release schedule

A workflow should complement your business’s software development release cycle. If you plan to release multiple times a day, you will want to keep your main branch stable. If your release schedule is less frequent, you may want to consider using Git tags to tag a branch to a version.

## Summary

In this document we discussed Git workflows. We took an in-depth look at a Centralized Workflow with practical examples. Expanding on the Centralized Workflow we discussed additional specialized workflows. Some key takeaways from this document are:

* There is no one-size-fits-all Git workflow
* A workflow should be simple and enhance the productivity of your team
* Your business requirements should help shape your Git workflow

To read about the next Git workflow check out our comprehensive breakdown of the [Feature Branch Workflow](https://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow).

# Git Feature Branch Workflow

 The core idea behind the Feature Branch Workflow is that all feature development should take place in a dedicated branch instead of the main branch. This encapsulation makes it easy for multiple developers to work on a particular feature without disturbing the main codebase. It also means the main branch will never contain broken code, which is a huge advantage for continuous integration environments.

Encapsulating feature development also makes it possible to leverage pull requests, which are a way to initiate discussions around a branch. They give other developers the opportunity to sign off on a feature before it gets integrated into the official project. Or, if you get stuck in the middle of a feature, you can open a pull request asking for suggestions from your colleagues. The point is, pull requests make it incredibly easy for your team to comment on each other’s work.

The Git Feature Branch Workflow is a composable workflow that can be leveraged by other high-level Git workflows. We discussed other Git workflows on [the Git workflow overview page](https://www.atlassian.com/git/tutorials/comparing-workflows). Git Feature Branch Workflow is branching model focused, meaning that it is a guiding framework for managing and creating branches. Other workflows are more repo focused. The Git Feature Branch Workflow can be incorporated into other workflows. The [Gitflow](https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow), and [Git Forking Workflows](https://www.atlassian.com/git/tutorials/comparing-workflows/forking-workflow) traditionally use a Git Feature Branch Workflow in regards to their branching models.

## How it works

The Feature Branch Workflow assumes a central repository, and main represents the official project history. Instead of committing directly on their local main branch, developers create a new branch every time they start work on a new feature. Feature branches should have descriptive names, like animated-menu-items or issue-#1061. The idea is to give a clear, highly-focused purpose to each branch. Git makes no technical distinction between the main branch and feature branches, so developers can edit, stage, and commit changes to a feature branch.

In addition, feature branches can (and should) be pushed to the central repository. This makes it possible to share a feature with other developers without touching any official code. Since main is the only “special” branch, storing several feature branches on the central repository doesn’t pose any problems. Of course, this is also a convenient way to back up everybody’s local commits. The following is a walk-through of the life-cycle of a feature branch.

### Start with the main branch

All feature branches are created off the latest code state of a project. This guide assumes this is maintained and updated in the main branch.

git checkout main  
git fetch origin   
git reset --hard origin/main

This switches the repo to the main branch, pulls the latest commits and resets the repo's local copy of main to match the latest version.

### Create a new-branch

Use a separate branch for each feature or issue you work on. After creating a branch, check it out locally so that any changes you make will be on that branch.

git checkout -b new-feature

This checks out a branch called new-feature based on main, and the -b flag tells Git to create the branch if it doesn’t already exist.

### Update, add, commit, and push changes

On this branch, edit, stage, and commit changes in the usual fashion, building up the feature with as many commits as necessary. Work on the feature and make commits like you would any time you use Git. When ready, push your commits, updating the feature branch on Bitbucket.

git status  
git add <some-file>  
git commit

### Push feature branch to remote

It’s a good idea to push the feature branch up to the central repository. This serves as a convenient backup, when collaborating with other developers, this would give them access to view commits to the new branch.

git push -u origin new-feature

This command pushes new-feature to the central repository (origin), and the -u flag adds it as a remote tracking branch. After setting up the tracking branch, git push can be invoked without any parameters to automatically push the new-feature branch to the central repository. To get feedback on the new feature branch, create a pull request in a repository management solution like [Bitbucket Cloud](https://bitbucket.org/product) or [Bitbucket Data Center](https://www.atlassian.com/software/bitbucket/enterprise/data-center). From there, you can add reviewers and make sure everything is good to go before merging.

