

# Distributed Git

Now that you have a remote Git repository set up as a focal point for all the developers to share their code, and you're familiar with basic Git commands in a local workflow, you'll look at how to utilize some of the distributed workflows that Git affords you.

In this chapter, you'll see how to work with Git in a distributed environment as a contributor and an integrator. That is, you'll learn how to contribute code successfully to a project and make it as easy on you and the project maintainer as possible, and also how to maintain a project successfully with a number of developers contributing.

## Distributed Workflows

Unlike Centralized Version Control Systems (CVCSs), the distributed nature of Git allows you to be far more flexible in how developers collaborate on projects. In centralized systems, every developer is a node working more or less equally on a central hub. In Git, however, every developer is potentially both a node and a hub—that is, every developer can both contribute code to other repositories and maintain a public repository on which others can base their work and which they can contribute to. This opens a vast range of workflow possibilities for your project and/or your team, so we'll cover a few common paradigms that take advantage of this flexibility. We'll go over the strengths and possible weaknesses of each design; you can choose a single one to use, or you can mix and match features from each.

### Centralized Workflow

In centralized systems, there is generally a single collaboration model—the centralized workflow. One central hub, or *repository*, can accept code, and everyone synchronizes their work to it. A number of developers are nodes—consumers of that hub—and synchronize to that one place.

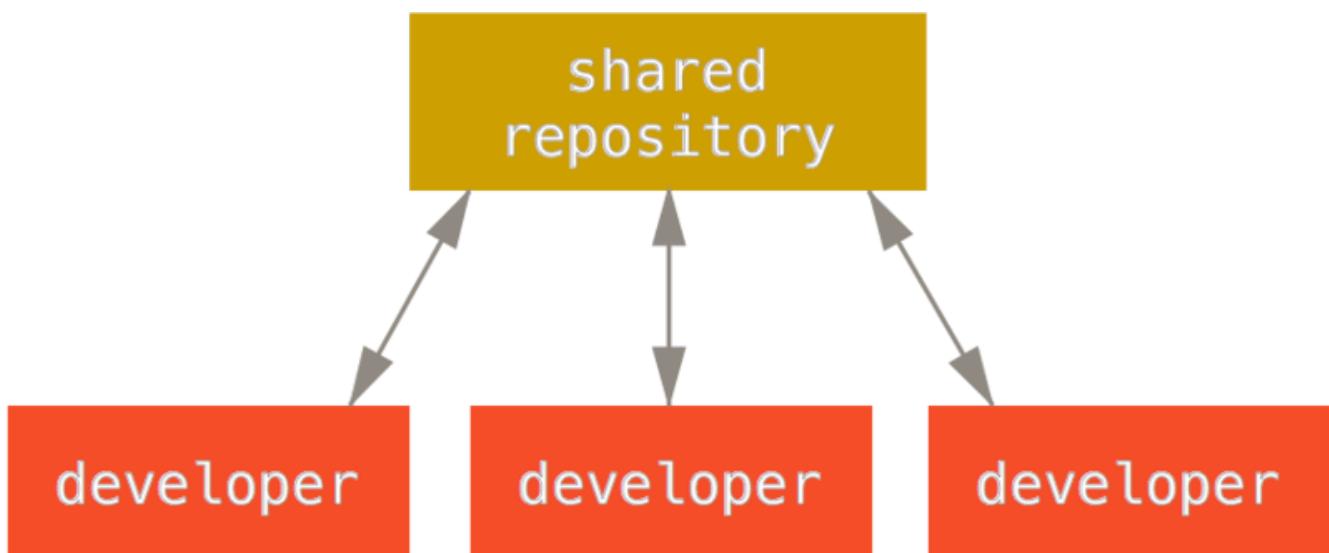


Figure 54. Centralized workflow.

This means that if two developers clone from the hub and both make changes, the first developer to push their changes back up can do so with no problems. The second developer must merge in the first one's work before pushing changes up, so as not to overwrite the first developer's changes.

This concept is as true in Git as it is in Subversion (or any CVCS), and this model works perfectly well in Git.

If you are already comfortable with a centralized workflow in your company or team, you can easily continue using that workflow with Git. Simply set up a single repository, and give everyone on your team push access; Git won't let users overwrite each other. Say John and Jessica both start working at the same time. John finishes his change and pushes it to the server. Then Jessica tries to push her changes, but the server rejects them. She is told that she's trying to push non-fast-forward changes and that she won't be able to do so until she fetches and merges. This workflow is attractive to a lot of people because it's a paradigm that many are familiar and comfortable with.

This is also not limited to small teams. With Git's branching model, it's possible for hundreds of developers to successfully work on a single project through dozens of branches simultaneously.

