Chapter 1: Getting started with Git

Remarks

Git is a free, distributed version control system which allows programmers to keep track of code changes, via "snapshots" (commits), in its current state. Utilizing commits allows programmers to test, debug, and create new features collaboratively. All commits are kept in what is known as a "Git Repository" that can be hosted on your computer, private servers, or open source websites, such at Github.

Git also allows for users to create new "branches" of the code, which allows different versions of the code to live alongside each other. This enables scenarios where one branch contains the most recent stable version, a different branch contains a set of new features being developed, and a yet another branch contains a different set of features. Git makes the process of creating these branches, and then subsequently merging them back together, nearly painless.

Git has 3 different "areas" for your code:

- **Working directory**: The area that you will be doing all of your work in (creating, editing, deleting, and organizing files)
- **Staging area**: The area where you will list the changes that you have made to the working directory
- **Repository**: Where Git permanently stores the changes you have made as different versions of the project

Git was originally created for managing the Linux kernel source. By making them easier, it encourages small commits, forking of projects and merging between forks, and having lots of short-lived branches.

The biggest change for people who are used to CVS or Subversion is that every checkout contains not only the source tree, but also the whole history of the project. Common operations like diffing of revisions, checking out older revisions, committing (to your local history), creating a branch, checking out a different branch, merging branches or patch files can all be done locally without having to communicate with a central server. Thus the biggest source of latency and unreliability is removed. Communicating with the "upstream" repository is only needed to get the latest changes, and to publish your local changes to other developers. This turns what was previously a technical constraint (whoever has the repository owns the project) into an organisational choice (your "upstream" is whomever you choose to sync with).

Versions

Version	Release Date
2.13	2017-05-10
2.12	2017-02-24

Version	Release Date
2.11.1	2017-02-02
2.11	2016-11-29
2.10.2	2016-10-28
2.10	2016-09-02
2.9	2016-06-13
2.8	2016-03-28
2.7	2015-10-04
2.6	2015-09-28
2.5	2015-07-27
2.4	2015-04-30
2.3	2015-02-05
2.2	2014-11-26
2.1	2014-08-16
2.0	2014-05-28
1.9	2014-02-14
1.8.3	2013-05-24
1.8	2012-10-21
1.7.10	2012-04-06
1.7	2010-02-13
1.6.5	2009-10-10
1.6.3	2009-05-07
1.6	2008-08-17
1.5.3	2007-09-02
1.5	2007-02-14
1.4	2006-06-10

Version	Release Date
1.3	2006-04-18
1.2	2006-02-12
1.1	2006-01-08
1.0	2005-12-21
0.99	2005-07-11

Examples

Create your first repository, then add and commit files

At the command line, first verify that you have Git installed:

On all operating systems:

```
git --version
```

On UNIX-like operating systems:

```
which git
```

If nothing is returned, or the command is not recognized, you may have to install Git on your system by downloading and running the installer. See the Git homepage for exceptionally clear and easy installation instructions.

After installing Git, configure your username and email address. Do this *before* making a commit.

Once Git is installed, navigate to the directory you want to place under version control and create an empty Git repository:

```
git init
```

This creates a hidden folder, `.git`, which contains the plumbing needed for Git to work.

Next, check what files Git will add to your new repository; this step is worth special care:

```
git status
```

Review the resulting list of files; you can tell Git which of the files to place into version control (avoid adding files with confidential information such as passwords, or files that just clutter the repo):

```
git add <file/directory name #1> <file/directory name #2> < ... >
```

If all files in the list should be shared with everyone who has access to the repository, a single command will add everything in your current directory and its subdirectories:

```
git add .
```

This will "stage" all files to be added to version control, preparing them to be committed in your first commit.

For files that you want never under version control, create and populate a file named `.gitignore` before running the `add` command.

Commit all the files that have been added, along with a commit message:

```
git commit -m "Initial commit"
```

This creates a new commit with the given message. A commit is like a save or snapshot of your entire project. You can now push, or upload, it to a remote repository, and later you can jump back to it if necessary.
If you omit the `-m` parameter, your default editor will open and you can edit and save the commit message there.

Adding a remote

To add a new remote, use the `git remote add` command on the terminal, in the directory your repository is stored at.

The `git remote add` command takes two arguments:

1. A remote name, for example, `origin`
2. A remote URL, for example, `https://<your-git-service-address>/user/repo.git`

```
    git remote add origin https://<your-git-service-address>/owner/repository.git
```

NOTE: Before adding the remote you have to create the required repository in your git service, You'll be able to push/pull commits after adding your remote.

Clone a repository

The `git clone` command is used to copy an existing Git repository from a server to the local machine.

For example, to clone a GitHub project:

```
cd <path where you'd like the clone to create a directory>
git clone https://github.com/username/projectname.git
```

To clone a BitBucket project:

```
cd <path where you'd like the clone to create a directory>
git clone https://yourusername@bitbucket.org/username/projectname.git
```

This creates a directory called `projectname` on the local machine, containing all the files in the remote Git repository. This includes source files for the project, as well as a `.git` sub-directory which contains the entire history and configuration for the project.

