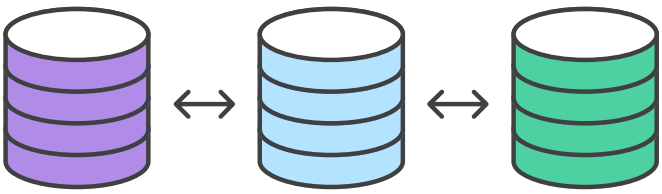
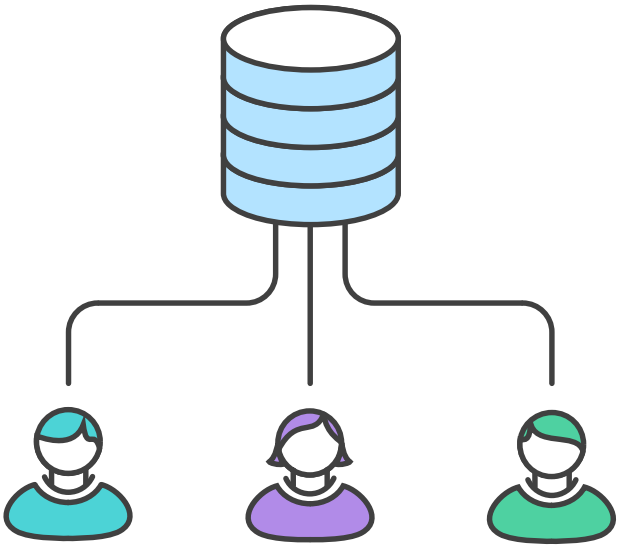


Comparing Workflows

The array of possible workflows can make it hard to know where to begin when implementing Git in the workplace. This page provides a starting point by surveying the most common Git workflows for enterprise teams.

As you read through, remember that these workflows are designed to be guidelines rather than concrete rules. We want to show you what's possible, so you can mix and match aspects from different workflows to suit your individual needs.

Centralized Workflow



Transitioning to a distributed version control system may seem like a daunting task, but *you don't have to change your existing workflow* to take advantage of Git. Your team can develop projects in the exact same way as they do with Subversion.

However, using Git to power your development workflow presents a few advantages over SVN. First, it gives every developer their own *local* copy of the entire project. This isolated environment lets each developer work independently of all other changes to a project—they can add commits to their local repository and completely forget about upstream developments until it's convenient for them.

Second, it gives you access to Git's robust branching and merging model. Unlike SVN, Git branches are designed to be a fail-safe mechanism for integrating code and sharing changes between repositories.

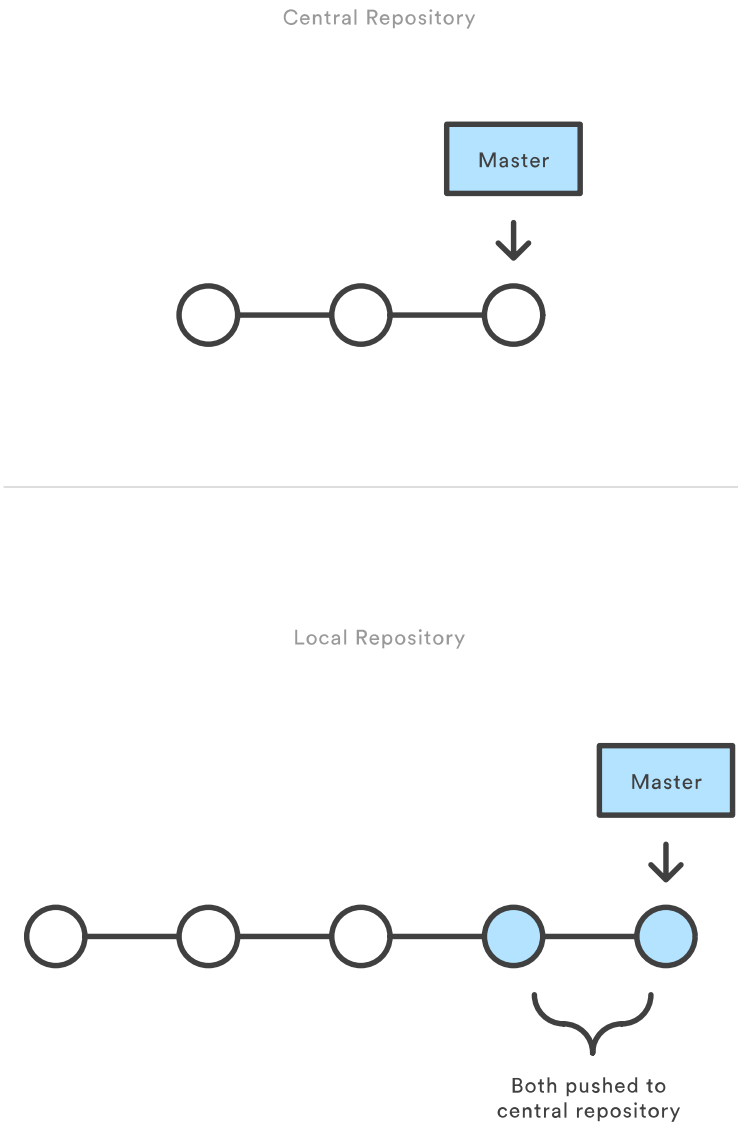
How It Works

Like Subversion, the Centralized Workflow uses a central repository to serve as the single point-of-entry for all changes to the project. Instead of `trunk`, the default development branch is called `master` and all changes are committed into this branch. This workflow doesn't

require any other branches besides `master`.

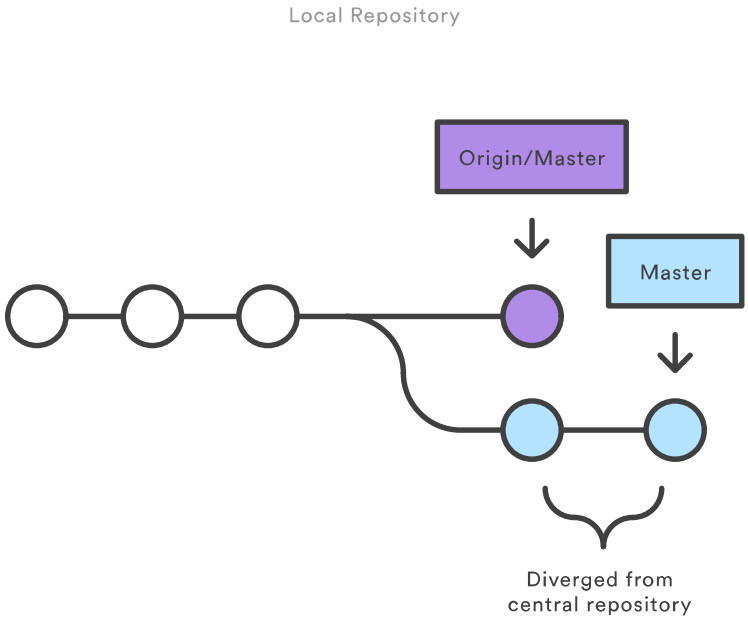
Developers start by cloning the central repository. In their own local copies of the project, they edit files and commit changes as they would with SVN; however, these new commits are stored *locally* —they're completely isolated from the central repository. This lets developers defer synchronizing upstream until they're at a convenient break point.

To publish changes to the official project, developers "push" their local `master` branch to the central repository. This is the equivalent of `svn commit`, except that it adds all of the local commits that aren't already in the central `master` branch.



Managing Conflicts

The central repository represents the official project, so its commit history should be treated as sacred and immutable. If a developer's local commits diverge from the central repository, Git will refuse to push their changes because this would overwrite official commits.



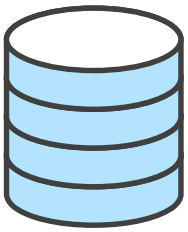
Before the developer can publish their feature, they need to fetch the updated central commits and rebase their changes on top of them. This is like saying, "I want to add my changes to what everyone else has already done." The result is a perfectly linear history, just like in traditional SVN workflows.

If local changes directly conflict with upstream commits, Git will pause the rebasing process and give you a chance to manually resolve the conflicts. The nice thing about Git is that it uses the same `git status` and `git add` commands for both generating commits and resolving merge conflicts. This makes it easy for new developers to manage their own merges. Plus, if they get themselves into trouble, Git makes it very easy to abort the entire rebase and try again (or go find help).

Example

Let's take a step-by-step look at how a typical small team would collaborate using this workflow. We'll see how two developers, John and Mary, can work on separate features and share their contributions via a centralized repository.

Someone initializes the central repository



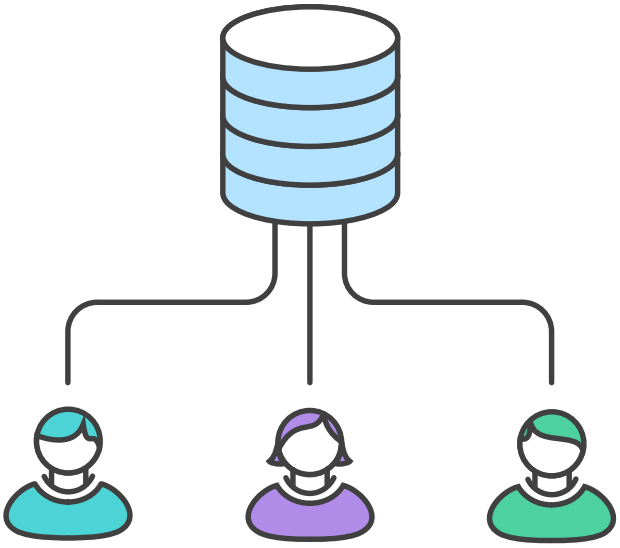
First, someone needs to create the central repository on a server. If it's a new project, you can initialize an empty repository. Otherwise, you'll need to import an existing Git or SVN repository.

