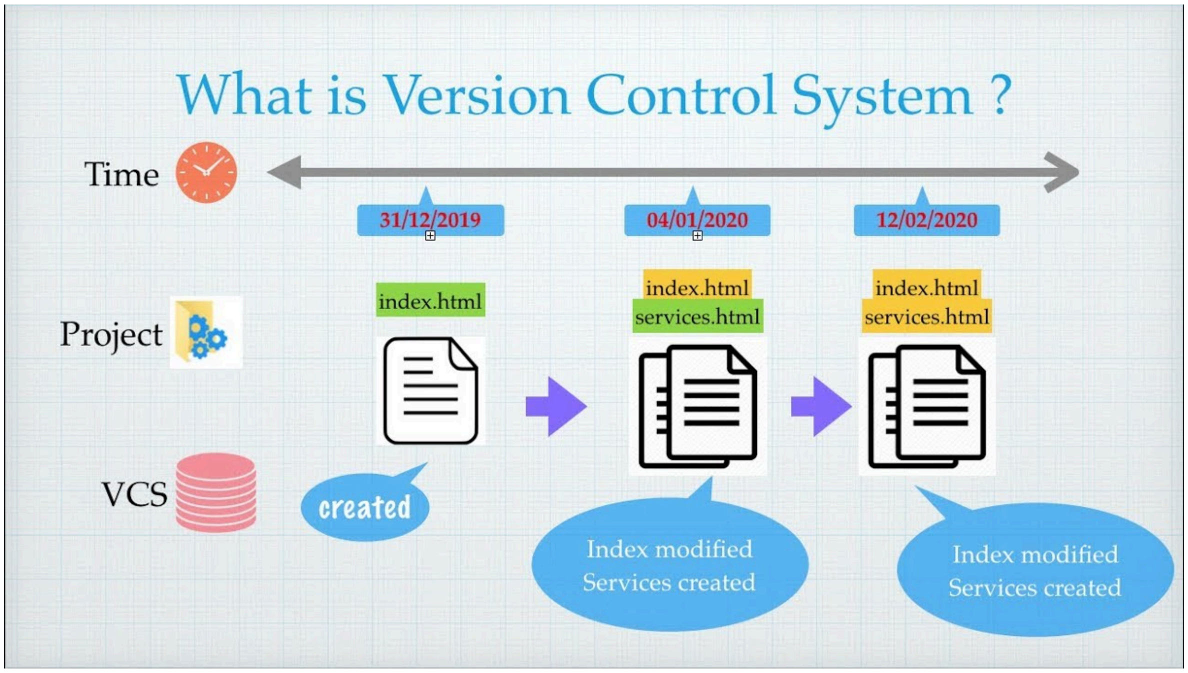
# Version Control Systems

Detailed documentation of Git

kevin comba

# Table of Contents

# VERSION CONTROL



1746388980610

Prepared by Kevin Comba

Version control, also known as source control, is the practice of tracking and managing changes to software code. Version control systems are software tools that help software teams manage changes to source code over time.

