Setting up a repository

git init / git clone / git config

This tutorial provides an overview of how to set up a repository (repo) under Git version control. This resource will walk you through initializing a Git repository for a new or existing project. Included below are workflow examples of repositories both created locally and cloned from remote repositories. This guide assumes a basic familiarity with a command-line interface.

The high level points this guide will cover are:

- Initializing a new Git repo
- Cloning an existing Git repo

- Common Git version control commands

By the end of this module, you should be able to create a Git repo, use common Git commands, commit a modified file, view your project's history and configure a connection to a Git hosting service (Bitbucket).

What is a Git repository?

A Git repository is a virtual storage of your project. It allows you to save versions of your code, which you can access when needed.

Initializing a new repository: git init

To create a new repo, you'll use the `git init` command. `git init` is a one-time command you use during the initial setup of a new repo. Executing this command will create a new `.git` subdirectory in your current working directory. This will also create a new master branch.

Versioning an existing project with a new git repository

the `git init` command.

```
cd /path/to/your/existing/code
git init
```

Pointing `git init` to an existing project directory will execute the same initialization setup as mentioned above, but scoped to that project directory.

```
git init <project directory>
```

Visit the git init page for a more detailed resource on `git init`.

Cloning an existing repository: git clone

If a project has already been set up in a central repository, the clone command is the most common way for users to obtain a local development clone. Like `git init`, cloning is generally a one-time operation. Once a developer has obtained a working copy, all version control operations are managed through their local repository.

```
git clone <repo url>
```

corresponding URL formats. In this example, we'll be using the Git SSH protocol. Git SSH URLs follow a template of: `git@HOSTNAME:USERNAME/REPONAME.git`

An example Git SSH URL would be: `git@bitbucket.org:rhyolight/javascript-data-store.git` where the template values match:

- `HOSTNAME: bitbucket.org`
- `USERNAME: rhyolight`
- `REPONAME: javascript-data-store`

When executed, the latest version of the remote repo files on the master branch will be pulled down and added to a new folder. The new folder will be named after the REPONAME in this case `javascript-data-store`. The folder will contain the full history of the remote repository and a newly created master branch.

For more documentation on `git clone` usage and supported Git URL formats, visit the git clone Page.

Saving changes to the repository: git add and git commit

`/path/to/project`. The steps being taken in this example are:

- Change directories to `/path/to/project`
- Create a new file `CommitTest.txt` with contents ~"test content for git tutorial"~
- git add `CommitTest.txt` to the repository staging area
- Create a new commit with a message describing what work was done in the commit

```
cd /path/to/project
echo "test content for git tutorial" >> CommitT
git add CommitTest.txt
git commit -m "added CommitTest.txt to the repo
```

After executing this example, your repo will now have `CommitTest.txt` added to the history and will track future updates to the file.

This example introduced two additional git commands: `add` and `commit`. This was a very limited example, but both commands are covered more in depth on the git add and git commit pages. Another common use case for `git add` is the `--all` option. Executing `git add --all` will take any changed and untracked files in the repo and add them to the repo and update the repo's working tree.

collaboration: git push

It's important to understand that Git's idea of a "working copy" is very different from the working copy you get by checking out source code from an SVN repository. Unlike SVN, Git makes no distinction between the working copies and the central repository—they're all full-fledged Git repositories.

This makes collaborating with Git fundamentally different than with SVN. Whereas SVN depends on the relationship between the central repository and the working copy, Git's collaboration model is based on repository-to-repository interaction. Instead of checking a working copy into SVN's central repository, you push or pull commits from one repository to another.

Of course, there's nothing stopping you from giving certain Git repos special meaning. For example, by simply designating one Git repo as the "central" repository, it's possible to replicate a centralized workflow using Git. This is accomplished through conventions rather than being hardwired into the VCS itself.

Bare vs. cloned repositories

If you used `git clone` in the previous "Initializing a new Repository" section to set up your local repository, your repository is already configured for remote collaboration. `git clone` will automatically configure your repo with a remote pointed to the Git URL you cloned it from. This

If you used `git init` to make a fresh repo, you'll have no remote repo to push changes to. A common pattern when initializing a new repo is to go to a hosted Git service like Bitbucket and create a repo there. The service will provide a Git URL that you can then add to your local Git repository and `git push` to the hosted repo. Once you have created a remote repo with your service of choice you will need to update your local repo with a mapping. We discuss this process in the Configuration & Set Up guide below.