Central repositories should always be bare repositories (they shouldn't have a working directory), which can be created as follows:

```
ssh user@host git init --bare /path/to/repo.git
```

Be sure to use a valid SSH username for `user`, the domain or IP address of your server for `host`, and the location where you'd like to store your repo for `/path/to/repo.git`. Note that the `.git` extension is conventionally appended to the repository name to indicate that it's a bare repository.

Everybody clones the central repository

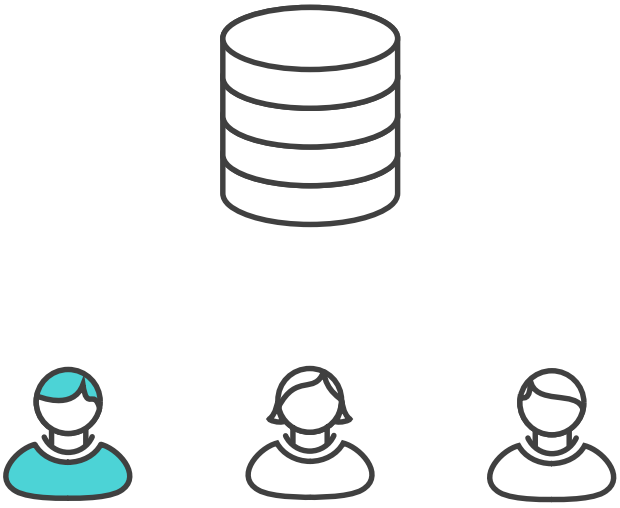


Next, each developer creates a local copy of the entire project. This is accomplished via the `git clone` command:

```
git clone ssh://user@host/path/to/repo.git
```

When you clone a repository, Git automatically adds a shortcut called `origin` that points back to the “parent” repository, under the assumption that you'll want to interact with it further on down the road.

John works on his feature



In his local repository, John can develop features using the standard Git commit process: edit, stage, and commit. If you're not familiar with the staging area, it's a way to prepare a commit without having to include every change in the working directory. This lets you create highly focused commits, even if you've made a lot of local changes.

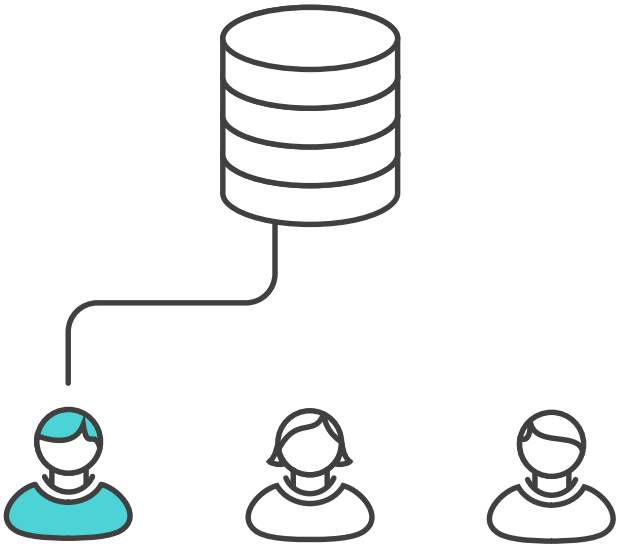
```
git status # View the state of the repo
git add <some-file> # Stage a file
git commit # Commit a file</some-file>
```

Remember that since these commands create local commits, John can repeat this process as many times as he wants without worrying about what's going on in the central repository. This can be very useful for large features that need to be broken down into simpler, more atomic chunks.

Mary works on her feature

Meanwhile, Mary is working on her own feature in her own local repository using the same edit/stage/commit process. Like John, she doesn't care what's going on in the central repository, and she *really* doesn't care what John is doing in his local repository, since all local repositories are *private*.

John publishes his feature

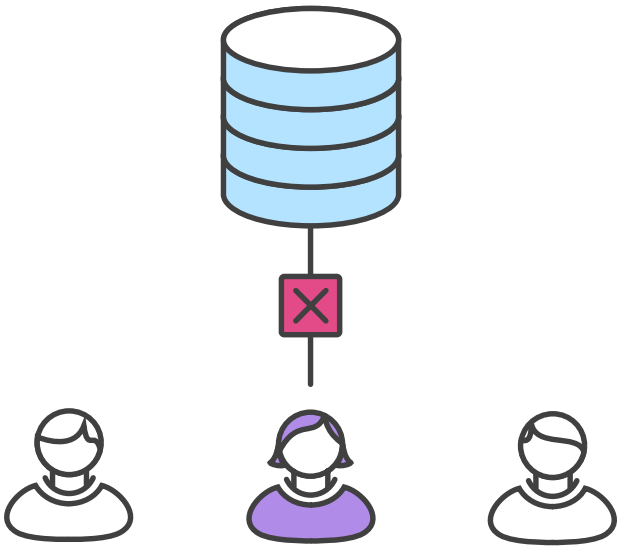


Once John finishes his feature, he should publish his local commits to the central repository so other team members can access it. He can do this with the `git push` command, like so:

```
git push origin master
```

Remember that `origin` is the remote connection to the central repository that Git created when John cloned it. The `master` argument tells Git to try to make the `origin`'s `master` branch look like his local `master` branch. Since the central repository hasn't been updated since John cloned it, this won't result in any conflicts and the push will work as expected.

Mary tries to publish her feature



Let's see what happens if Mary tries to push her feature after John has successfully published his changes to the central repository. She can use the exact same push command:

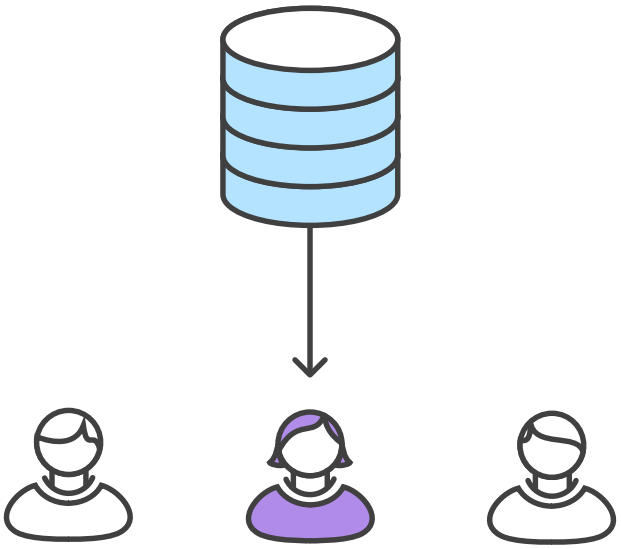
```
git push origin master
```

But, since her local history has diverged from the central repository, Git will refuse the request with a rather verbose error message:

```
error: failed to push some refs to '/path/to/repo.git'
hint: Updates were rejected because the tip of your current
hint: its remote counterpart. Merge the remote changes (e.g.
hint: before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help
```

This prevents Mary from overwriting official commits. She needs to pull John's updates into her repository, integrate them with her local changes, and then try again.

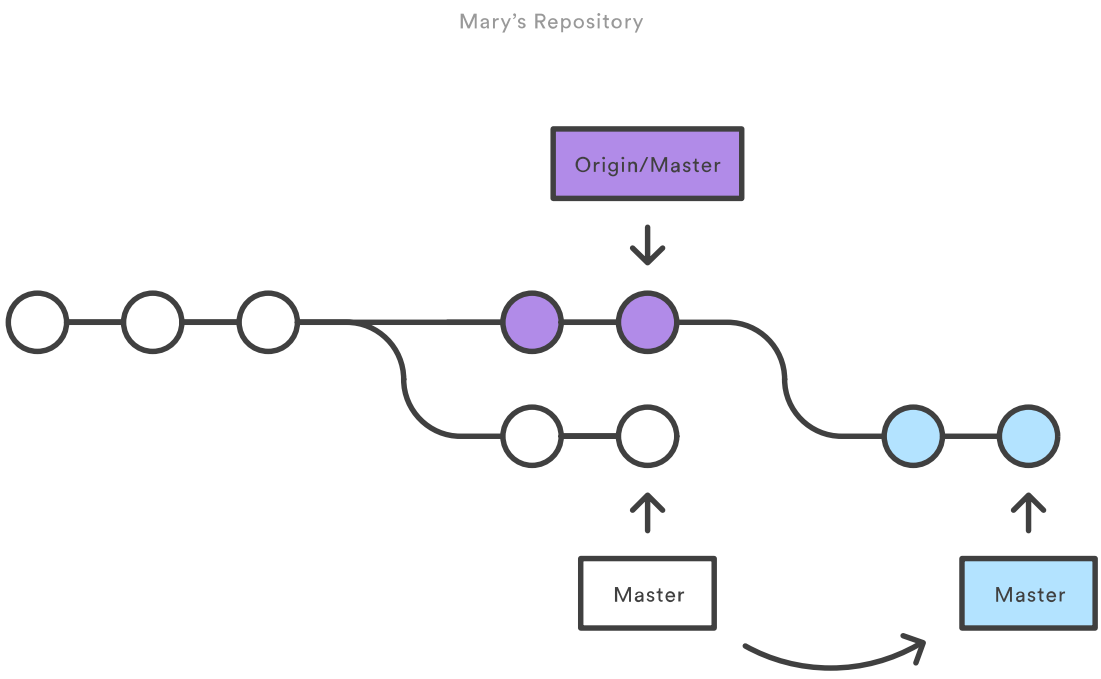
Mary rebases on top of John's commit(s)