### Resolve feedback

Now teammates comment and approve the pushed commits. Resolve their comments locally, commit, and push the suggested changes to Bitbucket. Your updates appear in the pull request.

### Merge your pull request

Before you merge, you may have to resolve merge conflicts if others have made changes to the repo. When your pull request is approved and conflict-free, you can add your code to the main branch. Merge from the pull request in Bitbucket.

## Pull requests

Aside from isolating feature development, branches make it possible to discuss changes via pull requests. Once someone completes a feature, they don’t immediately merge it into main. Instead, they push the feature branch to the central server and file a pull request asking to merge their additions into main. This gives other developers an opportunity to review the changes before they become a part of the main codebase.

Code review is a major benefit of pull requests, but they’re actually designed to be a generic way to talk about code. You can think of pull requests as a discussion dedicated to a particular branch. This means that they can also be used much earlier in the development process. For example, if a developer needs help with a particular feature, all they have to do is file a pull request. Interested parties will be notified automatically, and they’ll be able to see the question right next to the relevant commits.

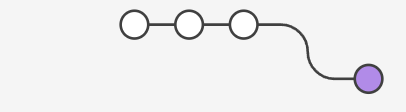
Once a pull request is accepted, the actual act of publishing a feature is much the same as in the [Centralized Workflow](https://www.atlassian.com/git/tutorials/comparing-workflows). First, you need to make sure your local main is synchronized with the upstream main. Then, you merge the feature branch into main and push the updated main back to the central repository.

Pull requests can be facilitated by product repository management solutions like Bitbucket Cloud or Bitbucket Server. View the Bitbucket Server pull requests documentation for an example.

## Example

The following is an example of the type of scenario in which a feature branching workflow is used. The scenario is that of a team doing code review around on a new feature pull request. This is one example of the many purposes this model can be used for.

### Mary begins a new feature



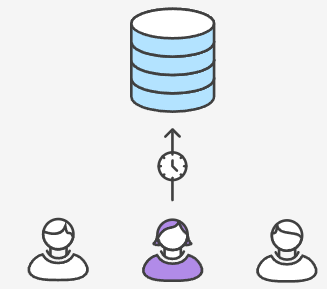
Before she starts developing a feature, Mary needs an isolated branch to work on. She can request a new branch with the following command:

git checkout -b marys-feature main

This checks out a branch called marys-feature based on main, and the -b flag tells Git to create the branch if it doesn’t already exist. On this branch, Mary edits, stages, and commits changes in the usual fashion, building up her feature with as many commits as necessary:

git status  
git add <some-file>  
git commit

### Mary goes to lunch

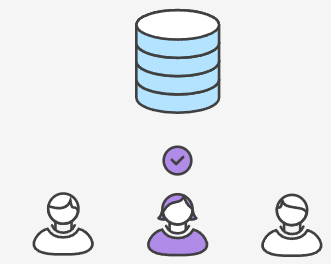


Mary adds a few commits to her feature over the course of the morning. Before she leaves for lunch, it’s a good idea to push her feature branch up to the central repository. This serves as a convenient backup, but if Mary was collaborating with other developers, this would also give them access to her initial commits.

git push -u origin marys-feature

This command pushes marys-feature to the central repository (origin), and the -u flag adds it as a remote tracking branch. After setting up the tracking branch, Mary can call git push without any parameters to push her feature.

### Mary finishes her feature

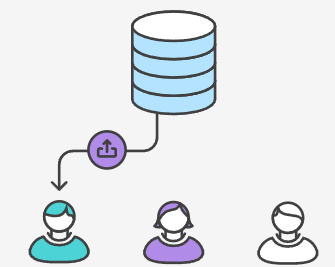


When Mary gets back from lunch, she completes her feature. Before merging it into main, she needs to file a pull request letting the rest of the team know she's done. But first, she should make sure the central repository has her most recent commits:

git push

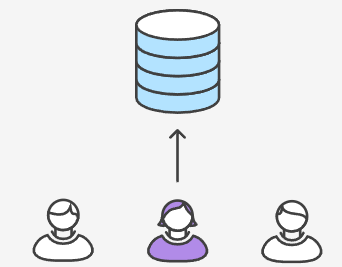
Then, she files the pull request in her Git GUI asking to merge marys-feature into main, and team members will be notified automatically. The great thing about pull requests is that they show comments right next to their related commits, so it's easy to ask questions about specific changesets.