## Integration-Manager Workflow

Because Git allows you to have multiple remote repositories, it's possible to have a workflow where each developer has write access to their own public repository and read access to everyone else's. This scenario often includes a canonical repository that represents the "official" project. To contribute to that project, you create your own public clone of the project and push your changes to it. Then, you can send a request to the maintainer of the main project to pull in your changes. The maintainer can then add your repository as a remote, test your changes locally, merge them into their branch, and push back to their repository. The process works as follows (see [Integration-manager workflow](#)):

1. The project maintainer pushes to their public repository.
2. A contributor clones that repository and makes changes.
3. The contributor pushes to their own public copy.
4. The contributor sends the maintainer an email asking them to pull changes.
5. The maintainer adds the contributor's repository as a remote and merges locally.
6. The maintainer pushes merged changes to the main repository.

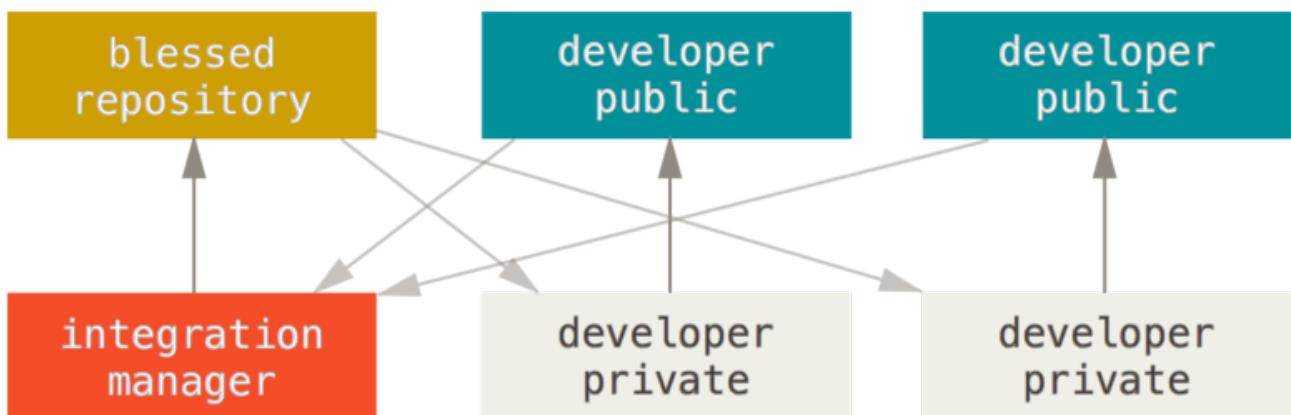


Figure 55. Integration-manager workflow.

This is a very common workflow with hub-based tools like GitHub or GitLab, where it's easy to fork a project and push your changes into your fork for everyone to see. One of the main advantages of

this approach is that you can continue to work, and the maintainer of the main repository can pull in your changes at any time. Contributors don't have to wait for the project to incorporate their changes—each party can work at their own pace.

## Dictator and Lieutenants Workflow

This is a variant of a multiple-repository workflow. It's generally used by huge projects with hundreds of collaborators; one famous example is the Linux kernel. Various integration managers are in charge of certain parts of the repository; they're called *lieutenants*. All the lieutenants have one integration manager known as the benevolent dictator. The benevolent dictator pushes from his directory to a reference repository from which all the collaborators need to pull. The process works like this (see [Benevolent dictator workflow](#)):

1. Regular developers work on their topic branch and rebase their work on top of `master`. The `master` branch is that of the reference directory to which the dictator pushes.
2. Lieutenants merge the developers' topic branches into their `master` branch.
3. The dictator merges the lieutenants' `master` branches into the dictator's `master` branch.
4. Finally, the dictator pushes that `master` branch to the reference repository so the other developers can rebase on it.