Mary can use `git pull` to incorporate upstream changes into her repository. This command is sort of like `svn update`—it pulls the entire upstream commit history into Mary's local repository and tries to integrate it with her local commits:

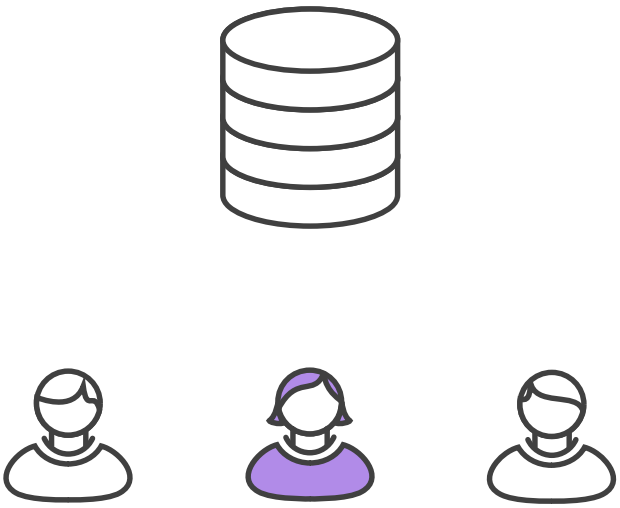
```
git pull --rebase origin master
```

The `--rebase` option tells Git to move all of Mary's commits to the tip of the `master` branch after synchronising it with the changes from the central repository, as shown below:



The pull would still work if you forgot this option, but you would wind up with a superfluous “merge commit” every time someone needed to synchronize with the central repository. For this workflow, it’s always better to rebase instead of generating a merge commit.

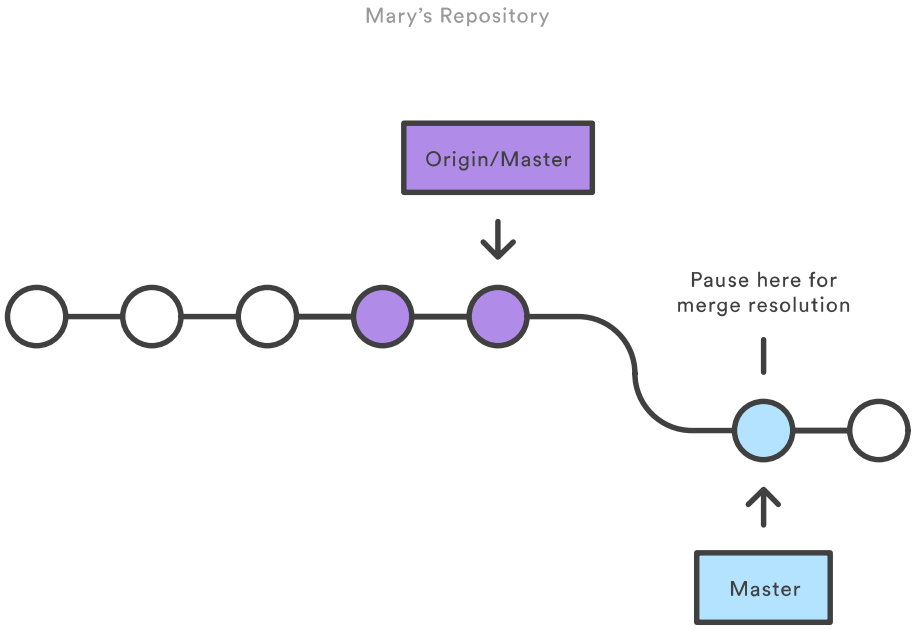
Mary resolves a merge conflict



Rebasing works by transferring each local commit to the updated `master` branch one at a time. This means that you catch merge conflicts on a commit-by-commit basis rather than resolving all of them in one massive merge commit. This keeps your commits as focused as possible and makes for a clean project history. In turn, this makes it much easier to figure out where bugs were introduced and, if necessary, to roll back changes with minimal impact on the project.

If Mary and John are working on unrelated features, it’s unlikely that the rebasing process will generate conflicts. But if it does, Git will pause the rebase at the current commit and output the following message, along with some relevant instructions:

```
CONFLICT (content): Merge conflict in <some-file>
```



The great thing about Git is that *anyone* can resolve their own merge conflicts. In our example, Mary would simply run a `git status` to see where the problem is. Conflicted files will appear in the Unmerged paths section:

```
# Unmerged paths:
# (use "git reset HEAD <some-file>..." to unstage)
# (use "git add/rm <some-file>..." as appropriate to mark re
#
# both modified: <some-file>
```

Then, she'll edit the file(s) to her liking. Once she's happy with the result, she can stage the file(s) in the usual fashion and let `git rebase` do the rest:

```
git add <some-file>
git rebase --continue
```

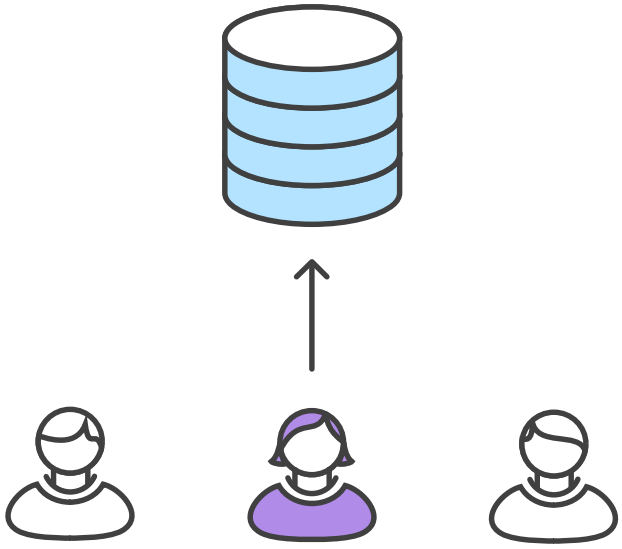
And that's all there is to it. Git will move on to the next commit and repeat the process for any other commits that generate conflicts.

If you get to this point and realize and you have no idea what's going on, don't panic. Just execute the following command and you'll be right back to where you started before you ran

[git pull --rebase](/tutorials/syncing/git-pull):

```
git rebase --abort
```

Mary successfully publishes her feature



After she's done synchronizing with the central repository, Mary will be able to publish her changes successfully:

```
git push origin master
```

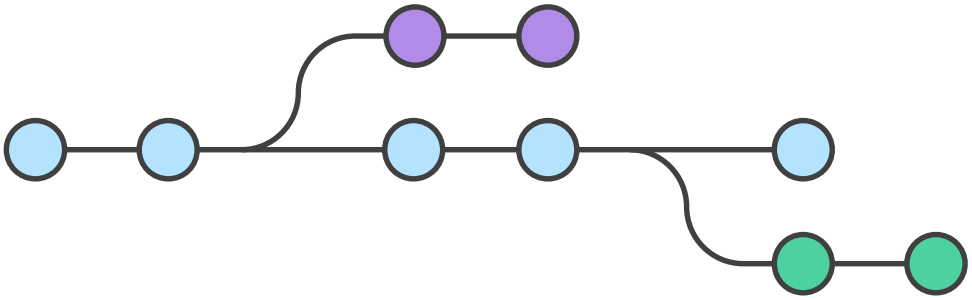
Where To Go From Here

As you can see, it's possible to replicate a traditional Subversion development environment using only a handful of Git commands. This is great for transitioning teams off of SVN, but it doesn't leverage the distributed nature of Git.

If your team is comfortable with the Centralized Workflow but wants to streamline its collaboration efforts, it's definitely worth exploring the benefits of the Feature Branch Workflow. By dedicating an isolated

branch to each feature, it's possible to initiate in-depth discussions around new additions before integrating them into the official project.

Feature Branch Workflow



Once you've got the hang of the Centralized Workflow, adding feature branches to your development process is an easy way to encourage collaboration and streamline communication between developers.

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.

How It Works

The Feature Branch Workflow still uses a central repository, and `master` still represents the official project history. But, instead of committing directly on their local `master` branch, developers create a new branch every time they start work on a new feature. Feature branches should have descriptive names, like `animated-menu-items` or `issue-#1061`. The idea is to give a clear, highly-focused purpose to each branch.

Git makes no technical distinction between the `master` branch and feature branches, so developers can edit, stage, and commit changes to a feature branch just as they did in the Centralized Workflow.