To specify a different name of the directory, e.g. `MyFolder`:

```
git clone https://github.com/username/projectname.git MyFolder
```

Or to clone in the current directory:

```
git clone https://github.com/username/projectname.git .
```

Note:

1. When cloning to a specified directory, the directory must be empty or non-existent.
2. You can also use the `ssh` version of the command:

   ```
   git clone git@github.com:username/projectname.git
   ```

The `https` version and the `ssh` version are equivalent. However, some hosting services such as GitHub recommend that you use `https` rather than `ssh`.

Setting up the upstream remote

If you have cloned a fork (e.g. an open source project on Github) you may not have push access to the upstream repository, so you need both your fork but be able to fetch the upstream repository.

First check the remote names:

```
$ git remote -v
origin    https://github.com/myusername/repo.git (fetch)
origin    https://github.com/myusername/repo.git (push)
upstream  # this line may or may not be here
```

If `upstream` is there already (it is on *some* Git versions) you need to set the URL (currently it's empty):

```
$ git remote set-url upstream https://github.com/projectusername/repo.git
```

If the upstream is **not** there, or if you also want to add a friend/colleague's fork (currently they do not exist):

```
$ git remote add upstream https://github.com/projectusername/repo.git
```

```
$ git remote add dave https://github.com/dave/repo.git
```

Sharing code

To share your code you create a repository on a remote server to which you will copy your local repository.

To minimize the use of space on the remote server you create a bare repository: one which has only the `.git` objects and doesn't create a working copy in the filesystem. As a bonus you set this remote as an upstream server to easily share updates with other programmers.

On the remote server:

```
git init --bare /path/to/repo.git
```

On the local machine:

```
git remote add origin ssh://username@server:/path/to/repo.git
```

(Note that `ssh:` is just one possible way of accessing the remote repository.)

Now copy your local repository to the remote:

```
git push --set-upstream origin master
```

Adding `--set-upstream` (or `-u`) created an upstream (tracking) reference which is used by argument-less Git commands, e.g. `git pull`.

Setting your user name and email

```
You need to set who you are *before* creating any commit.  That will allow commits to have the
right author name and email associated to them.
```

It has nothing to do with authentication when pushing to a remote repository (e.g. when pushing to a remote repository using your GitHub, BitBucket, or GitLab account)

To declare that identity for *all* repositories, use `git config --global`
This will store the setting in your user's `.gitconfig` file: e.g. `$HOME/.gitconfig` or for Windows, `%USERPROFILE%\.gitconfig`.

```
git config --global user.name "Your Name"
git config --global user.email mail@example.com
```

To declare an identity for a single repository, use `git config` inside a repo.
This will store the setting inside the individual repository, in the file `$GIT_DIR/config`. e.g. `/path/to/your/repo/.git/config`.

```
cd /path/to/my/repo
git config user.name "Your Login At Work"
git config user.email mail_at_work@example.com
```

Settings stored in a repository's config file will take precedence over the global config when you use that repository.

Tips: if you have different identities (one for open-source project, one at work, one for private repos, ...), and you don't want to forget to set the right one for each different repos you are working on:

- **Remove a global identity**

  ```
  git config --global --remove-section user.name
  git config --global --remove-section user.email
  ```

2.8

- To force git to look for your identity only within a repository's settings, not in the global config:

  ```
  git config --global user.useConfigOnly true
  ```

That way, if you forget to set your `user.name` and `user.email` for a given repository and try to make a commit, you will see:

```
no name was given and auto-detection is disabled
no email was given and auto-detection is disabled
```

Learning about a command

To get more information about any git command – i.e. details about what the command does, available options and other documentation – use the `--help` option or the `help` command.

For example, to get all available information about the `git diff` command, use:

```
git diff --help
git help diff
```

Similarly, to get all available information about the `status` command, use:

```
git status --help
git help status
```

If you only want a quick help showing you the meaning of the most used command line flags, use `-h`:

```
git checkout -h
```

Set up SSH for Git

If you are using **Windows** open Git Bash. If you are using **Mac** or **Linux** open your Terminal.

Before you generate an SSH key, you can check to see if you have any existing SSH keys.

List the contents of your `~/.ssh` directory:

```
$ ls -al ~/.ssh
# Lists all the files in your ~/.ssh directory
```

Check the directory listing to see if you already have a public SSH key. By default the filenames of the public keys are one of the following:

```
id_dsa.pub
id_ecdsa.pub
id_ed25519.pub
id_rsa.pub
```

If you see an existing public and private key pair listed that you would like to use on your Bitbucket, GitHub (or similar) account you can copy the contents of the `id_*.pub` file.

If not, you can create a new public and private key pair with the following command:

```
$ ssh-keygen
```

Press the Enter or Return key to accept the default location. Enter and re-enter a passphrase when prompted, or leave it empty.

Ensure your SSH key is added to the ssh-agent. Start the ssh-agent in the background if it's not already running:

```
$ eval "$(ssh-agent -s)"
```

Add you SSH key to the ssh-agent. Notice that you'll need te replace `id_rsa` in the command with the name of your **private key file**:

```
$ ssh-add ~/.ssh/id_rsa
```

If you want to change the upstream of an existing repository from HTTPS to SSH you can run the following command:

```
$ git remote set-url origin ssh://git@bitbucket.server.com:7999/projects/your_project.git
```

In order to clone a new repository over SSH you can run the following command:

```
$ git clone ssh://git@bitbucket.server.com:7999/projects/your_project.git
```

Git Installation

Let's get into using some Git. First things first—you have to install it. You can get it a number of ways; the two major ones are to install it from source or to install an existing package for your platform.

