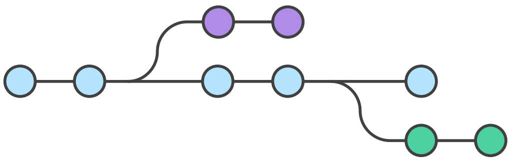


Using Branches

This tutorial is a comprehensive introduction to Git branches. First, we‘ll take a look at creating branches, which is like requesting a new project history. Then, we’ll see how git checkout can be used to select a branch. Finally, we'll learn how git merge can integrate the history of independent branches.

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.

git branch

Tutorials

Getting Started

Collaborating

Syncing

Making a Pull Request

Using Branches

git branch

git checkout

git merge

Comparing Workflows

Migrating to Git

Advanced Tips

first module of this series. 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` and `git merge` commands.

Usage

```
git branch
```

List all of the branches in your repository.

```
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>
```

Tutorials

Getting Started

Collaborating

Syncing

Making a Pull Request

Using Branches

git branch

git checkout

git merge

Comparing Workflows

Migrating to Git

Advanced Tips

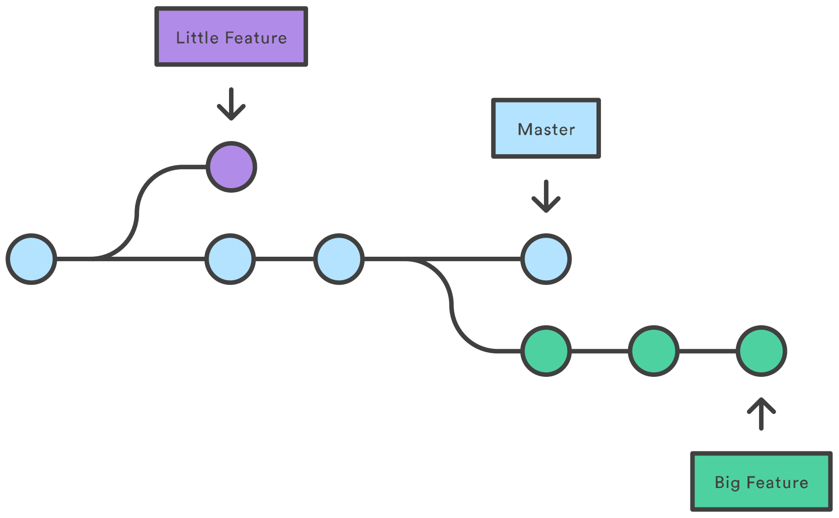
associated with a particular line of development.

```
git branch -m <branch>
```

Rename the current branch to `<branch>`.

Discussion

In Git, branches are a part of your everyday development process. 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 sure that unstable code is never committed to the main code base, and it gives you the chance to clean up your feature's history before merging it into the main branch.



For example, 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

Tutorials

Getting Started

Collaborating

Syncing

Making a Pull Request

Using Branches

git branch

git checkout

git merge

Comparing Workflows

Migrating to Git

Advanced Tips

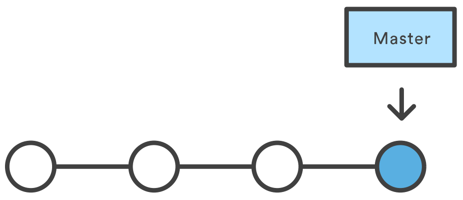
The implementation behind Git branches is much more lightweight than SVN's model. Instead of copying files from directory to directory, Git stores a branch as a reference to a commit. In this sense, a branch represents the *tip* of a series of commits—it's not a *container* for commits. The history for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

This has a dramatic impact on Git's merging model. Whereas merges in SVN are done on a file-basis, Git lets you work on the more abstract level of commits. You can actually see merges in the project history as a joining of two independent commit histories.