In addition, feature branches can (and should) be pushed to the central repository. This makes it possible to share a feature with other developers without touching any official code. Since `master` is the only “special” branch, storing several feature branches on the central repository doesn’t pose any problems. Of course, this is also a convenient way to back up everybody’s local commits.

Pull Requests

Aside from isolating feature development, branches make it possible to discuss changes via pull requests. Once someone completes a feature, they don’t immediately merge it into `master`. Instead, they push the feature branch to the central server and file a pull request asking to merge their additions into `master`. This gives other developers an opportunity to review the changes before they become a part of the main codebase.

Code review is a major benefit of pull requests, but they’re actually designed to be a generic way to talk about code. You can think of pull

requests as a discussion dedicated to a particular branch. This means that they can also be used much earlier in the development process. For example, if a developer needs help with a particular feature, all they have to do is file a pull request. Interested parties will be notified automatically, and they'll be able to see the question right next to the relevant commits.

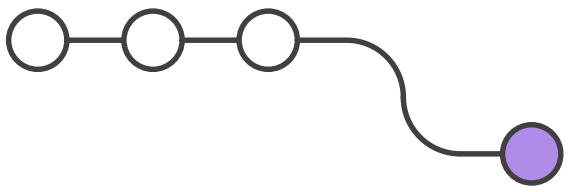
Once a pull request is accepted, the actual act of publishing a feature is much the same as in the Centralized Workflow. First, you need to make sure your local `master` is synchronized with the upstream `master`. Then, you merge the feature branch into `master` and push the updated `master` back to the central repository.

Pull requests can be facilitated by product repository management solutions like Bitbucket or Stash. View the Stash pull requests documentation for an example.

Example

The example included below demonstrates a pull request as a form of code review, but remember that they can serve many other purposes.

Mary begins a new feature



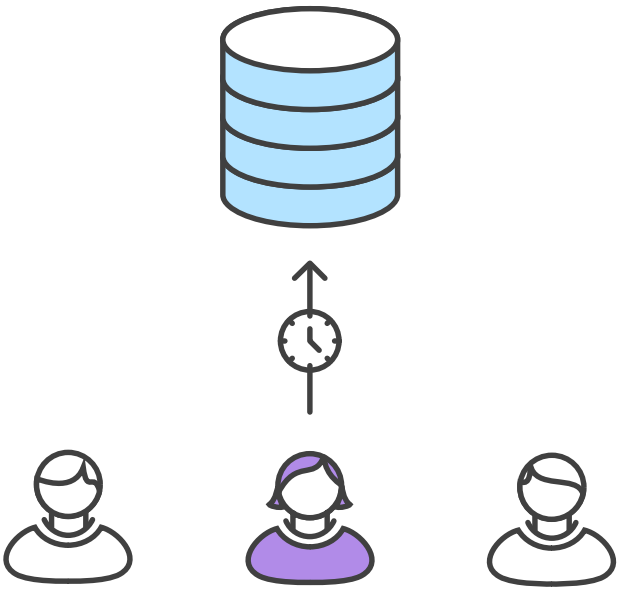
Before she starts developing a feature, Mary needs an isolated branch to work on. She can request a new branch with the following command:

```
git checkout -b marys-feature master
```

This checks out a branch called `marys-feature` based on `master`, and the `-b` flag tells Git to create the branch if it doesn't already exist. On this branch, Mary edits, stages, and commits changes in the usual fashion, building up her feature with as many commits as necessary:

```
git status
git add <some-file>
git commit
```

Mary goes to lunch

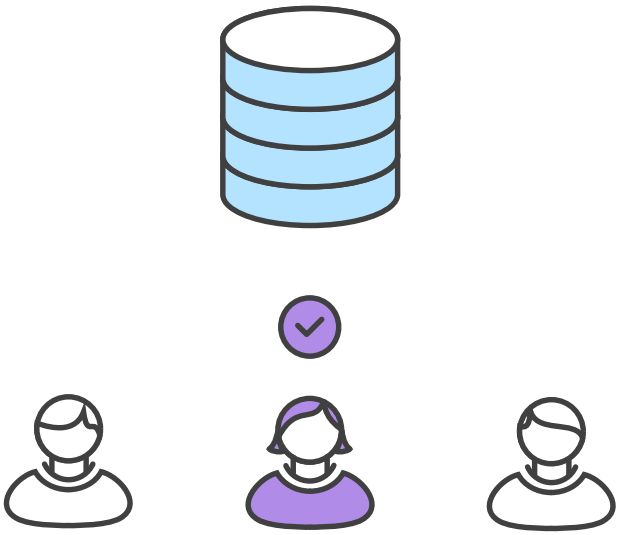


Mary adds a few commits to her feature over the course of the morning. Before she leaves for lunch, it's a good idea to push her feature branch up to the central repository. This serves as a convenient backup, but if Mary was collaborating with other developers, this would also give them access to her initial commits.

```
git push -u origin marys-feature
```

This command pushes `marys-feature` to the central repository (`origin`), and the `-u` flag adds it as a remote tracking branch. After setting up the tracking branch, Mary can call `git push` without any parameters to push her feature.

Mary finishes her feature

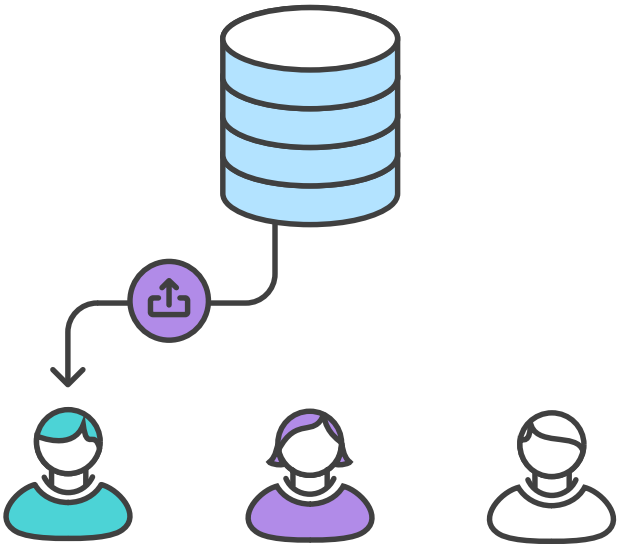


When Mary gets back from lunch, she completes her feature. Before merging it into `master`, she needs to file a pull request letting the rest of the team know she's done. But first, she should make sure the central repository has her most recent commits:

```
git push
```

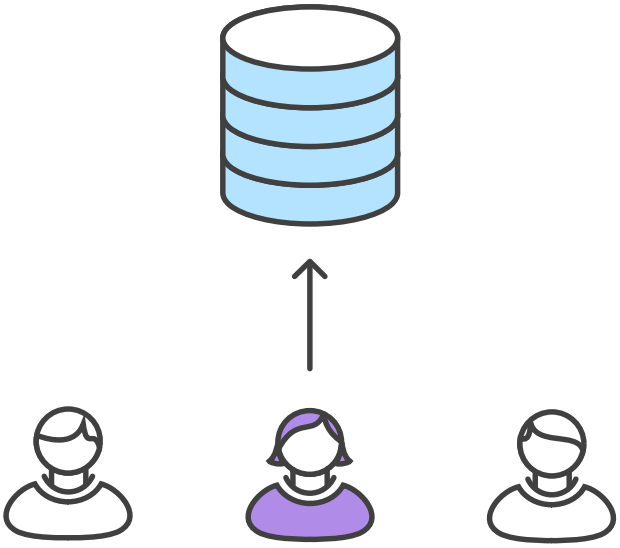
Then, she files the pull request in her Git GUI asking to merge `marys-feature` into `master`, and team members will be notified automatically. The great thing about pull requests is that they show comments right next to their related commits, so it's easy to ask questions about specific changesets.

Bill receives the pull request



Bill gets the pull request and takes a look at `marys-feature`. He decides he wants to make a few changes before integrating it into the official project, and he and Mary have some back-and-forth via the pull request.

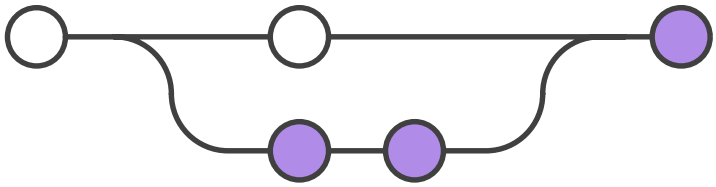
Mary makes the changes



To make the changes, Mary uses the exact same process as she did to create the first iteration of her feature. She edits, stages, commits, and pushes updates to the central repository. All her activity shows up in the pull request, and Bill can still make comments along the way.

If he wanted, Bill could pull `marys-feature` into his local repository and work on it on his own. Any commits he added would also show up in the pull request.

Mary publishes her feature



Once Bill is ready to accept the pull request, someone needs to merge the feature into the stable project (this can be done by either Bill or Mary):

```
git checkout master
git pull
git pull origin marys-feature
git push
```

First, whoever's performing the merge needs to check out their `master` branch and make sure it's up to date. Then, `git pull origin marys-feature` merges the central repository's copy of `marys-feature`. You could also use a simple `git merge marys-feature`, but the command shown above makes sure you're always pulling the most up-to-date version of the feature branch. Finally, the updated `master` needs to get pushed back to `origin`.