Installing from Source

If you can, it's generally useful to install Git from source, because you'll get the most recent version. Each version of Git tends to include useful UI enhancements, so getting the latest version is often the best route if you feel comfortable compiling software from source. It is also the case that many Linux distributions contain very old packages; so unless you're on a very up-to-date distro or are using backports, installing from source may be the best bet.

To install Git, you need to have the following libraries that Git depends on: curl, zlib, openssl, expat, and libiconv. For example, if you're on a system that has yum (such as Fedora) or apt-get (such as a Debian based system), you can use one of these commands to install all of the dependencies:

```
$ yum install curl-devel expat-devel gettext-devel \
  openssl-devel zlib-devel

$ apt-get install libcurl4-gnutls-dev libexpat1-dev gettext \
  libz-dev libssl-dev
```

When you have all the necessary dependencies, you can go ahead and grab the latest snapshot from the Git web site:

http://git-scm.com/download Then, compile and install:

```
$ tar -zxf git-1.7.2.2.tar.gz
$ cd git-1.7.2.2
$ make prefix=/usr/local all
$ sudo make prefix=/usr/local install
```

After this is done, you can also get Git via Git itself for updates:

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
```

Installing on Linux

If you want to install Git on Linux via a binary installer, you can generally do so through the basic package-management tool that comes with your distribution. If you're on Fedora, you can use yum:

```
$ yum install git
```

Or if you're on a Debian-based distribution like Ubuntu, try apt-get:

```
$ apt-get install git
```

Installing on Mac

There are three easy ways to install Git on a Mac. The easiest is to use the graphical Git installer, which you can download from the SourceForge page.

http://sourceforge.net/projects/git-osx-installer/

Figure 1-7. Git OS X installer. The other major way is to install Git via MacPorts (http://www.macports.org). If you have MacPorts installed, install Git via

```
$ sudo port install git +svn +doc +bash_completion +gitweb
```

You don’t have to add all the extras, but you’ll probably want to include +svn in case you ever have to use Git with Subversion repositories (see Chapter 8).

Homebrew (http://brew.sh/) is another alternative to install Git. If you have Homebrew installed, install Git via

```
$ brew install git
```

Installing on Windows

Installing Git on Windows is very easy. The msysGit project has one of the easier installation procedures. Simply download the installer exe file from the GitHub page, and run it:

```
http://msysgit.github.io
```

After it’s installed, you have both a command-line version (including an SSH client that will come in handy later) and the standard GUI.

Note on Windows usage: you should use Git with the provided msysGit shell (Unix style), it allows to use the complex lines of command given in this book. If you need, for some reason, to use the native Windows shell / command line console, you have to use double quotes instead of single quotes (for parameters with spaces in them) and you must quote the parameters ending with the circumflex accent (^) if they are last on the line, as it is a continuation symbol in Windows.

Read Getting started with Git online: https://riptutorial.com/git/topic/218/getting-started-with-git

Chapter 2: .mailmap file: Associating contributor and email aliases

Syntax

- # Only replace email addresses
  <primary@example.org> <alias@example.org>
- # Replace name by email address
  Contributor <primary@example.org>
- # Merge multiple aliases under one name and email
  # Note this will not associate 'Other <alias2@example.org>'.
  Contributor <primary@example.org> <alias1@example.org> Contributor
  <alias2@example.org>

Remarks

A `.mailmap` file may be created in any text editor and is just a plain text file containing optional contributor names, primary email addresses, and their aliases. it has to be placed in the project's root, next to the `.git` directory.

Keep in mind that this just modifies the visual output of commands like `git shortlog` or `git log --use-mailmap`. This will **not** rewrite commit history or prevent commits with varying names and/or email addresses.

To prevent commits based on information such as email addresses, you should use git hooks instead.

Examples

Merge contributers by aliases to show commit count in shortlog.

When contributors add to a project from different machines or operating systems, it may happen that they use different email addresses or names for this, which will fragment contributor lists and statistics.

Running `git shortlog -sn` to get a list of contributors and the number of commits by them could result in the following output:

```
Patrick Rothfuss 871
Elizabeth Moon 762
E. Moon 184
Rothfuss, Patrick 90
```

This fragmentation/disassociation may be adjusted by providing a plain text file `.mailmap`,

containing email mappings.

All names and email addresses listed in one line will be associated to the first named entity respectively.

For the example above, a mapping could look like this:

```
Patrick Rothfuss <fussy@kingkiller.com> Rothfuss, Patrick <fussy@kingkiller.com>
Elizabeth Moon <emoon@marines.mil> E. Moon <emoon@scifi.org>
```

Once this file exists in the project's root, running `git shortlog -sn` again will result in a condensed list:

```
Patrick Rothfuss 961
Elizabeth Moon 946
```

Read .mailmap file: Associating contributor and email aliases online: https://riptutorial.com/git/topic/1270/-mailmap-file--associating-contributor-and-email-aliases

Chapter 3: Aliases

Examples

Simple aliases

There are two ways of creating aliases in Git:

- with the `~/.gitconfig` file:

```
[alias]
    ci = commit
    st = status
    co = checkout
```

- with the command line:

```
git config --global alias.ci "commit"
git config --global alias.st "status"
git config --global alias.co "checkout"
```

After the alias is created - type:

- `git ci` instead of `git commit`,
- `git st` instead of `git status`,
- `git co` instead of `git checkout`.

As with regular git commands, aliases can be used beside arguments. For example:

```
git ci -m "Commit message..."
git co -b feature-42
```

List / search existing aliases

You can list existing git aliases using `--get-regexp`:

```
$ git config --get-regexp '^alias\.'
```

Searching aliases

To search aliases, add the following to your `.gitconfig` under `[alias]`:

```
aliases = !git config --list | grep ^alias\\. | cut -c 7- | grep -Ei --color \"$1\" "#"
```

Then you can:

- `git aliases` - show ALL aliases
- `git aliases commit` - only aliases containing "commit"

Advanced Aliases

Git lets you use non-git commands and full `sh` shell syntax in your aliases if you prefix them with `!`.

In your `~/.gitconfig` file:

```
[alias]
    temp = !git add -A && git commit -m "Temp"
```

The fact that full shell syntax is available in these prefixed aliases also means you can use shell functions to construct more complex aliases, such as ones which utilize command line arguments:

```
[alias]
    ignore = "!f() { echo $1 >> .gitignore; }; f"
```

The above alias defines the `f` function, then runs it with any arguments you pass to the alias. So running `git ignore .tmp/` would add `.tmp/` to your `.gitignore` file.

In fact, this pattern is so useful that Git defines `$1`, `$2`, etc. variables for you, so you don't even have to define a special function for it. (But keep in mind that Git will also append the arguments anyway, even if you access it via these variables, so you may want to add a dummy command at the end.)

Note that aliases prefixed with `!` in this way are run from the root directory of your git checkout, even if your current directory is deeper in the tree. This can be a useful way to run a command from the root without having to `cd` there explicitly.

```
[alias]
    ignore = "! echo $1 >> .gitignore"
```

Temporarily ignore tracked files

To temporarily mark a file as ignored (pass file as parameter to alias) - type:

```
unwatch = update-index --assume-unchanged
```

To start tracking file again - type:

```
watch = update-index --no-assume-unchanged
```

To list all files that has been temporarily ignored - type:

```
unwatched = "!git ls-files -v | grep '^[[:lower:]]'"
```

To clear the unwatched list - type:

```
watchall = "!git unwatched | xargs -L 1 -I % sh -c 'git watch `echo % | cut -c 2-`'"
```

Example of using the aliases:

```
git unwatch my_file.txt
git watch my_file.txt
git unwatched
git watchall
```

Show pretty log with branch graph

```
[alias]
  logp=log --pretty=format:'%h %ad | %s%d [%an]' --graph --date=short

  lg =  log --graph --date-order --first-parent \
     --pretty=format:'%C(auto)%h%Creset %C(auto)%d%Creset %s %C(green)(%ad) %C(bold
cyan)<%an>%Creset'
  lgb = log --graph --date-order --branches --first-parent \
     --pretty=format:'%C(auto)%h%Creset %C(auto)%d%Creset %s %C(green)(%ad) %C(bold
cyan)<%an>%Creset'
 lga = log --graph --date-order --all \
   --pretty=format:'%C(auto)%h%Creset %C(auto)%d%Creset %s %C(green)(%ad) %C(bold
cyan)<%an>%Creset'
```

Here an explanation of the options and placeholder used in the `--pretty` format (exhaustive list are available with `git help log`)

--graph - draw the commit tree

--date-order - use commit timestamp order when possible

--first-parent - follow only the first parent on merge node.

--branches - show all local branches (by default, only current branch is shown)

--all - show all local and remotes branches

%h - hash value for commit (abbreviated)

%ad - Date stamp (author)

%an - Author username

%an - Commit username

%C(auto) - to use colors defined in [color] section

%Creset - to reset color

%d - --decorate (branch & tag names)

%s - commit message

%ad - author date (will follow --date directive) (and not commiter date)

%an - author name (can be %cn for commiter name)

Updating code while keeping a linear history

Sometimes you need to keep a linear (non-branching) history of your code commits. If you are working on a branch for a while, this can be tricky if you have to do a regular `git pull` since that will record a merge with upstream.

```
[alias]
  up = pull --rebase
```

This will update with your upstream source, then reapply any work you have not pushed on top of whatever you pulled down.

To use:

```
git up
```

See which files are being ignored by your .gitignore configuration

```
[ alias ]

    ignored = ! git ls-files --others --ignored --exclude-standard --directory \
            && git ls-files --others -i --exclude-standard
```

Shows one line per file, so you can grep (only directories):

```
$ git ignored | grep '/$'
.yardoc/
doc/
```

Or count:

```
~$ git ignored | wc -l
199811                # oops, my home directory is getting crowded
```

Unstage staged files

Normally, to remove files that are staged to be committed using the `git reset` commit, `reset` has a lot of functions depending on the arguments provided to it. To completely unstage all files staged, we can make use of git aliases to create a new alias that uses `reset` but now we do not need to remember to provide the correct arguments to `reset`.