Example

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. So, if you start with a repository that looks like this:



Then, you create a branch using the following command:

```
git branch crazy-experiment
```

Tutorials

Getting Started

Collaborating

Syncing

Making a Pull Request

Using Branches

git branch

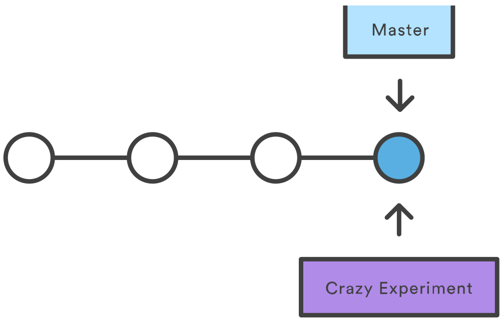
git checkout

git merge

Comparing Workflows

Migrating to Git

Advanced Tips



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. Please see the `git checkout` section of this module for more information.

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
If you are sure you want to delete it, run 'git b
```

This protects you from losing your reference to those commits, which means you would effectively lose 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:

without warnings, so use it judiciously.

Getting Started

Collaborating

Syncing

Making a Pull Request

Using Branches

git branch

git checkout

git merge

Comparing Workflows

Migrating to Git

Advanced Tips

git checkout

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.

In the previous module, we saw how `git checkout` can be used to view old commits. Checking out branches is similar 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.

Usage

```
git checkout <existing-branch>
```

Check out the specified branch, which should have already been created with `git branch`. This makes <existing-branch> the current branch, and updates the working directory to match.

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

Getting Started

Collaborating

Syncing

Making a Pull Request

Using Branches

git branch

git checkout

git merge

Comparing Workflows

Migrating to Git

Advanced Tips

`git checkout <new-branch>.`

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

Same as the above invocation, but base the new branch off of <existing-branch> instead of the current branch.

Discussion

`git checkout` works hand-in-hand with `git branch`. When you want to start a new feature, you create a branch with `git branch`, then check it out with `git checkout`. You can work on multiple features in a single repository by switching between them with `git checkout`.

Tutorials

Getting Started

Collaborating

Syncing

Making a Pull Request

Using Branches

git branch

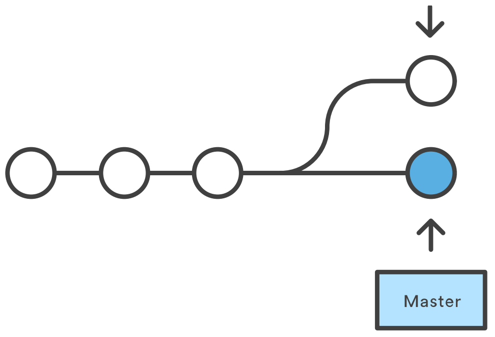
git checkout

git merge

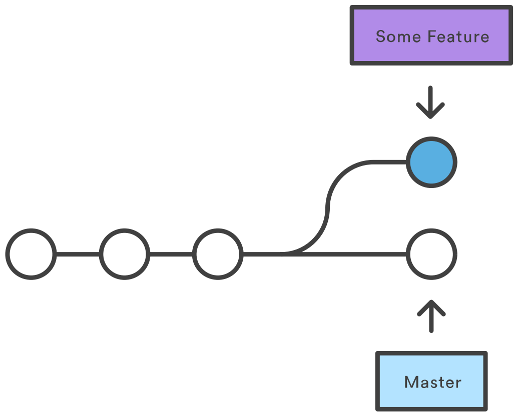
Comparing Workflows

Migrating to Git

Advanced Tips



Checking Out Some Feature



Having a dedicated branch for each new feature is a dramatic shift from the 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.

Detached HEADs

Now that we've seen the three main uses of `git checkout` we can talk about that "detached `HEAD`" we encountered in the previous module.

Tutorials

Getting Started

Collaborating

Syncing

Making a Pull Request

Using Branches

git branch

git checkout

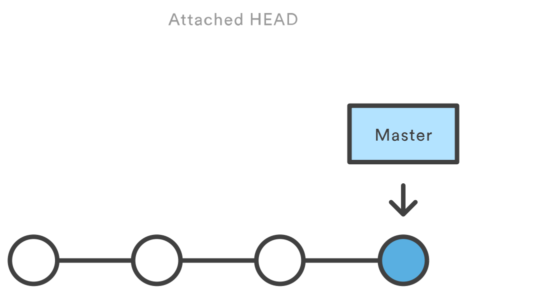
git merge

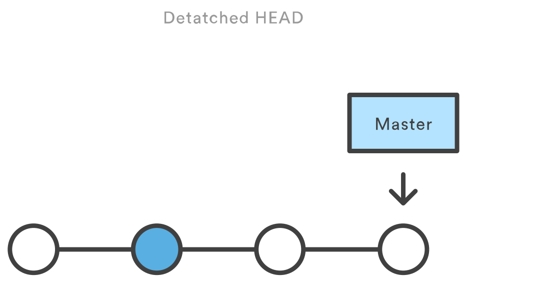
Comparing Workflows

Migrating to Git

Advanced Tips

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:

Tutorials

Getting Started

Collaborating

Syncing

Making a Pull Request

Using Branches

git branch

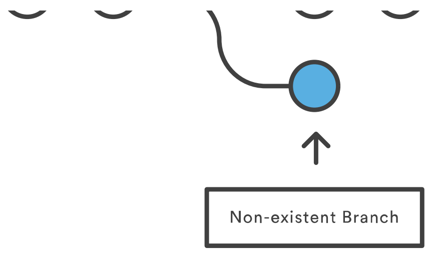
git checkout

git merge

Comparing Workflows

Migrating to Git

Advanced Tips



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.

Example

The following example demonstrates the basic Git branching process. When you want to start working on a new feature, you create a dedicated branch and switch into it:

```
git branch new-feature
git checkout new-feature
```

Then, you can commit new snapshots just like we've seen in previous modules:

```
# Edit some files
git add <file>
git commit -m "Started work on a new feature"
# Repeat
```

All of these are recorded in `new-feature`, which is completely isolated from `master`. You can add as many

Tutorials

Getting Started

Collaborating

Syncing

Making a Pull Request

Using Branches

git branch

git checkout

git merge

Comparing Workflows

Migrating to Git

Advanced Tips

the `master` branch:

```
git checkout master
```

This shows you the state of the repository before you started your feature. From here, you have the option to merge in the completed feature, branch off a brand new, unrelated feature, or do some work with the stable version of your project.

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.

Usage

```
git merge <branch>
```

Merge the specified branch into the current branch. Git

Tutorials

Getting Started

Collaborating

Syncing

Making a Pull Request

Using Branches

git branch

git checkout

git merge

Comparing Workflows

Migrating to Git

Advanced Tips

```
git merge --no-ff <branch>
```

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

Discussion

Once you’ve finished developing a feature in an isolated branch, it's important to be able to get it back into the main code base. Depending on the structure of your repository, Git has several distinct algorithms to accomplish this: a fast-forward merge or a 3-way 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 `master` would look something like the following:

Tutorials

Getting Started

Collaborating

Syncing

Making a Pull Request

Using Branches

git branch

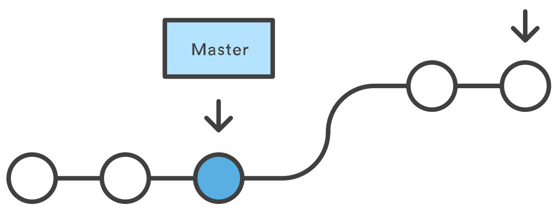
git checkout

git merge

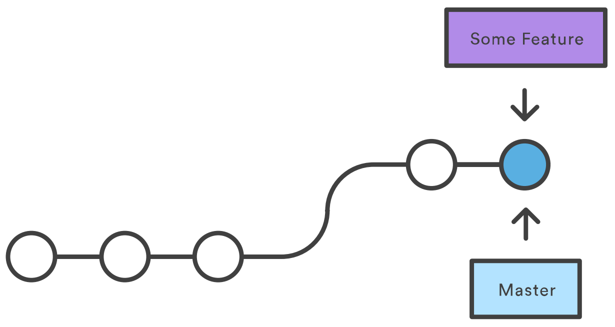
Comparing Workflows

Migrating to Git

Advanced Tips



After a Fast-Forward Merge



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.

Tutorials

Getting Started

Collaborating

Syncing

Making a Pull Request

Using Branches

git branch

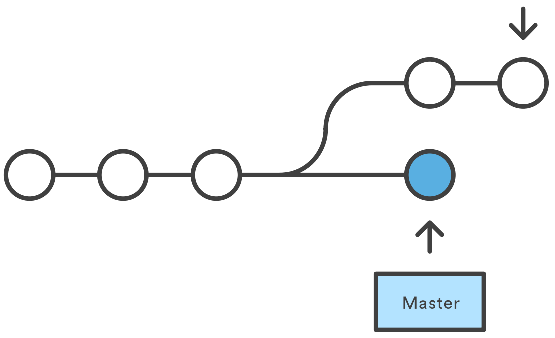
git checkout

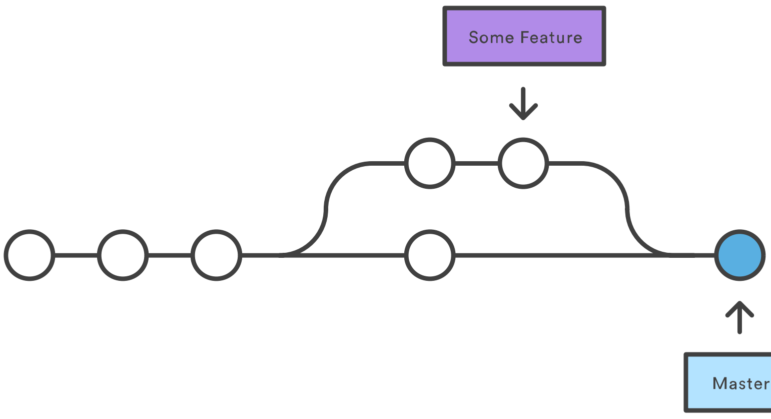
git merge

Comparing Workflows

Migrating to Git

Advanced Tips