### Bill receives the pull request



Bill gets the pull request and takes a look at marys-feature. He decides he wants to make a few changes before integrating it into the official project, and he and Mary have some back-and-forth via the pull request.

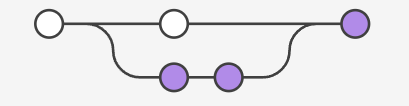
### Mary makes the changes



To make the changes, Mary uses the exact same process as she did to create the first iteration of her feature. She edits, stages, commits, and pushes updates to the central repository. All her activity shows up in the pull request, and Bill can still make comments along the way.

If he wanted, Bill could pull marys-feature into his local repository and work on it on his own. Any commits he added would also show up in the pull request.

### Mary publishes her feature



Once Bill is ready to accept the pull request, someone needs to merge the feature into the stable project (this can be done by either Bill or Mary):

git checkout main  
git pull  
git pull origin marys-feature  
git push

This process often results in a merge commit. Some developers like this because it’s like a symbolic joining of the feature with the rest of the code base. But, if you’re partial to a linear history, it’s possible to rebase the feature onto the tip of main before executing the merge, resulting in a fast-forward merge.

Some GUI’s will automate the pull request acceptance process by running all of these commands just by clicking an “Accept” button. If yours doesn’t, it should at least be able to automatically close the pull request when the feature branch gets merged into main.

Meanwhile, John is doing the exact same thing

While Mary and Bill are working on marys-feature and discussing it in her pull request, John is doing the exact same thing with his own feature branch. By isolating features into separate branches, everybody can work independently, yet it’s still trivial to share changes with other developers when necessary.

## Summary

In this document, we discussed the Git Feature Branch Workflow. This workflow helps organize and track branches that are focused on business domain feature sets. Other Git workflows like the Git Forking Workflow and the Gitflow Workflow are repo focused and can leverage the Git Feature Branch Workflow to manage their branching models. This document demonstrated a high-level code example and fictional example for implementing the Git Feature Branch Workflow. Some key associations to make with the Feature Branch Workflow are:

* focused on branching patterns
* can be leveraged by other repo oriented workflows
* promotes collaboration with team members through pull requests and merge reviews

Utilizing [git rebase](https://www.atlassian.com/git/tutorials/rewriting-history/git-rebase) during the review and merge stages of a feature branch will create enforce a cohesive Git history of feature merges. A feature branching model is a great tool to promote collaboration within a team environment.

Go one click deeper into Git workflows by reading our comprehensive tutorial of the [Gitflow Workflow.](https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow)

# Gitflow Workflow

Gitflow is a legacy Git workflow that was originally a disruptive and novel strategy for managing Git branches. Gitflow has fallen in popularity in favor of [trunk-based workflows](https://www.atlassian.com/continuous-delivery/continuous-integration/trunk-based-development), which are now considered best practices for modern continuous software development and [DevOps](https://www.atlassian.com/devops/what-is-devops) practices. Gitflow also can be challenging to use with [CI/CD](https://www.atlassian.com/continuous-delivery). This post details Gitflow for historical purposes.

## What is Gitflow?

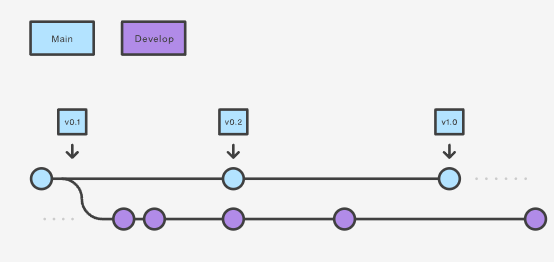
Giflow is an alternative Git branching model that involves the use of feature branches and multiple primary branches. It was first published and made popular by [Vincent Driessen at nvie](http://nvie.com/posts/a-successful-git-branching-model/). Compared to trunk-based development, Giflow has numerous, longer-lived branches and larger commits. Under this model, developers create a feature branch and delay merging it to the main trunk branch until the feature is complete. These long-lived feature branches require more collaboration to merge and have a higher risk of deviating from the trunk branch. They can also introduce conflicting updates.