```
git config --global alias.unstage "reset --"
```

Now, any time you want to **unstage** stages files, type `git unstage` and you are good to go.

Read Aliases online: https://riptutorial.com/git/topic/337/aliases

Chapter 4: Analyzing types of workflows

Remarks

Using version control software like Git may be a little scary at first, but its intuitive design specializing with branching helps make a number of different types of workflows possible. Pick one that is right for your own development team.

Examples

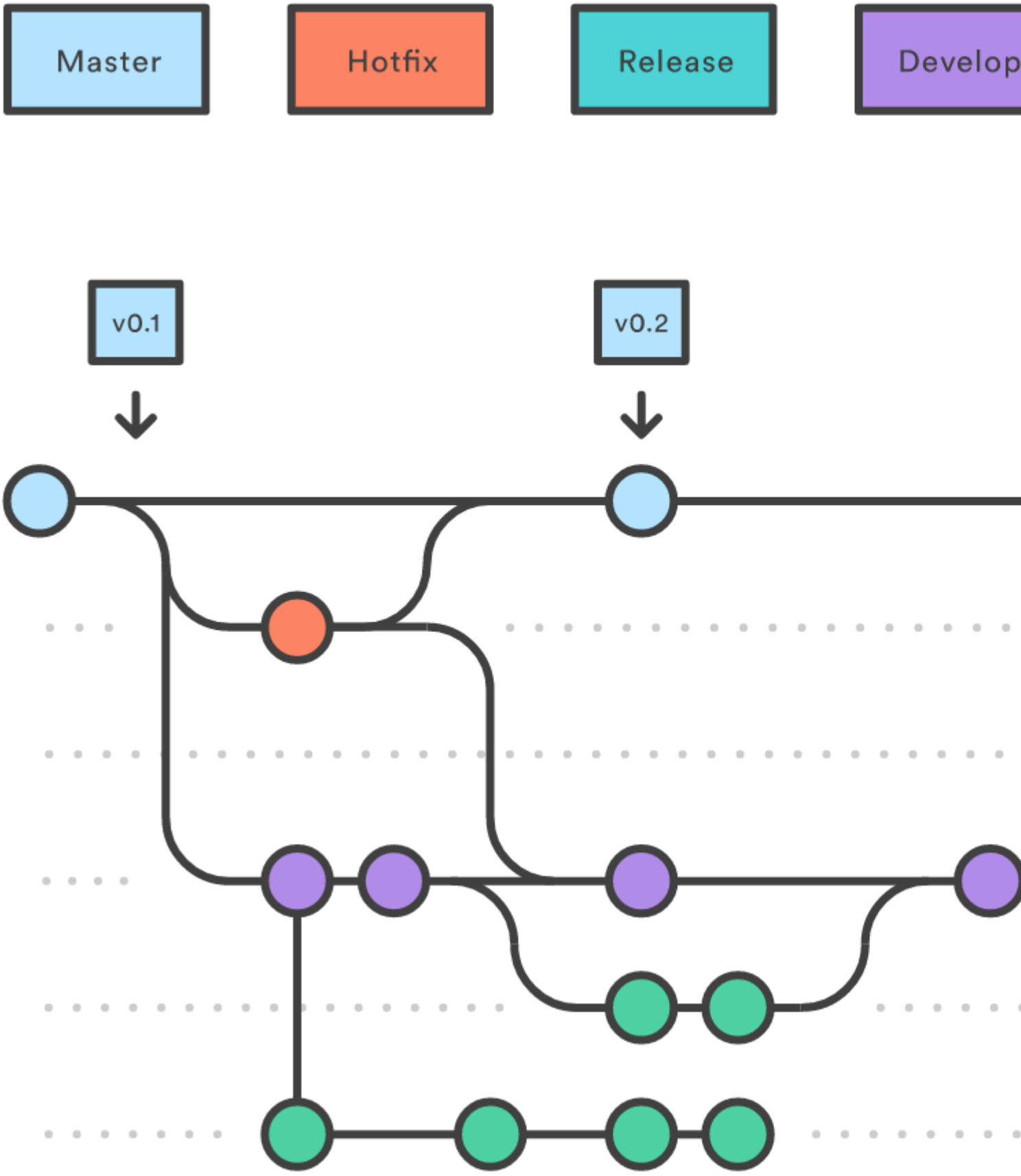
Gitflow Workflow

Originally proposed by Vincent Driessen, Gitflow is a development workflow using git and several pre-defined branches. This can seen as a special case of the Feature Branch Workflow.

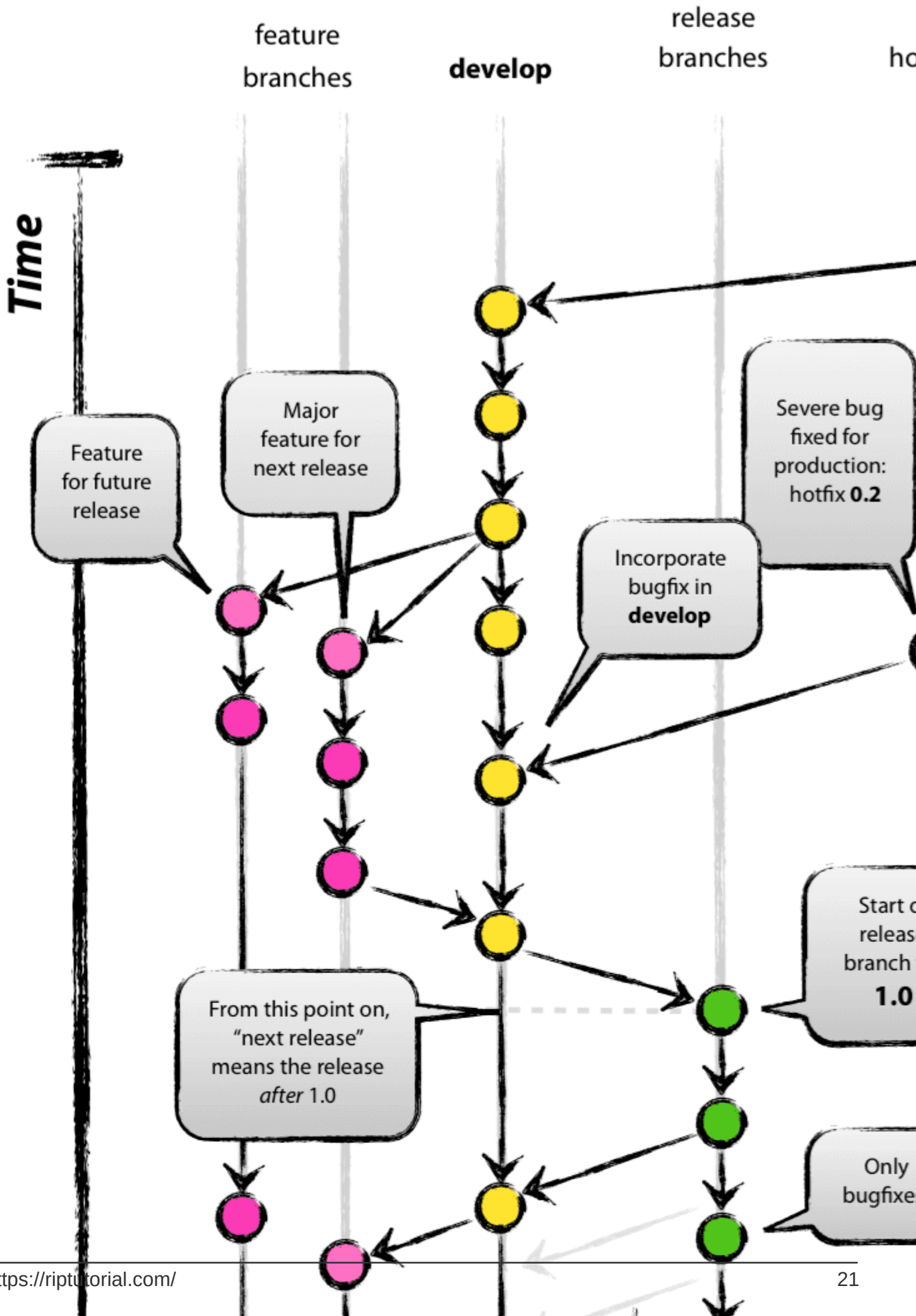
The idea of this one is to have separate branches reserved for specific parts in development:

- `master` branch is always the most recent *production* code. Experimental code does not belong here.
- `develop` branch contains all of the latest *development*. These developmental changes can be pretty much anything, but larger features are reserved for their own branches. Code here is always worked on and merged into `release` before release / deployment.
- `hotfix` branches are for minor bug fixes, which cannot wait until the next release. `hotfix` branches come off of `master` and are merged back into both `master` and `develop`.
- `release` branches are used to release new development from `develop` to `master`. Any last minute changes, such as bumping version numbers, are done in the release branch, and then are merged back into `master` and `develop`. When deploying a new version, `master` should be tagged with the current version number (e.g. using semantic versioning) for future reference and easy rollback.
- `feature` branches are reserved for bigger features. These are specifically developed in designated branches and integrated with `develop` when finished. Dedicated `feature` branches help to separate development and to be able to deploy *done* features independently from each other.

A visual representation of this model:

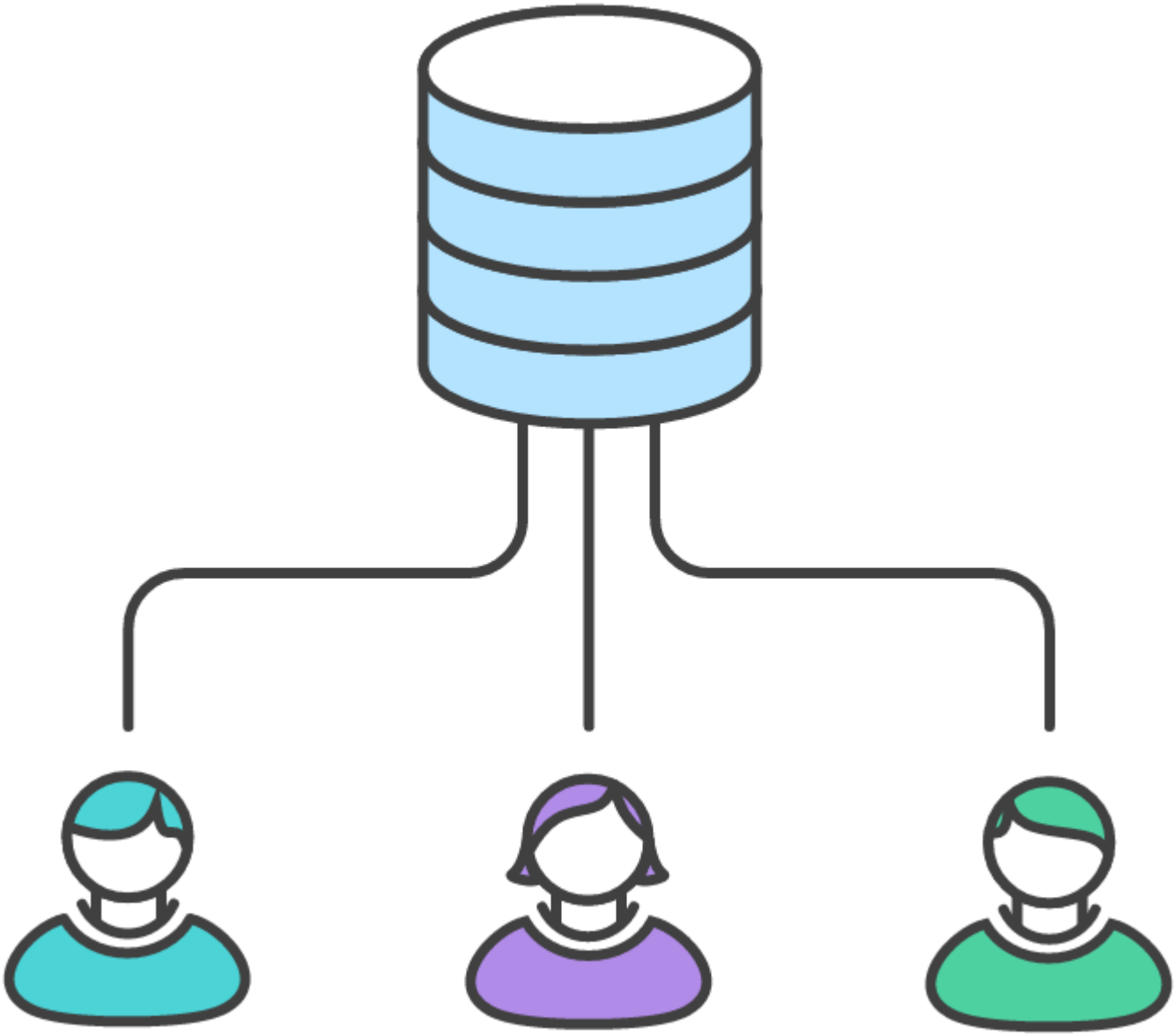


The original representation of this model:


feature branches