While you can use either of these merge strategies, many developers like to use fast-forward merges (facilitated through rebasing) 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.

Resolving Conflicts

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

Getting Started

Collaborating

Syncing

Making a Pull Request

Using Branches

git branch

git checkout

git merge

Comparing Workflows

Migrating to Git

Advanced Tips

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 master
# Unmerged paths:
# (use "git add/rm ..." as appropriate to mark res
#
# both modified: hello.py
#
```

Then, 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.

Example

Fast-Forward Merge

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

Tutorials

Getting Started

Collaborating

Syncing

Making a Pull Request

Using Branches

git branch

git checkout

git merge

Comparing Workflows

Migrating to Git

Advanced Tips

```
git checkout -b new-feature master

# 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 master
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 master branch.

3-Way Merge

The next example is very similar, but requires a 3-way merge because `master` 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 master

# Edit some files
git add <file>
git commit -m "Start a feature"

# Edit some files
git add <file>
git commit -m "Finish a feature"
```

Tutorials

Getting Started

Collaborating

Syncing

Making a Pull Request

Using Branches

git branch

git checkout

git merge

Comparing Workflows

Migrating to Git

Advanced Tips

```
git add <file>
git commit -m "Make some super-stable changes to r

# 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 `master` 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 `master` 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 `master` and doing a fast-forward merge. This prevents superfluous merge commits from cluttering up the project history.

Next up:

Comparing Workflows

Tutorials

Getting Started

Collaborating

Syncing

Making a Pull Request

Using Branches

git branch

git checkout

git merge

Comparing Workflows

Migrating to Git

Advanced Tips

Powered By

Tutorials

Enter Your Email For Git News

Getting Started

Collaborating

Syncing

Making a Pull Request

Using Branches

git branch

git checkout

git merge

Comparing Workflows

Migrating to Git

Advanced Tips