# Introduction

## What is GIT?

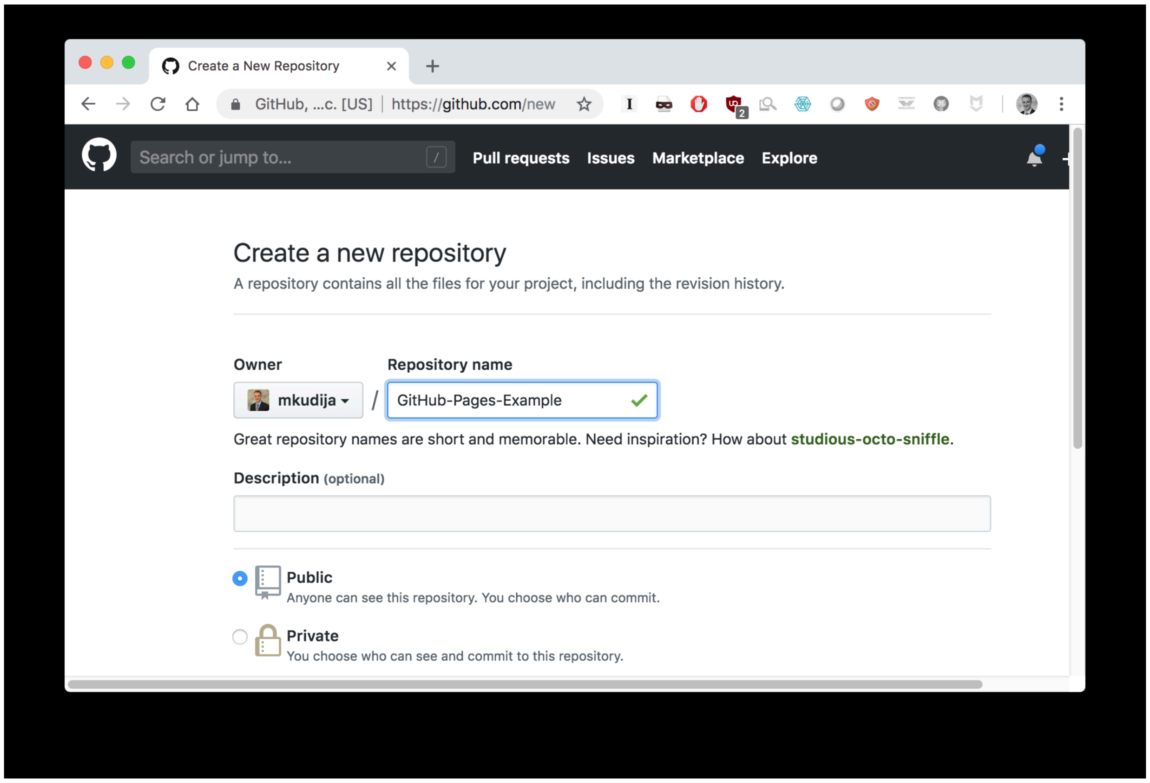
1746383652587

Git is a distributed version control system designed to track changes in source code during software development. Unlike centralized systems, Git allows each developer to have a full-fledged repository with complete history and full version tracking capabilities, independent of network access or a central server.

By far, the most widely used modern version control system in the world today is Git. Git is a mature, actively maintained open source project originally developed in 2005 by Linus Torvalds, the famous creator of the Linux operating system kernel

### Online platforms using Git?

- GITHUB



Create a new repository

A repository contains all the files for your project, including the revision history.

**Owner** mkudija / **Repository name** GitHub-Pages-Example ✓

Great repository names are short and memorable. Need inspiration? How about **studious-octo-sniffle**.

**Description** (optional)

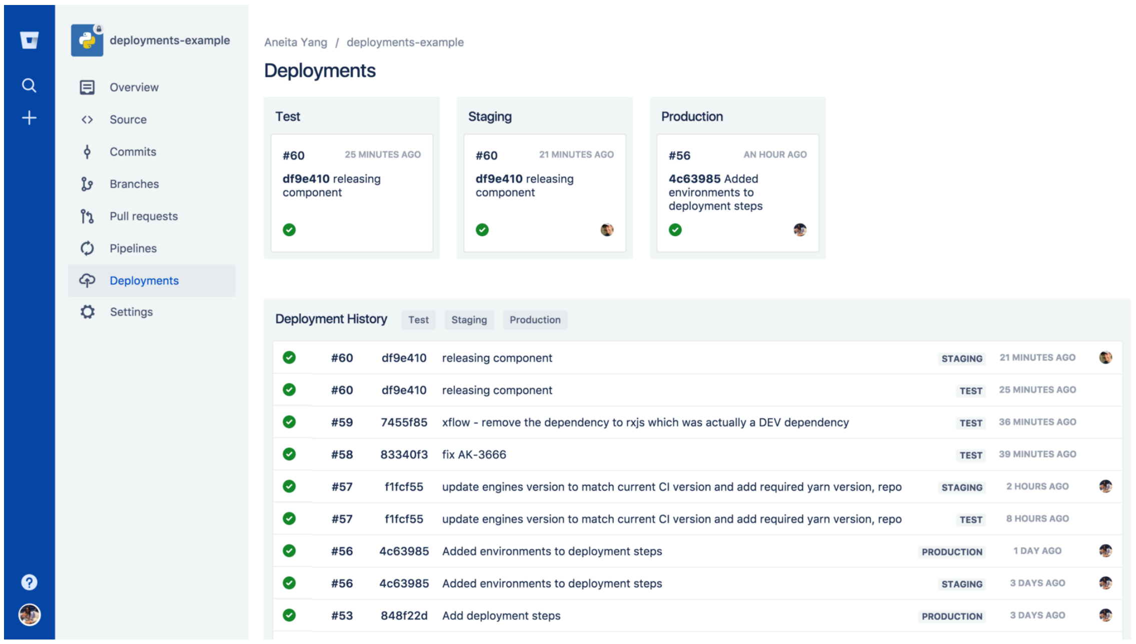
[● **Public**] Anyone can see this repository. You choose who can commit.