develop
release branches
Time
Feature for future release
Major feature for next release
Severe bug fixed for production: hotfix 0.2
Incorporate bugfix in develop
From this point on, "next release" means the release after 1.0

branch contains all active development. Contributors will need to be especially sure they pull the latest changes before continuing development, for this branch will be changing rapidly. Everyone has access to this repo and can commit changes right to the master branch.

Visual representation of this model:



This is the classic version control paradigm, upon which older systems like Subversion and CVS were built. Softwares that work this way are called Centralized Version Control Systems, or CVCS's. While Git is capable of working this way, there are notable disadvantages, such as being required to precede every pull with a merge. It's very possible for a team to work this way, but the constant merge conflict resolution can end up eating a lot of valuable time.

This is why Linus Torvalds created Git not as a CVCS, but rather as a *DVCS*, or *Distributed Version Control System*, similar to Mercurial. The advantage to this new way of doing things is the flexibility demonstrated in the other examples on this page.

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 `master` branch. This encapsulation makes it easy for multiple developers to work on a particular feature without disturbing the main codebase. It also means the `master` 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.

based on Atlassian Tutorials.

GitHub Flow

Popular within many open source projects but not only.

Master branch of a specific location (Github, Gitlab, Bitbucket, local server) contains the latest shippable version. For each new feature/bug fix/architectural change each developer creates a branch.

Changes happen on that branch and can be discussed in a pull request, code review, etc. Once accepted they get merged to the master branch.

Full flow by Scott Chacon:

- Anything in the master branch is deployable
- To work on something new, create a descriptively named branch off of master (ie: new-oauth2-scopes)
- Commit to that branch locally and regularly push your work to the same named branch on the server
