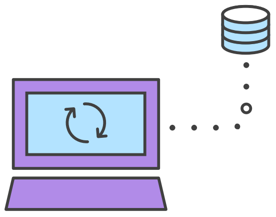


Syncing

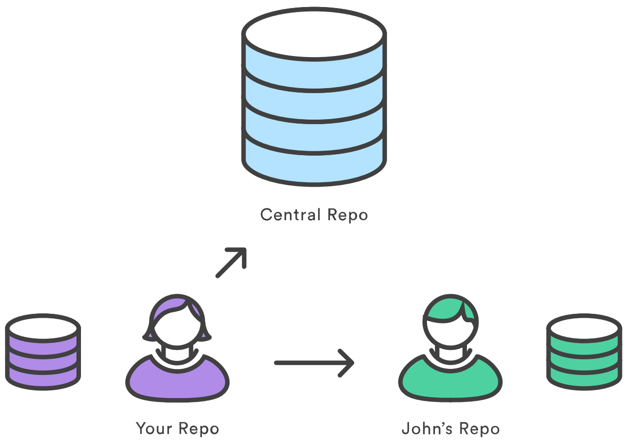
SVN uses a single central repository to serve as the communication hub for developers, and collaboration takes place by passing changesets between the developers' working copies and the central repository. This is different from Git's collaboration model, which gives every developer their own copy of 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 commands presented below let you manage connections with other repositories, publish local history by "pushing" branches to other repositories, and see what others have contributed by "pulling" branches into your local repository.

Tutorials

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.



Usage

```
git remote
```

List the remote connections you have to other repositories.

```
git remote -v
```

Tutorials

```
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>.

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

Tutorials

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.

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.gi
```

teams working on a large project.

git fetch

The `git fetch` command imports commits from a remote repository into your local repo. The resulting commits are stored as remote branches instead of the normal local branches that we've been working with. This gives you a chance to review changes before integrating them into your copy of the project.

Usage

```
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.

Discussion

Fetching is what you do when you want to see what everybody else has been working on. Since fetched content is represented as a remote branch, it has absolutely no effect on your local development work.

Tutorials

central history has progressed, but it doesn't force you to actually merge the changes into your repository.

Remote Branches

Remote branches are just like local branches, except they represent commits from somebody else's repository. 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. Remote branches are prefixed by the remote they belong to so that you don't mix them up with local branches. For example, the next code snippet shows the branches you might see after fetching from the origin remote:

```
git branch -r
# origin/master
# origin/develop
# origin/some-feature
```

Again, you can inspect these 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.

Examples

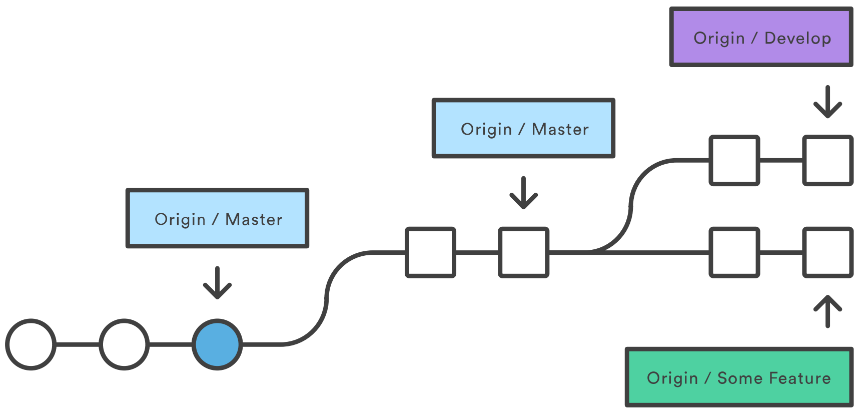
Tutorials

```
git fetch origin
```

This will display the branches that were downloaded:

```
a1e8fb5..45e66a4 master -> origin/master
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 master, you can run a `git log` using `origin/master` as a filter

```
git log --oneline master..origin/master
```

To approve the changes and merge them into your local `master` branch with the following commands:

```
git checkout master
```

Tutorials

```
git merge origin/master
```

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

git pull

Merging upstream changes into your local repository is a common task in Git-based collaboration workflows. We already know how to do this with `git fetch` followed by `git merge`, but `git pull` rolls this into a single command.