This process often results in a merge commit. Some developers like this because it's like a symbolic joining of the feature with the rest of the code base. But, if you're partial to a linear history, it's possible to rebase the feature onto the tip of `master` before executing the merge, resulting in a fast-forward merge.

Some GUI's will automate the pull request acceptance process by running all of these commands just by clicking an "Accept" button. If yours doesn't, it should at least be able to automatically close the pull request when the feature branch gets merged into `master`

Meanwhile, John is doing the exact same thing

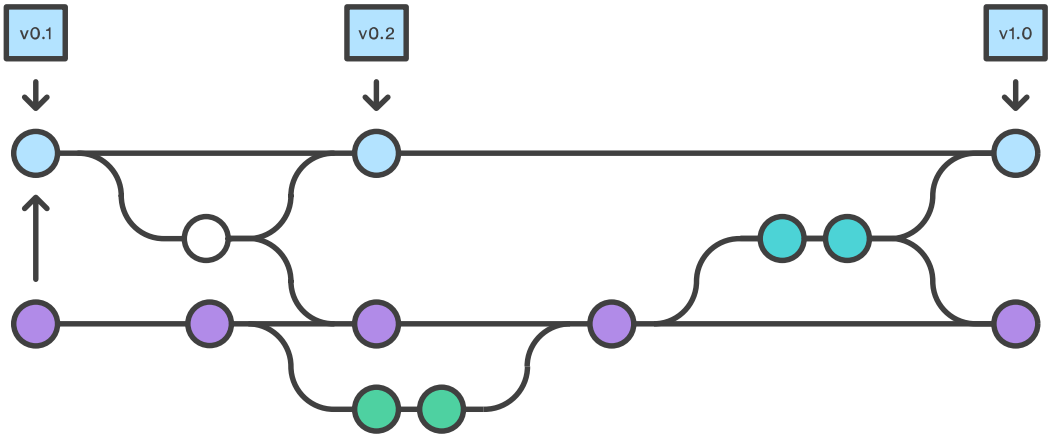
While Mary and Bill are working on `marys-feature` and discussing it in her pull request, John is doing the exact same thing with his own feature branch. By isolating features into separate branches, everybody can work independently, yet it's still trivial to share changes with other developers when necessary.

Where To Go From Here

For a walkthrough of feature branching on Bitbucket, check out the Using Git Branches documentation. By now, you can hopefully see how feature branches are a way to quite literally multiply the functionality of the single `master` branch used in the Centralized Workflow. In addition, feature branches also facilitate pull requests, which makes it possible to discuss specific commits right inside of your version control GUI.

The Feature Branch Workflow is an incredibly flexible way to develop a project. The problem is, sometimes it's too flexible. For larger teams, it's often beneficial to assign more specific roles to different branches. The Gitflow Workflow is a common pattern for managing feature development, release preparation, and maintenance.

Gitflow Workflow



The Gitflow Workflow section below is derived from Vincent Driessen at nvie.

The Gitflow Workflow defines a strict branching model designed around the project release. While somewhat more complicated than the Feature Branch Workflow, this provides a robust framework for managing larger projects.

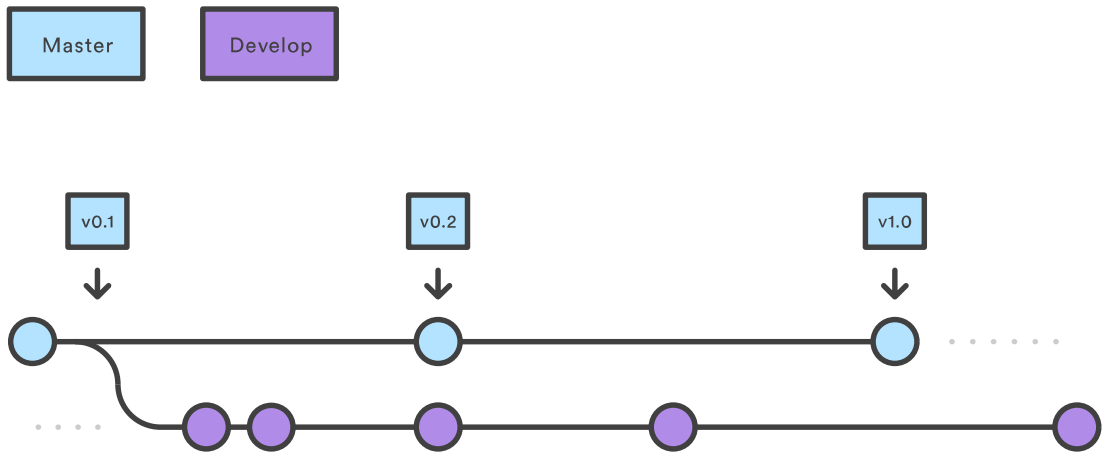
This workflow doesn't add any new concepts or commands beyond what's required for the Feature Branch Workflow. Instead, it assigns very specific roles to different branches and defines how and when they should interact. In addition to feature branches, it uses individual branches for preparing, maintaining, and recording releases. Of course, you also get to leverage all the benefits of the Feature Branch Workflow: pull requests, isolated experiments, and more efficient collaboration.

How It Works

The Gitflow Workflow still uses a central repository as the communication hub for all developers. And, as in the other workflows, developers work locally and push branches to the central repo. The only difference is the branch structure of the project.

Historical Branches

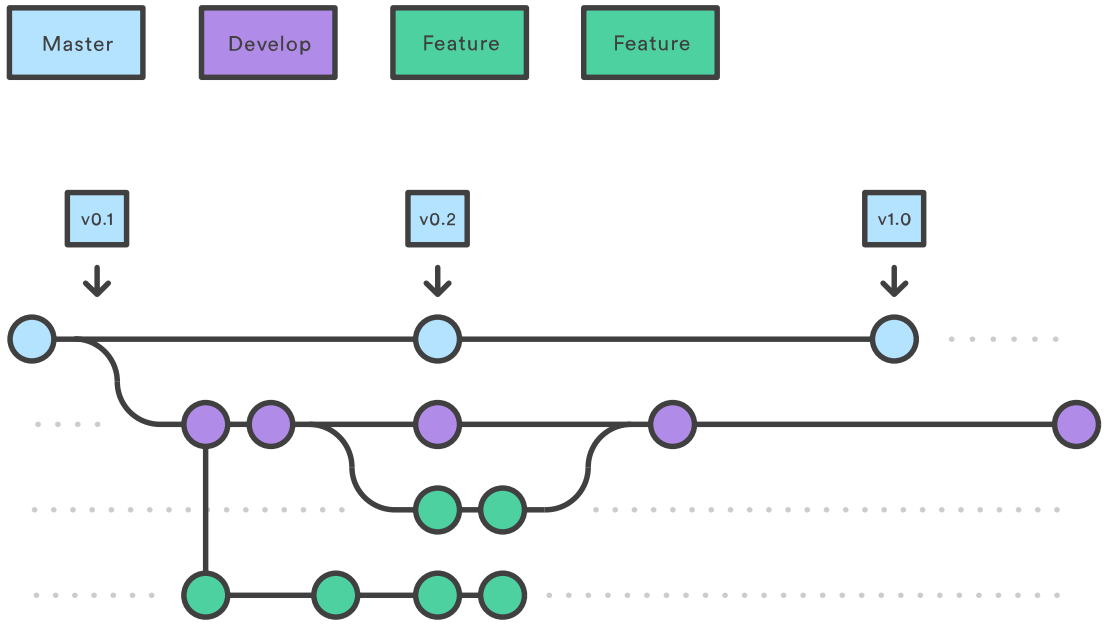
Instead of a single `master` branch, this workflow uses two branches to record the history of the project. The `master` branch stores the official release history, and the `develop` branch serves as an integration branch for features. It's also convenient to tag all commits in the `master` branch with a version number.



The rest of this workflow revolves around the distinction between these two branches.

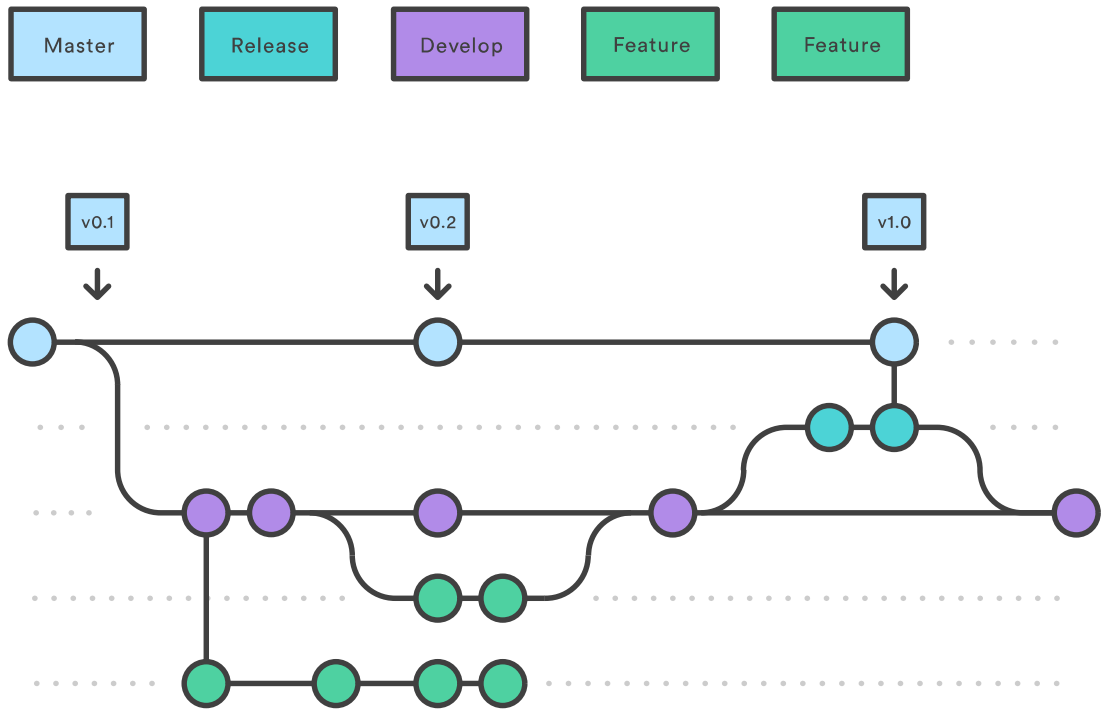
Feature Branches

Each new feature should reside in its own branch, which can be pushed to the central repository for backup/collaboration. But, instead of branching off of `master`, feature branches use develop as their parent branch. When a feature is complete, it gets merged back into `develop`. Features should never interact directly with `master`.