- When you need feedback or help, or you think the branch is ready for merging, open a pull request
- After someone else has reviewed and signed off on the feature, you can merge it into master
- Once it is merged and pushed to 'master', you can and should deploy immediately

Originally presented on Scott Chacon's personal web site.

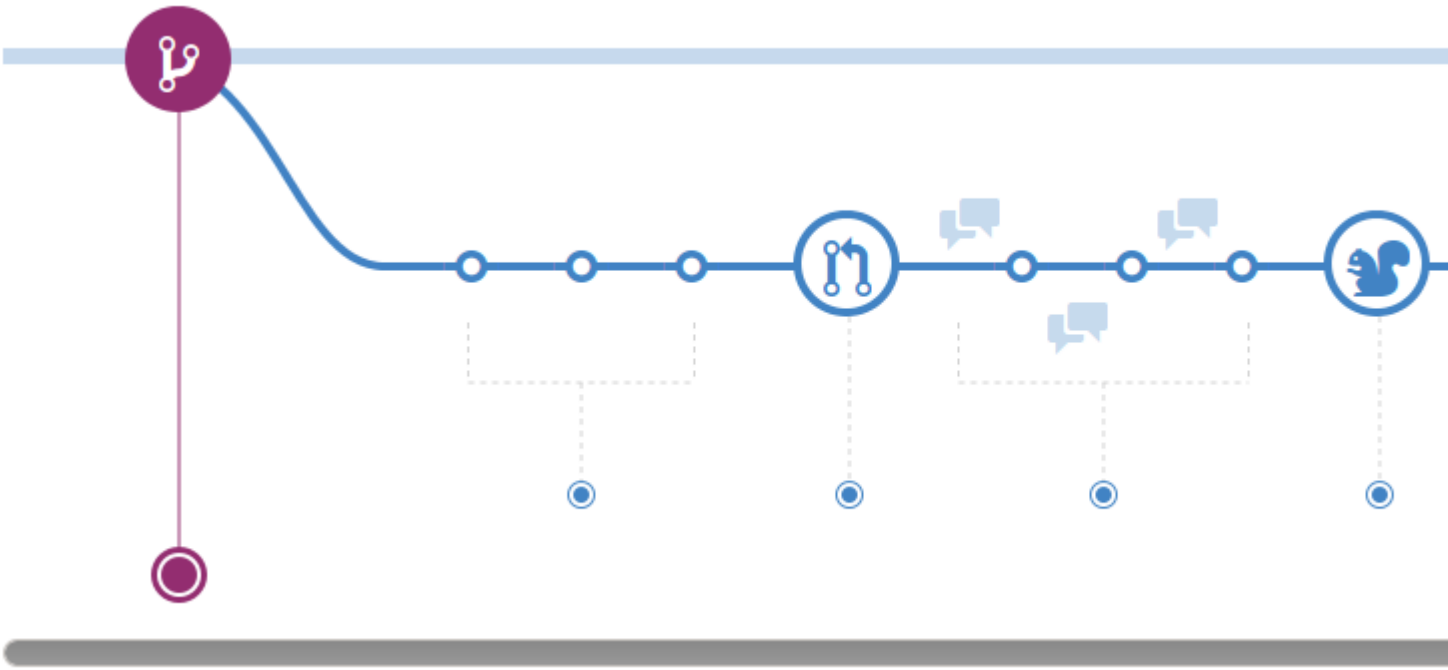


Image courtesy of the GitHub Flow reference

Read Analyzing types of workflows online: https://riptutorial.com/git/topic/1276/analyzing-types-of-workflows

Chapter 5: Archive

Syntax

- git archive [--format=<fmt>] [--list] [--prefix=<prefix>/] [<extra>] [-o <file> | --output=<file>] [--worktree-attributes] [--remote=<repo> [--exec=<git-upload-archive>]] <tree-ish> [<path>...]

Parameters

Parameter	Details
--format=<fmt>	Format of the resulting archive: `tar` or `zip`. If this options is not given and the output file is specified, the format is inferred from the filename if possible. Otherwise, defaults to `tar`.
-l, --list	Show all available formats.
-v, --verbose	Report progress to stderr.
--prefix=<prefix>/	Prepend <prefix>/ to each filename in the archive.
-o <file>, --output=<file>	Write the archive to <file> instead of stdout.
--worktree-attributes	Look for attributes in `.gitattributes` files in the working tree.
<extra>	This can be any options that the archiver backend understands. For `zip` backend, using `-0` will store the files without deflating them, while `-1` through `-9` can be used to adjust compression speed and ratio.
--remote=<repo>	Retrieve a tar archive from a remote repository `<repo>` rather than the local repository.
--exec=<git-upload-archive>	Used with `--remote` to specify the path to the `<git-upload-archive` on the remote.
<tree-ish>	The tree or commit to produce an archive for.
<path>	Without an optional parameter, all files and directories in the current working directory are included in the archive. If one or more paths are specified, only these are included.

Examples

Create an archive of git repository with directory prefix

It is considered good practice to use a prefix when creating git archives, so that extraction will place all files inside a directory. To create an archive of `HEAD` with a directory prefix:

```
git archive --output=archive-HEAD.zip --prefix=src-directory-name HEAD
```

When extracted all the files will be extracted inside a directory named `src-directory-name` in the current directory.

Create archive of git repository based on specific branch, revision, tag or directory

It is also possible to create archives of other items than `HEAD`, such as branches, commits, tags, and directories.

To create an archive of a local branch `dev`:

```
git archive --output=archive-dev.zip --prefix=src-directory-name dev
```

To create an archive of a remote branch `origin/dev`:

```
git archive --output=archive-dev.zip --prefix=src-directory-name origin/dev
```

To create an archive of a tag `v.01`:

```
git archive --output=archive-v.01.zip --prefix=src-directory-name v.01
```

Create an archive of files inside a specific sub directory (`sub-dir`) of revision `HEAD`:

```
git archive zip --output=archive-sub-dir.zip --prefix=src-directory-name HEAD:sub-dir/
```

Create an archive of git repository

With `git archive` it is possible to create compressed archives of a repository, for example for distributing releases.

Create a tar archive of current `HEAD` revision:

```
git archive --format tar HEAD | cat > archive-HEAD.tar
```

Create a tar archive of current `HEAD` revision with gzip compression:

```
git archive --format tar HEAD | gzip > archive-HEAD.tar.gz
```

This can also be done with (which will use the in-built tar.gz handling):

```
git archive --format tar.gz HEAD > archive-HEAD.tar.gz
```

Create a zip archive of current `HEAD` revision:

```
git archive --format zip HEAD > archive-HEAD.zip
```

Alternatively it is possible to just specify an output file with valid extension and the format and compression type will be inferred from it:

```
git archive --output=archive-HEAD.tar.gz HEAD
```

Read Archive online: https://riptutorial.com/git/topic/2815/archive