Gitflow can be used for projects that have a scheduled release cycle and for the [DevOps best practice](https://www.atlassian.com/devops/what-is-devops/devops-best-practices) of [continuous delivery](https://www.atlassian.com/continuous-delivery). This workflow doesn’t add any new concepts or commands beyond what’s required for the [Feature Branch Workflow](https://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow). Instead, it assigns very specific roles to different branches and defines how and when they should interact. In addition to feature branches, it uses individual branches for preparing, maintaining, and recording releases. Of course, you also get to leverage all the benefits of the Feature Branch Workflow: pull requests, isolated experiments, and more efficient collaboration.

## Getting Started

Gitflow is really just an abstract idea of a Git workflow. This means it dictates what kind of branches to set up and how to merge them together. We will touch on the purposes of the branches below. The git-flow toolset is an actual command line tool that has an installation process. The installation process for git-flow is straightforward. Packages for git-flow are available on multiple operating systems. On OSX systems, you can execute brew install git-flow. On windows you will need to [download and install git-flow](https://git-scm.com/download/win). After installing git-flow you can use it in your project by executing git flow init. Git-flow is a wrapper around Git. The git flow init command is an extension of the default [git init](https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-init) command and doesn't change anything in your repository other than creating branches for you.

## How it works



### Develop and main branches

Instead of a single main branch, this workflow uses two branches to record the history of the project. The main branch stores the official release history, and the develop branch serves as an integration branch for features. It's also convenient to tag all commits in the main branch with a version number.

The first step is to complement the default main with a develop branch. A simple way to do this is for one developer to create an empty develop branch locally and push it to the server:

git branch develop  
git push -u origin develop

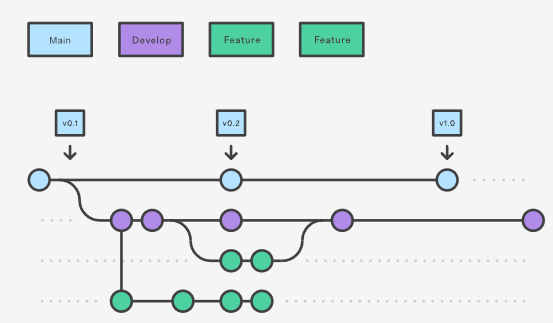
This branch will contain the complete history of the project, whereas main will contain an abridged version. Other developers should now clone the central repository and create a tracking branch for develop.

When using the git-flow extension library, executing git flow init on an existing repo will create the develop branch:

$ git flow init  
  
  
Initialized empty Git repository in ~/project/.git/  
No branches exist yet. Base branches must be created now.  
Branch name for production releases: [main]  
Branch name for "next release" development: [develop]  
  
  
How to name your supporting branch prefixes?  
Feature branches? [feature/]  
Release branches? [release/]  
Hotfix branches? [hotfix/]  
Support branches? [support/]  
Version tag prefix? []  
  
  
$ git branch  
\* develop  
 main

## Feature branches

Each new feature should reside in its own branch, which can be [pushed to the central repository](https://www.atlassian.com/git/tutorials/syncing/git-push) for backup/collaboration. But, instead of branching off of main, feature branches use develop as their parent branch. When a feature is complete, it gets [merged back into develop](https://www.atlassian.com/git/tutorials/using-branches/git-merge). Features should never interact directly with main.



Note that feature branches combined with the develop branch is, for all intents and purposes, the Feature Branch Workflow. But, the Gitflow workflow doesn’t stop there.

Feature branches are generally created off to the latest develop branch.

### Creating a feature branch

Without the git-flow extensions:

git checkout develop  
git checkout -b feature\_branch

When using the git-flow extension:

git flow feature start feature\_branch

Continue your work and use Git like you normally would.

### Finishing a feature branch

When you’re done with the development work on the feature, the next step is to merge the feature\_branch into develop.