Usage

```
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>`.

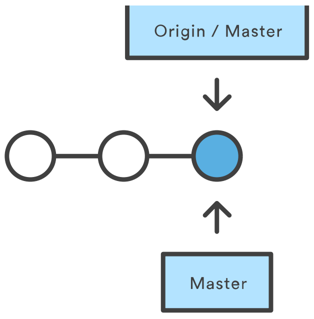
```
git pull --rebase <remote>
```

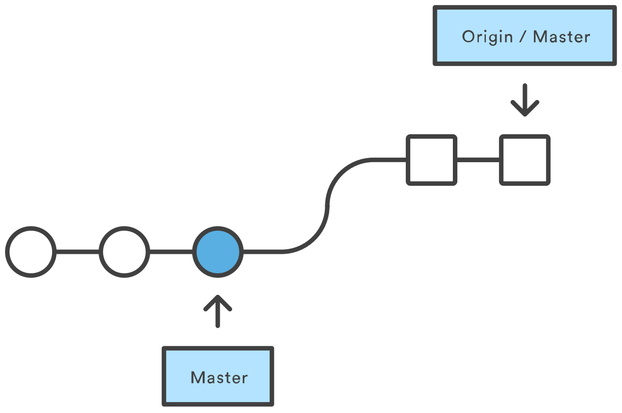
Same as the above command, but instead of using `git merge` to integrate the remote branch with the local one, use `git rebase`.

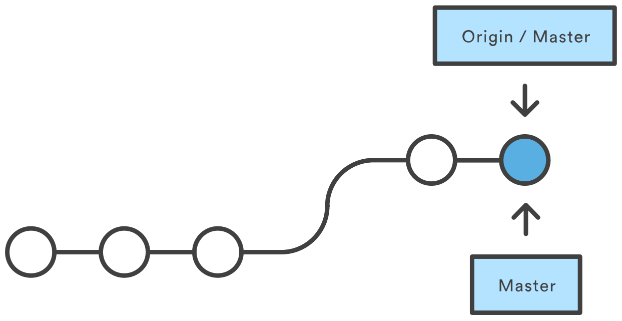
Tutorials

`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.

Tutorials







You start out thinking your repository is synchronized, but then `git fetch` reveals that origin's version of

Tutorials

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`.

Examples

The following example demonstrates how to synchronize with the central repository's `master branch`:

```
git checkout master
git pull --rebase origin
```

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

Tutorials

Pushing is how you transfer commits from your local repository to a remote repo. It's the counterpart to `git fetch`, but whereas fetching imports commits to local branches, pushing exports commits to remote branches. This has the potential to overwrite changes, so you need to be careful how you use it. These issues are discussed below.

Usage

```
git push <remote> <branch>
```

Push the specified branch to <remote>, 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
```

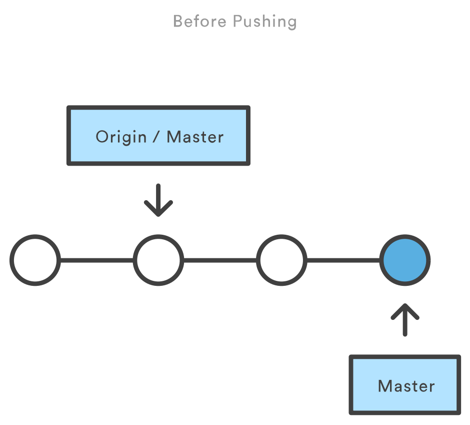
Push all of your local branches to the specified remote.

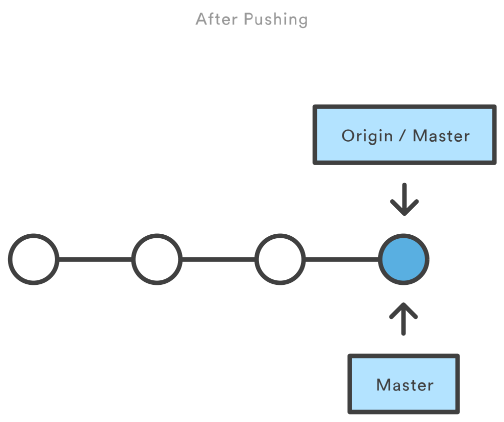
```
git push <remote> --tags
```

Tutorials

Discussion

The most common use case for `git push` is to publish your local changes to a central repository. After you've accumulated several local commits and are ready to share them with the rest of the team, you (optionally) clean them up with an interactive rebase, then push them to the central repository.





The above diagram shows what happens when your essentially the same as running `git merge master` from inside the remote repository.

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.

Only Push to Bare Repositories

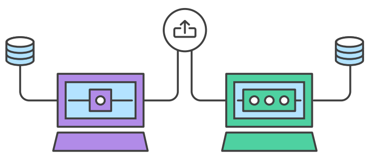
In addition, you should only push to repositories that have been created with the `--bare` flag. Since pushing messes with the remote branch structure, it's important to never push to another developer's repository. But because bare repos don't have a working directory, it's impossible to interrupt anybody's developments.

Tutorials

methods for publishing local contributions to the central repository. First, it makes sure your local `master` 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 `master` to the central repository.

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

Since we already made sure the local `master` 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.



Next up:

Tutorials

START NEXT TUTORIAL

Powered By

Tutorials

Enter Your Email For Git News