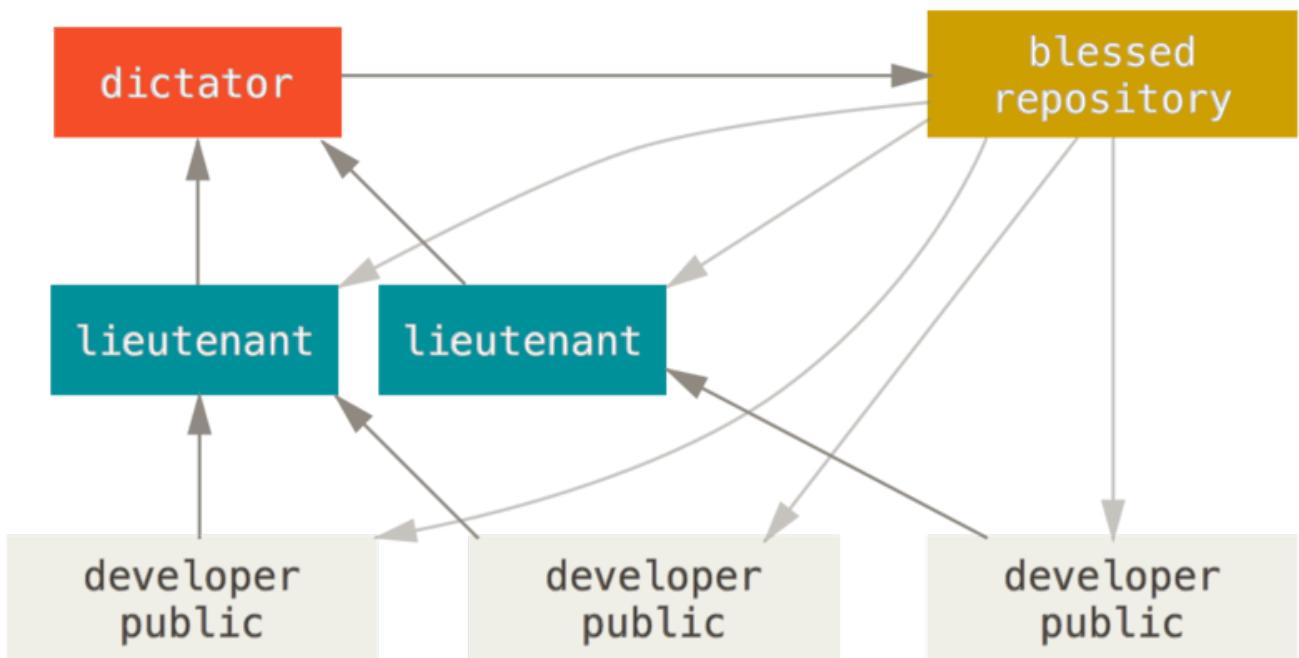


Figure 56. Benevolent dictator workflow.

This kind of workflow isn't common, but can be useful in very big projects, or in highly hierarchical environments. It allows the project leader (the dictator) to delegate much of the work and collect large subsets of code at multiple points before integrating them.

## Workflows Summary

These are some commonly used workflows that are possible with a distributed system like Git, but you can see that many variations are possible to suit your particular real-world workflow. Now that you can (hopefully) determine which workflow combination may work for you, we'll cover some

more specific examples of how to accomplish the main roles that make up the different flows. In the next section, you'll learn about a few common patterns for contributing to a project.

## Contributing to a Project

The main difficulty with describing how to contribute to a project are the numerous variations on how to do that. Because Git is very flexible, people can and do work together in many ways, and it's problematic to describe how you should contribute—every project is a bit different. Some of the variables involved are active contributor count, chosen workflow, your commit access, and possibly the external contribution method.

The first variable is active contributor count—how many users are actively contributing code to this project, and how often? In many instances, you'll have two or three developers with a few commits a day, or possibly less for somewhat dormant projects. For larger companies or projects, the number of developers could be in the thousands, with hundreds or thousands of commits coming in each day. This is important because with more and more developers, you run into more issues with making sure your code applies cleanly or can be easily merged. Changes you submit may be rendered obsolete or severely broken by work that is merged in while you were working or while your changes were waiting to be approved or applied. How can you keep your code consistently up to date and your commits valid?

The next variable is the workflow in use for the project. Is it centralized, with each developer having equal write access to the main codeline? Does the project have a maintainer or integration manager who checks all the patches? Are all the patches peer-reviewed and approved? Are you involved in that process? Is a lieutenant system in place, and do you have to submit your work to them first?

The next variable is your commit access. The workflow required in order to contribute to a project is much different if you have write access to the project than if you don't. If you don't have write access, how does the project prefer to accept contributed work? Does it even have a policy? How much work are you contributing at a time? How often do you contribute?

All these questions can affect how you contribute effectively to a project and what workflows are preferred or available to you. We'll cover aspects of each of these in a series of use cases, moving from simple to more complex; you should be able to construct the specific workflows you need in practice from these examples.

## Commit Guidelines

Before we start looking at the specific use cases, here's a quick note about commit messages. Having a good guideline for creating commits and sticking to it makes working with Git and collaborating with others a lot easier. The Git project provides a document that lays out a number of good tips for creating commits from which to submit patches—you can read it in the Git source code in the [Documentation/SubmittingPatches](#) file.

First, your submissions should not contain any whitespace errors. Git provides an easy way to check for this—before you commit, run `git diff --check`, which identifies possible whitespace errors and lists them for you.