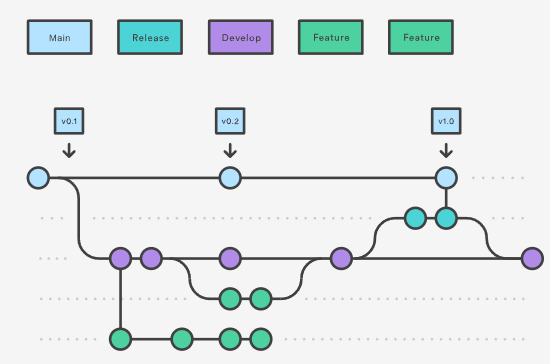
Without the git-flow extensions:

git checkout develop  
git merge feature\_branch

Using the git-flow extensions:

git flow feature finish feature\_branch

## Release branches



Once develop has acquired enough features for a release (or a predetermined release date is approaching), you fork a release branch off of develop. Creating this branch starts the next release cycle, so no new features can be added after this point—only bug fixes, documentation generation, and other release-oriented tasks should go in this branch. Once it's ready to ship, the release branch gets merged into main and tagged with a version number. In addition, it should be merged back into develop, which may have progressed since the release was initiated.

Using a dedicated branch to prepare releases makes it possible for one team to polish the current release while another team continues working on features for the next release. It also creates well-defined phases of development (e.g., it's easy to say, “This week we're preparing for version 4.0,” and to actually see it in the structure of the repository).

Making release branches is another straightforward branching operation. Like feature branches, release branches are based on the develop branch. A new release branch can be created using the following methods.

Without the git-flow extensions:

git checkout develop  
git checkout -b release/0.1.0

When using the git-flow extensions:

$ git flow release start 0.1.0  
Switched to a new branch 'release/0.1.0'

Once the release is ready to ship, it will get merged it into main and develop, then the release branch will be deleted. It’s important to merge back into develop because critical updates may have been added to the release branch and they need to be accessible to new features. If your organization stresses code review, this would be an ideal place for a pull request.

To finish a release branch, use the following methods:

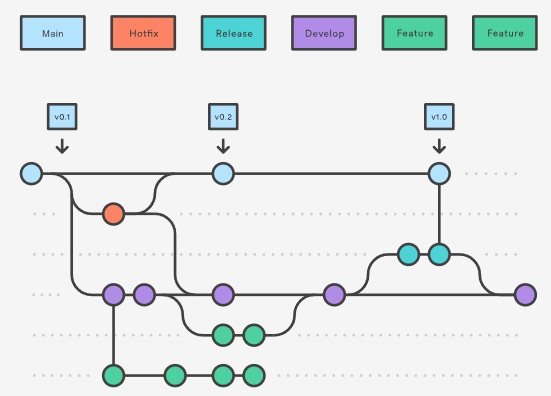
Without the git-flow extensions:

git checkout main  
git merge release/0.1.0

Or with the git-flow extension:

git flow release finish '0.1.0'

## Hotfix branches



Maintenance or “hotfix” branches are used to quickly patch production releases. Hotfix branches are a lot like release branches and feature branches except they're based on main instead of develop. This is the only branch that should fork directly off of main. As soon as the fix is complete, it should be merged into both main and develop (or the current release branch), and main should be tagged with an updated version number.

Having a dedicated line of development for bug fixes lets your team address issues without interrupting the rest of the workflow or waiting for the next release cycle. You can think of maintenance branches as ad hoc release branches that work directly with main. A hotfix branch can be created using the following methods:

Without the git-flow extensions:

git checkout main  
git checkout -b hotfix\_branch

When using the git-flow extensions:

$ git flow hotfix start hotfix\_branch

Similar to finishing a release branch, a hotfix branch gets merged into both main and develop.

git checkout main  
git merge hotfix\_branch  
git checkout develop  
git merge hotfix\_branch  
git branch -D hotfix\_branch

$ git flow hotfix finish hotfix\_branch

## Example

A complete example demonstrating a Feature Branch Flow is as follows. Assuming we have a repo setup with a main branch.

git checkout main  
git checkout -b develop  
git checkout -b feature\_branch  
# work happens on feature branch  
git checkout develop  
git merge feature\_branch  
git checkout main  
git merge develop  
git branch -d feature\_branch

In addition to the feature and release flow, a hotfix example is as follows:

git checkout main  
git checkout -b hotfix\_branch  
# work is done commits are added to the hotfix\_branch  
git checkout develop  
git merge hotfix\_branch  
git checkout main  
git merge hotfix\_branch

## Summary

Here we discussed the Gitflow Workflow. Gitflow is one of many styles of [Git workflows](https://www.atlassian.com/git/tutorials/comparing-workflows) you and your team can utilize.