Note that feature branches combined with the `develop` branch is, for all intents and purposes, the Feature Branch Workflow. But, the Gitflow Workflow doesn't stop there.

Release Branches



Once `develop` has acquired enough features for a release (or a

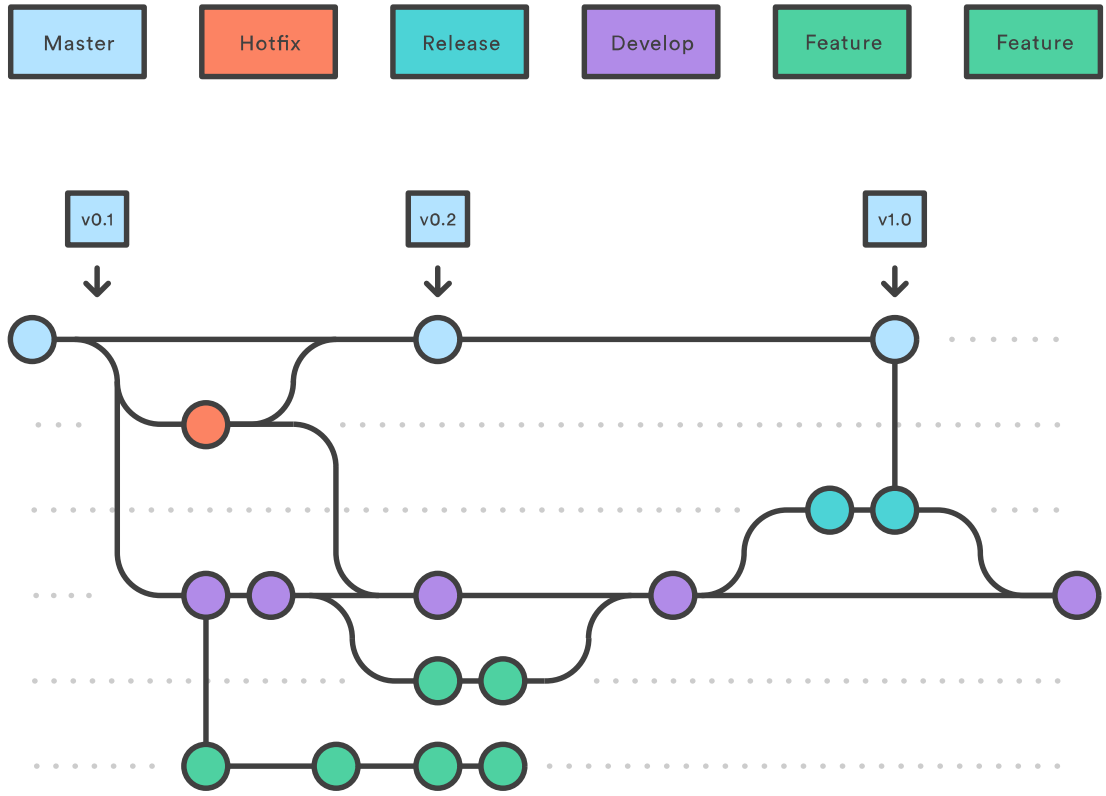
predetermined release date is approaching), you fork a release branch off of `develop`. Creating this branch starts the next release cycle, so no new features can be added after this point—only bug fixes, documentation generation, and other release-oriented tasks should go in this branch. Once it's ready to ship, the release gets merged into `master` and tagged with a version number. In addition, it should be merged back into `develop`, which may have progressed since the release was initiated.

Using a dedicated branch to prepare releases makes it possible for one team to polish the current release while another team continues working on features for the next release. It also creates well-defined phases of development (e.g., it‘s easy to say, “this week we’re preparing for version 4.0” and to actually see it in the structure of the repository).

Common conventions:

- branch off: `develop`
- merge into: `master`
- naming convention: `release-* or release/*`

Maintenance Branches



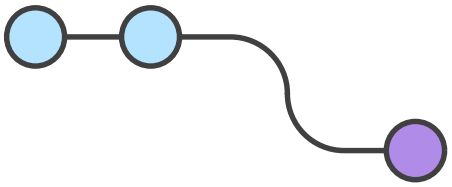
Maintenance or “hotfix” branches are used to quickly patch production releases. This is the only branch that should fork directly off of `master`. As soon as the fix is complete, it should be merged into both `master` and `develop` (or the current release branch), and `master` should be tagged with an updated version number.

Having a dedicated line of development for bug fixes lets your team address issues without interrupting the rest of the workflow or waiting for the next release cycle. You can think of maintenance branches as ad hoc release branches that work directly with `master`.

Example

The example below demonstrates how this workflow can be used to manage a single release cycle. We’ll assume you have already created a central repository.

Create a develop branch



The first step is to complement the default `master` with a `develop` branch. A simple way to do this is for one developer to create an empty `develop` branch locally and push it to the server:

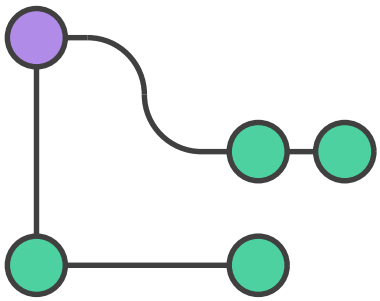
```
git branch develop
git push -u origin develop
```

This branch will contain the complete history of the project, whereas `master` will contain an abridged version. Other developers should now clone the central repository and create a tracking branch for develop:

```
git clone ssh://user@host/path/to/repo.git
git checkout -b develop origin/develop
```

Everybody now has a local copy of the historical branches set up.

Mary and John begin new features



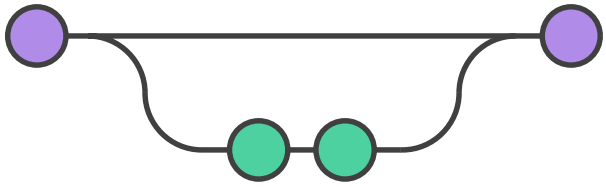
Our example starts with John and Mary working on separate features. They both need to create separate branches for their respective features. Instead of basing it on `master`, they should both base their feature branches on `develop`:

```
git checkout -b some-feature develop
```

Both of them add commits to the feature branch in the usual fashion: edit, stage, commit:

```
git status
git add <some-file>
git commit
```

Mary finishes her feature



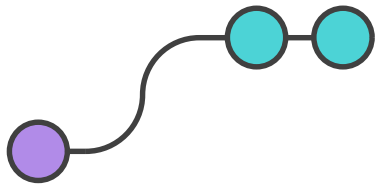
After adding a few commits, Mary decides her feature is ready. If her team is using pull requests, this would be an appropriate time to open one asking to merge her feature into `develop`. Otherwise, she can merge it into her local `develop` and push it to the central repository, like so:

```
git pull origin develop
git checkout develop
git merge some-feature
git push
git branch -d some-feature
```

The first command makes sure the `develop` branch is up to date before trying to merge in the feature. Note that features should never be merged directly into `master`. Conflicts can be resolved in the

same way as in the Centralized Workflow.

Mary begins to prepare a release



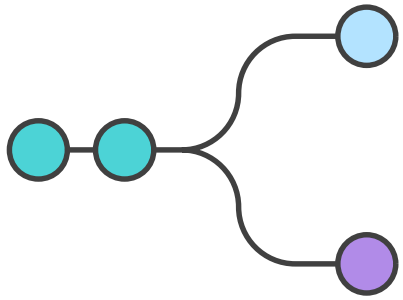
While John is still working on his feature, Mary starts to prepare the first official release of the project. Like feature development, she uses a new branch to encapsulate the release preparations. This step is also where the release's version number is established:

```
git checkout -b release-0.1 develop
```

This branch is a place to clean up the release, test everything, update the documentation, and do any other kind of preparation for the upcoming release. It's like a feature branch dedicated to polishing the release.

As soon as Mary creates this branch and pushes it to the central repository, the release is feature-frozen. Any functionality that isn't already in `develop` is postponed until the next release cycle.