**Private** You choose who can see and commit to this repository.

1746393067975

The most popular platform, known for its vast open-source community and integration with GitHub Actions for CI/CD.

- BITBUCKET



Deployments

| Test | Staging | Production |
|---|---|---|
| #60 25 MINUTES AGO df9e410 releasing component ✓ | #60 21 MINUTES AGO df9e410 releasing component ✓ | #56 AN HOUR AGO 4c63985 Added environments to deployment steps ✓ |

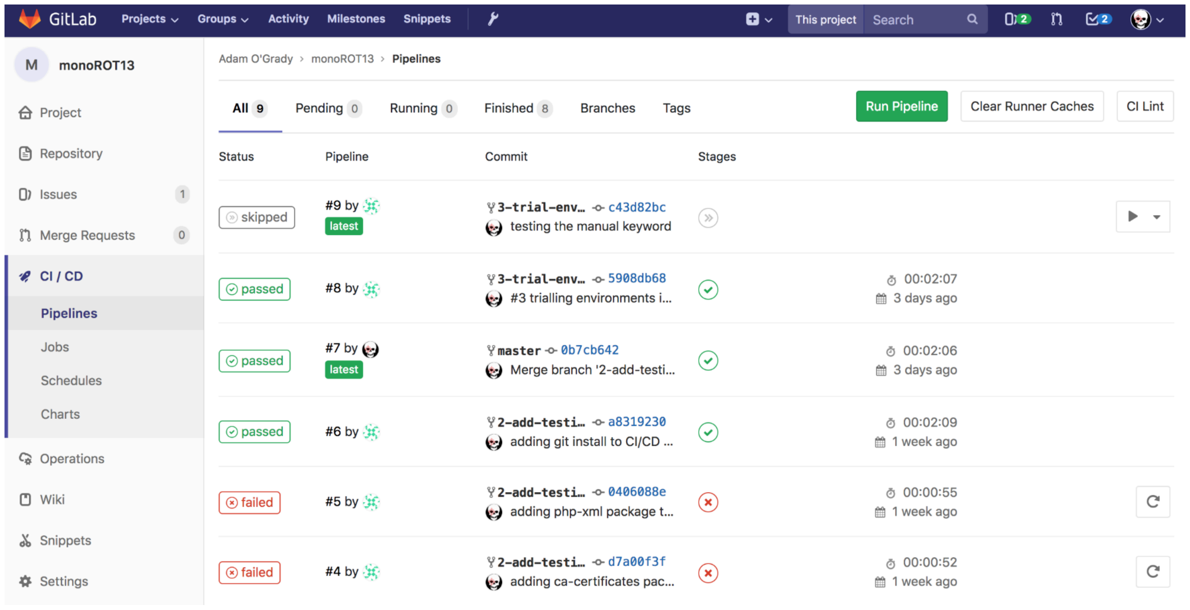
Deployment History Test Staging Production

| | | | | | |
|---|---|---|---|---|---|
| ✓ | #60 | df9e410 | releasing component | STAGING | 21 MINUTES AGO |
| ✓ | #60 | df9e410 | releasing component | TEST | 25 MINUTES AGO |
| ✓ | #59 | 7455f85 | xflow - remove the dependency to rxjs which was actually a DEV dependency | TEST | 36 MINUTES AGO |
| ✓ | #58 | 83340f3 | fix AK-3666 | TEST | 39 MINUTES AGO |
| ✓ | #57 | f1fcf55 | update engines version to match current CI version and add required yarn version, repo | STAGING | 2 HOURS AGO |
| ✓ | #57 | f1fcf55 | update engines version to match current CI version and add required yarn version, repo | TEST | 8 HOURS AGO |
| ✓ | #56 | 4c63985 | Added environments to deployment steps | PRODUCTION | 1 DAY AGO |
| ✓ | #56 | 4c63985 | Added environments to deployment steps | STAGING | 3 DAYS AGO |
| ✓ | #53 | 848f22d | Add deployment steps | PRODUCTION | 3 DAYS AGO |

1746393154178

Integrated with Atlassian tools like Jira and Trello, supporting both Git and Mercurial repositories.