Some key takeaways to know about Gitflow are:

* The workflow is great for a release-based software workflow.
* Gitflow offers a dedicated channel for hotfixes to production.

The overall flow of Gitflow is:

1. A develop branch is created from main
2. A release branch is created from develop
3. Feature branches are created from develop
4. When a feature is complete it is merged into the develop branch
5. When the release branch is done it is merged into develop and main
6. If an issue in main is detected a hotfix branch is created from main
7. Once the hotfix is complete it is merged to both develop and main

Next, learn about the [Forking Workflow](https://www.atlassian.com/git/tutorials/comparing-workflows/forking-workflow) or visit our [workflow comparison page](https://www.atlassian.com/git/tutorials/comparing-workflows).

# Forking Workflow

The Forking Workflow is fundamentally different than other popular Git workflows. Instead of using a single server-side repository to act as the “central” codebase, it gives every developer their own server-side repository. This means that each contributor has not one, but two Git repositories: a private local one and a public server-side one. The Forking Workflow is most often seen in public open source projects.

The main advantage of the Forking Workflow is that contributions can be integrated without the need for everybody to push to a single central repository. Developers push to their own server-side repositories, and only the project maintainer can push to the official repository. This allows the maintainer to accept commits from any developer without giving them write access to the official codebase.

The Forking Workflow typically follows a branching model based on the [Gitflow Workflow](https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow). This means that complete feature branches will be purposed for merge into the original project maintainer's repository. The result is a distributed workflow that provides a flexible way for large, organic teams (including untrusted third-parties) to collaborate securely. This also makes it an ideal workflow for open source projects.

## How it works

As in the other [Git workflows](https://www.atlassian.com/git/tutorials/comparing-workflows), the Forking Workflow begins with an official public repository stored on a server. But when a new developer wants to start working on the project, they do not directly clone the official repository.

Instead, they fork the official repository to create a copy of it on the server. This new copy serves as their personal public repository—no other developers are allowed to push to it, but they can pull changes from it (we’ll see why this is important in a moment). After they have created their server-side copy, the developer performs a [git clone](https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-clone) to get a copy of it onto their local machine. This serves as their private development environment, just like in the other workflows.

When they're ready to publish a local commit, they push the commit to their own public repository—not the official one. Then, they file a pull request with the main repository, which lets the project maintainer know that an update is ready to be integrated. The pull request also serves as a convenient discussion thread if there are issues with the contributed code. The following is a step-by-step example of this workflow.

1. A developer 'forks' an 'official' server-side repository. This creates their own server-side copy.
2. The new server-side copy is cloned to their local system.
3. A Git remote path for the 'official' repository is added to the local clone.
4. A new local feature branch is created.
5. The developer makes changes on the new branch.
6. New commits are created for the changes.
7. The branch gets pushed to the developer's own server-side copy.
8. The developer opens a pull request from the new branch to the 'official' repository.
9. The pull request gets approved for merge and is merged into the original server-side repository

To integrate the feature into the official codebase, the maintainer pulls the contributor’s changes into their local repository, checks to make sure it doesn’t break the project, merges it into their local main branch, then pushes the main branch to the official repository on the server. The contribution is now part of the project, and other developers should pull from the official repository to synchronize their local repositories.

It’s important to understand that the notion of an “official” repository in the Forking Workflow is merely a convention. In fact, the only thing that makes the official repository so official is that it’s the public repository of the project maintainer.

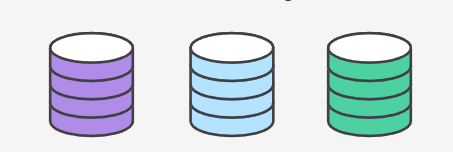
## Forking vs cloning

It's important to note that "forked" repositories and "forking" are not special operations. Forked repositories are created using the standard [git clone](https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-clone) command. Forked repositories are generally "server-side clones" and usually managed and hosted by a 3rd party Git service like [Bitbucket](https://bitbucket.org/product). There is no unique Git command to create forked repositories. A clone operation is essentially a copy of a repository and its history.

## Branching in the Forking Workflow

All of these personal public repositories are really just a convenient way to share branches with other developers. Everybody should still be using branches to isolate individual features, just like in the [Feature Branch Workflow](https://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow) and the [Gitflow Workflow.](https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow) The only difference is how those branches get shared. In the Forking Workflow, they are pulled into another developer’s local repository, while in the Feature Branch and Gitflow Workflows they are pushed to the official repository.