Mary finishes the release



Once the release is ready to ship, Mary merges it into `master` and `develop`, then deletes the release branch. It's important to merge back into `develop` because critical updates may have been added to the release branch and they need to be accessible to new features. Again, if Mary's organization stresses code review, this would be an ideal place for a pull request.

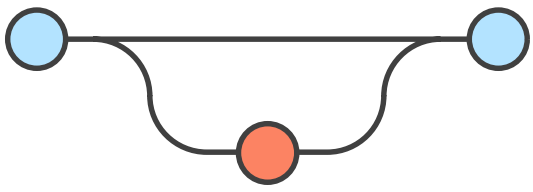
```
git checkout master
git merge release-0.1
git push
git checkout develop
git merge release-0.1
git push
git branch -d release-0.1
```

Release branches act as a buffer between feature development (`develop`) and public releases (`master`). Whenever you merge something into `master`, you should tag the commit for easy reference:

```
git tag -a 0.1 -m "Initial public release" master
git push --tags
```

Git comes with several hooks, which are scripts that execute whenever a particular event occurs within a repository. It's possible to configure a hook to automatically build a public release whenever you push the `master` branch to the central repository or push a tag.

End-user discovers a bug



After shipping the release, Mary goes back to developing features for

the next release with John. That is, until an end-user opens a ticket complaining about a bug in the current release. To address the bug, Mary (or John) creates a maintenance branch off of `master`, fixes the issue with as many commits as necessary, then merges it directly back into `master`.

```
git checkout -b issue-#001 master
# Fix the bug
git checkout master
git merge issue-#001
git push
```

Like release branches, maintenance branches contain important updates that need to be included in `develop`, so Mary needs to perform that merge as well. Then, she's free to delete the branch:

```
git checkout develop
git merge issue-#001
git push
git branch -d issue-#001
```

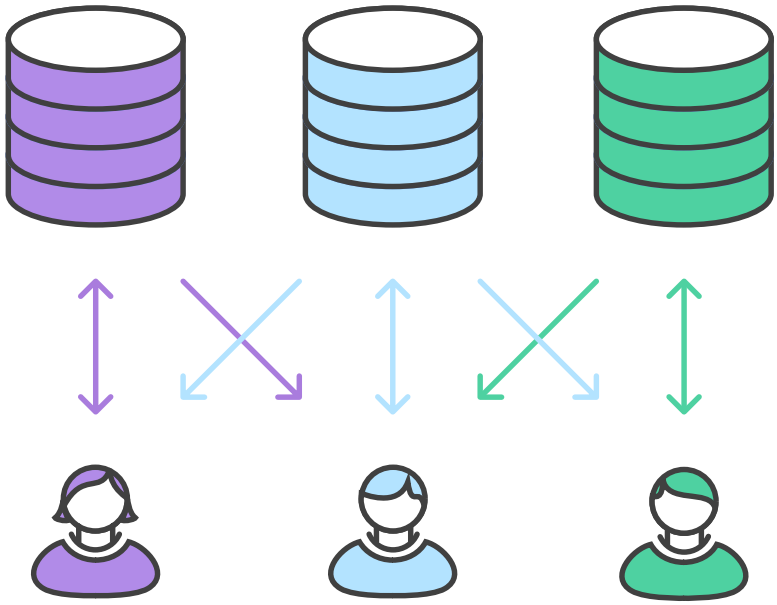
Where To Go From Here

By now, you're hopefully quite comfortable with the Centralized Workflow, the Feature Branch Workflow, and the Gitflow Workflow. You should also have a solid grasp on the potential of local repositories, the push/pull pattern, and Git's robust branching and merging model.

Remember that the workflows presented here are merely examples of what's possible—they are not hard-and-fast rules for using Git in the workplace. So, don't be afraid to adopt some aspects of a workflow and disregard others. The goal should always be to make Git work for you, not the other way around.

Forking Workflow

The Forking Workflow is fundamentally different than the other workflows discussed in this tutorial. Instead of using a single server-side repository to act as the “central” codebase, it gives *every* developer a server-side repository. This means that each contributor has not one, but two Git repositories: a private local one and a public server-side one.



The main advantage of the Forking Workflow is that contributions can be integrated without the need for everybody to push to a single central repository. Developers push to *their own* server-side repositories, and only the project maintainer can push to the official repository. This allows the maintainer to accept commits from any developer without giving them write access to the official codebase.

The result is a distributed workflow that provides a flexible way for large, organic teams (including untrusted third-parties) to collaborate securely. This also makes it an ideal workflow for open source projects.

How It Works

As in the other Git workflows, the Forking Workflow begins with an official public repository stored on a server. But when a new developer wants to start working on the project, they do not directly clone the official repository.

Instead, they **fork** the official repository to create a copy of it on the server. This new copy serves as their personal public repository—no other developers are allowed to push to it, but they can pull changes from it (we'll see why this is important in a moment). After they have created their server-side copy, the developer performs a `git clone` to get a copy of it onto their local machine. This serves as their private development environment, just like in the other workflows.

When they're ready to publish a local commit, they push the commit to their own public repository—not the official one. Then, they file a pull request with the main repository, which lets the project maintainer know that an update is ready to be integrated. The pull request also serves as a convenient discussion thread if there are issues with the contributed code.

To integrate the feature into the official codebase, the maintainer pulls the contributor's changes into their local repository, checks to make sure it doesn't break the project, merges it into his local `master` branch, then pushes the `master` branch to the official repository on the server. The contribution is now part of the project, and other developers should pull from the official repository to synchronize their local repositories.

The Official Repository

It's important to understand that the notion of an "official" repository in the Forking Workflow is merely a convention. From a technical

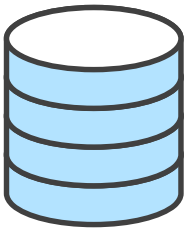
standpoint, Git doesn't see any difference between each developer's public repository and the official one. In fact, the only thing that makes the official repository so official is that it's the public repository of the project maintainer.

Branching in the Forking Workflow

All of these personal public repositories are really just a convenient way to share branches with other developers. Everybody should still be using branches to isolate individual features, just like in the Feature Branch Workflow and the Gitflow Workflow. The only difference is how those branches get shared. In the Forking Workflow, they are pulled into another developer's local repository, while in the Feature Branch and Gitflow Workflows they are pushed to the official repository.

Example

The project maintainer initializes the official repository



As with any Git-based project, the first step is to create an official repository on a server accessible to all of the team members. Typically, this repository will also serve as the public repository of the project maintainer.

Public repositories should always be bare, regardless of whether they represent the official codebase or not. So, the project maintainer should run something like the following to set up the official repository:

```
ssh user@host
git init --bare /path/to/repo.git
```

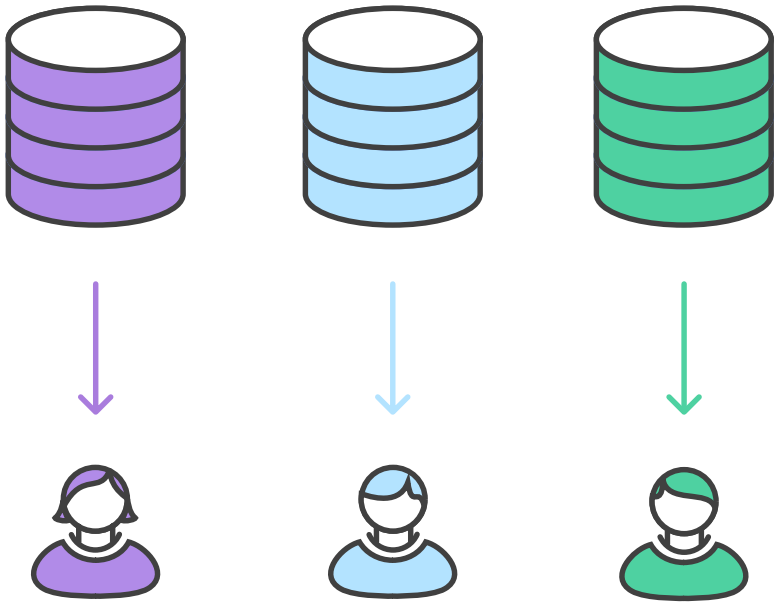
Bitbucket and Stash also provide a convenient GUI alternative to the above commands. This is the exact same process as setting up a central repository for the other workflows in this tutorial. The maintainer should also push the existing codebase to this repository, if necessary.

Developers fork the official repository

Next, all of the other developers need to fork this official repository. It's possible to do this by SSH'ing into the server and running `git clone` to copy it to another location on the server—yes, forking is basically just a server-side clone. But again, Bitbucket and Stash let developers fork a repository with the click of a button.

After this step, every developer should have their own server-side repository. Like the official repository, all of these should be bare repositories.

Developers clone their forked repositories



Next each developer needs to clone their own public repository. They can do with the familiar `git clone command`.