If you prefer to host your own remote repo, you'll need to set up a "Bare Repository." Both `git init` and `git clone` accept a `--bare` argument. The most common use case for bare repo is to create a remote central Git repository

Configuration & set up: git config

Once you have a remote repo setup, you will need to add a remote repo url to your local `git config`, and set an upstream branch for your local branches. The `git remote` command offers such utility.

```
git remote add <remote_name> <remote_repo_url>
```

you can push local branches to it.

```
git push -u <remote_name> <local_branch_name>
```

This command will push the local repo branch under `<local_branc_name>` to the remote repo at `<remote_name>`.

For more in-depth look at `git remote`, see the `Git remote page`.

In addition to configuring a remote repo URL, you may also need to set global Git configuration options such as username, or email. The `git config` command lets you configure your Git installation (or an individual repository) from the command line. This command can define everything from user info, to preferences, to the behavior of a repository. Several common configuration options are listed below.

Git stores configuration options in three separate files, which lets you scope options to individual repositories (local), user (Global), or the entire system (system):

- Local: `<repo>/.git/config` – Repository-specific settings.
- Global: `/.gitconfig` – User-specific settings. This is where options set with the `--global` flag are stored.

Define the author name to be used for all commits in the current repository. Typically, you'll want to use the `--global` flag to set configuration options for the current user.

```
git config --global user.name <name>
```

Define the author name to be used for all commits by the current user.

Adding the `--local` option or not passing a config level option at all, will set the `user.name` for the current local repository.

```
git config --local user.email <email>
```

Define the author email to be used for all commits by the current user.

```
git config --global alias.<alias-name> <git-cor
```

Create a shortcut for a Git command. This is a powerful utility to create custom shortcuts for commonly used git commands. A simplistic example would be:

```
git config --global alias.ci commit
```

shortcut to `git commit`. To learn more about git aliases visit the `git config` page.

```
git config --system core.editor <editor>
```

Define the text editor used by commands like `git commit` for all users on the current machine. The `<editor>` argument should be the command that launches the desired editor (e.g., vi). This example introduces the `--system` option. The `--system` option will set the configuration for the entire system, meaning all users and repos on a machine. For more detailed information on configuration levels visit the git config page.

```
git config --global --edit
```

Open the global configuration file in a text editor for manual editing. An in-depth guide on how to configure a text editor for git to use can be found on the Git config page.

Discussion

All configuration options are stored in plaintext files, so the `git config` command is really just a convenient command-line interface. Typically, you'll only need to configure a Git installation the first time you start working on a new development machine, and for virtually all

for personal and open source repositories, and your professional email address for work-related repositories.

Git stores configuration options in three separate files, which lets you scope options to individual repositories, users, or the entire system:

- `<repo>/.git/config` – Repository-specific settings.
- `~/.gitconfig` – User-specific settings. This is where options set with the `--global` flag are stored.
- `$(prefix)/etc/gitconfig` – System-wide settings.

When options in these files conflict, local settings override user settings, which override system-wide. If you open any of these files, you'll see something like the following:

```
[user] name = John Smith email = john@example.c
```

You can manually edit these values to the exact same effect as `git config`.

Example

The first thing you'll want to do after installing Git is tell it your name/email and customize some of the default settings. A typical initial configuration might look something like the following:

```
git --global user.name "John Smith" git ...
```

Select your favorite text editor

```
git config --global core.editor vim
```

Add some SVN-like aliases

```
git config --global alias.st status
git config --global alias.co checkout
git config --global alias.br branch
git config --global alias.up rebase
git config --global alias.ci commit
```

This will produce the `~/.gitconfig` file from the previous section. Take a more in-depth look at git config on the git config page.

Summary

Here we demonstarted how to create a git repository using two methods: git init and git clone. This guide can be applied to manage software source code or other content that needs to be versioned. Git add, git commit, git push, and git remote were also introduced and utilized at a high level.

Ready to learn Git?

Try this interactive tutorial.

Get started now

Next up:

git init

START NEXT TUTORIAL

Ready to learn Git?

Try this interactive tutorial.

Get started now

Recommend

Want future articles?

Site hosted by

Enter Your Email For Git News