## Fork a repository



All new developers to a Forking Workflow project need to fork the official repository. As previously stated, forking is just a standard git clone operation. It’s possible to do this by SSH’ing into the server and running git clone to copy it to another location on the server. Popular Git hosting services like Bitbucket, offer repo forking features that automate this step.

## Clone your fork

Next each developer needs to clone their own public forked repository. They can do this with the familiar git clone command.

Assuming the use of Bitbucket to host these repositories, developers on a project should have their own Bitbucket account and they should clone their forked copy of the repository with:

git clone https://user@bitbucket.org/user/repo.git

## Adding a remote

Whereas other Git workflows use a single origin remote that points to the central repository, the Forking Workflow requires two remotes—one for the official repository, and one for the developer’s personal server-side repository. While you can call these remotes anything you want, a common convention is to use origin as the remote for your forked repository (this will be created automatically when you run git clone) and upstream for the official repository.

git remote add upstream https://bitbucket.org/maintainer/repo

You’ll need to create the upstream remote yourself using the above command. This will let you easily keep your local repository up-to-date as the official project progresses. Note that if your upstream repository has authentication enabled (i.e., it's not open source), you'll need to supply a username, like so:

git remote add upstream https://user@bitbucket.org/maintainer/repo.git

This requires users to supply a valid password before cloning or pulling from the official codebase.

## Working in a branch: making & pushing changes

In the developer's local copy of the forked repository they can edit code, commit changes, and create branches just like in other Git workflows:

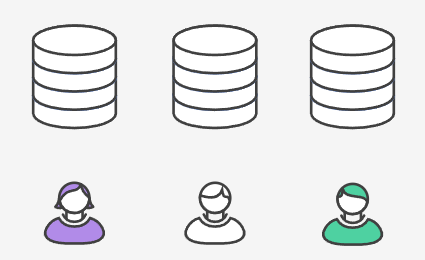
git checkout -b some-feature # Edit some code git commit -a -m "Add first draft of some feature"

All of their changes will be entirely private until they push it to their public repository. And, if the official project has moved forward, they can access new commits with git pull:

git pull upstream main

Since developers should be working in a dedicated feature branch, this should generally result in a fast-forward merge.

## Making a Pull Request



Once a developer is ready to share their new feature, they need to do two things. First, they have to make their contribution accessible to other developers by pushing it to their public repository. Their origin remote should already be set up, so all they should have to do is the following:

git push origin feature-branch

This diverges from the other workflows in that the origin remote points to the developer’s personal server-side repository, not the main codebase.

Second, they need to notify the project maintainer that they want to merge their feature into the official codebase. Bitbucket provides a “pull request” button that leads to a form asking you to specify which branch you want to merge into the official repository. Typically, you’ll want to integrate your feature branch into the upstream remote’s main branch.

## Summary

To recap, the Forking Workflow is commonly used in public open-source projects. Forking is a git clone operation executed on a server copy of a projects repo. A Forking Workflow is often used in conjunction with a Git hosting service like Bitbucket. A high-level example of a Forking Workflow is:

1. You want to contribute to an open source library hosted at bitbucket.org/userA/open-project
2. Using Bitbucket you create a fork of the repo to bitbucket.org/YourName/open-project
3. On your local system you execute git clone on https://bitbucket.org/YourName/open-project to get a local copy of the repo
4. You create a new feature branch in your local repo
5. Work is done to complete the new feature and git commit is executed to save the changes
6. You then push the new feature branch to your remote forked repo
7. Using Bitbucket you open up a pull request for the new branch against the original repo at bitbucket.org/userA/open-project

The Forking Workflow helps a maintainer of a project open up the repository to contributions from any developer without having to manually manage authorization settings for each individual contributor. This gives the maintainer more of a "pull" style workflow. Most commonly used in open-source projects, the Forking Workflow can also be applied to private business workflows to give more authoritative control over what is merged into a release. This can be useful in teams that have Deploy Managers or strict release cycles.

Unsure what workflow is right for you? Check out our comprehensive [Git workflow comparison page.](https://www.atlassian.com/git/tutorials/comparing-workflows)