Our example assumes the use of Bitbucket to host these repositories. Remember, in this situation, each developer should have their own Bitbucket account and they should clone their server-side repository using:

```
git clone https://user@bitbucket.org/user/repo.git
```

Whereas the other workflows in this tutorial use a single `origin` remote that points to the central repository, the Forking Workflow requires two remotes—one for the official repository, and one for the developer's personal server-side repository. While you can call these remotes anything you want, a common convention is to use `origin` as the remote for your forked repository (this will be created automatically when you run `git clone`) and `upstream` for the official repository.

```
git remote add upstream https://bitbucket.org/maintainer/rep
```

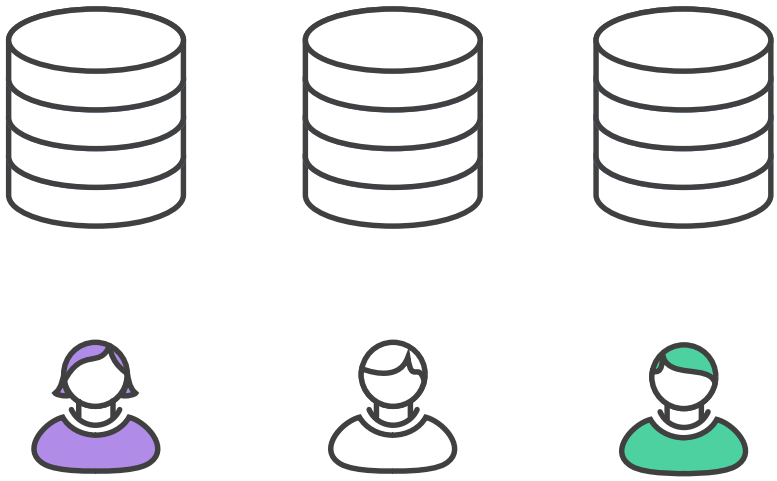
You'll need to create the upstream remote yourself using the above

command. This will let you easily keep your local repository up-to-date as the official project progresses. Note that if your upstream repository has authentication enabled (i.e., it's not open source), you'll need to supply a username, like so:

```
git remote add upstream https://user@bitbucket.org/maintaine
```

This requires users to supply a valid password before cloning or pulling from the official codebase.

Developers work on their features



In the local repositories that they just cloned, developers can edit code, commit changes, and create branches just like they did in the other workflows:

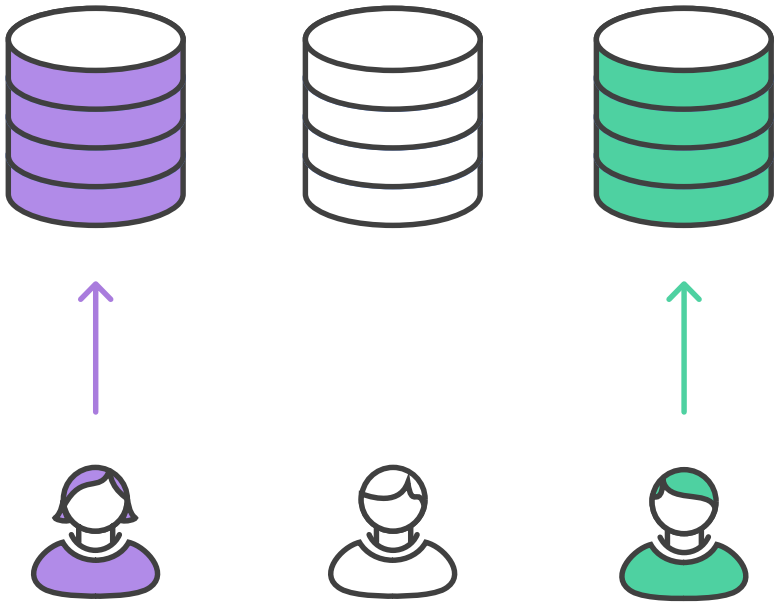
```
git checkout -b some-feature
# Edit some code
git commit -a -m "Add first draft of some feature"
```

All of their changes will be entirely private until they push it to their public repository. And, if the official project has moved forward, they can access new commits with `git pull`:

```
git pull upstream master
```

Since developers should be working in a dedicated feature branch, this should generally result in a fast-forward merge.

Developers publish their features



Once a developer is ready to share their new feature, they need to do two things. First, they have to make their contribution accessible to other developers by pushing it to their public repository. Their `origin` remote should already be set up, so all they should have to do is the following:

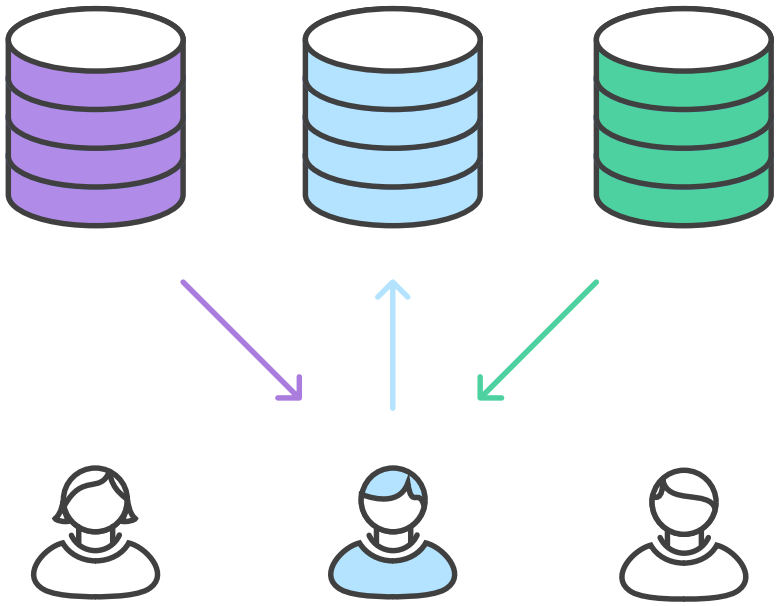
```
git push origin feature-branch
```

This diverges from the other workflows in that the `origin` remote points to the developer's personal server-side repository, not the main codebase.

Second, they need to notify the project maintainer that they want to merge their feature into the official codebase. Bitbucket and Stash

provide a “Pull request” button that leads to a form asking you to specify which branch you want to merge into the official repository. Typically, you’ll want to integrate your feature branch into the upstream remote’s `master` branch.

The project maintainer integrates their features



When the project maintainer receives the pull request, their job is to decide whether or not to integrate it into the official codebase. They can do this in one of two ways:

1. Inspect the code directly in the pull request

2. Pull the code into their local repository and manually merge it

The first option is simpler, as it lets the maintainer view a diff of the changes, comment on it, and perform the merge via a graphical user interface. However, the second option is necessary if the pull request results in a merge conflict. In this case, the maintainer needs to fetch the feature branch from the developer’s server-side repository, merge

it into their local `master` branch, and resolve any conflicts:

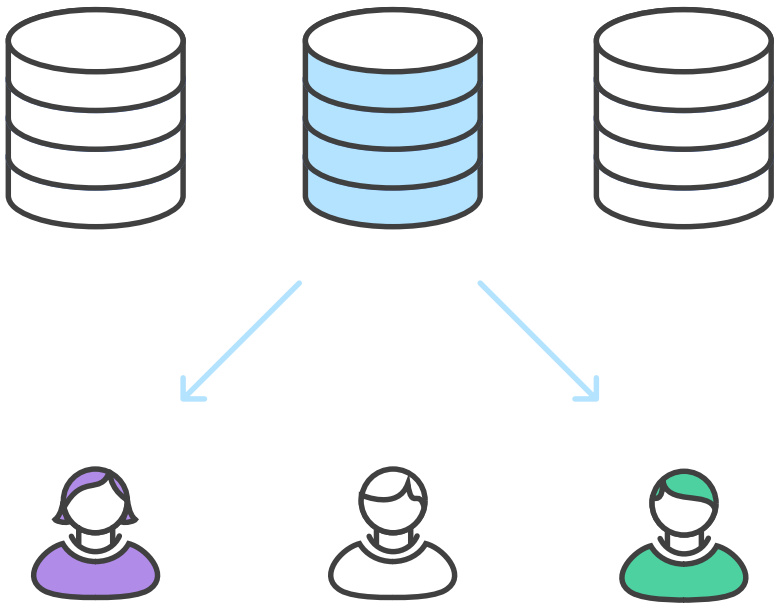
```
git fetch https://bitbucket.org/user/repo feature-branch
# Inspect the changes
git checkout master
git merge FETCH_HEAD
```

Once the changes are integrated into their local `master`, the maintainer needs to push it to the official repository on the server so that other developers can access it:

```
git push origin master
```

Remember that the maintainer's `origin` points to their public repository, which also serves as the official codebase for the project. The developer's contribution is now fully integrated into the project.

Developers synchronize with the official repository



Since the main codebase has moved forward, other developers should synchronize with the official repository:

```
git pull upstream master
```

Where To Go From Here

If you're coming from an SVN background, the Forking Workflow may seem like a radical paradigm shift. But don't be afraid—all it's really doing is introducing another level of abstraction on top of the Feature Branch Workflow. Instead of sharing branches directly though a single central repository, contributions are published to a server-side repository dedicated to the originating developer.

This article explained how a contribution flows from one developer into the official `master` branch, but the same methodology can be used to integrate a contribution into any repository. For example, if one part of your team is collaborating on a particular feature, they can share changes amongst themselves in the exact same manner —without touching the main repository.

This makes the Forking Workflow a very powerful tool for loosely-knit teams. Any developer can easily share changes with any other developer, and any branch can be efficiently merged into the official codebase.