- GITLAB

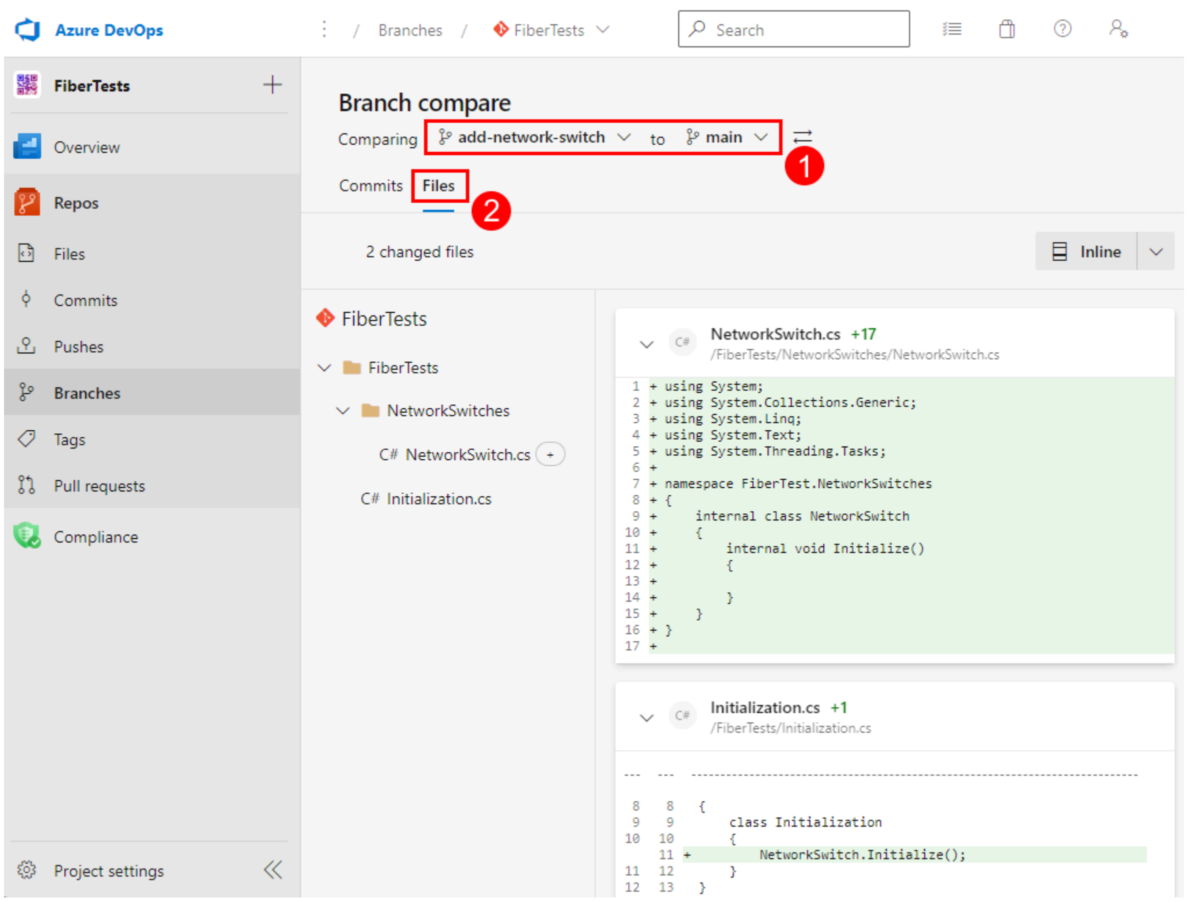


| Status | Pipeline | Commit | Stages | |
|---|---|---|---|---|
| skipped | #9 by latest | 3-trial-env… c43d82bc testing the manual keyword | | |
| passed | #8 by | 3-trial-env… 5908db68 #3 trialling environments i… | ✓ | 00:02:07 3 days ago |
| passed | #7 by latest | master 0b7cb642 Merge branch '2-add-testi… | ✓ | 00:02:06 3 days ago |
| passed | #6 by | 2-add-testi… a8319230 adding git install to CI/CD … | ✓ | 00:02:09 1 week ago |
| failed | #5 by | 2-add-testi… 0406088e adding php-xml package t… | ✗ | 00:00:55 1 week ago |
| failed | #4 by | 2-add-testi… d7a00f3f adding ca-certificates pac… | ✗ | 00:00:52 1 week ago |

1746388942493

Offers built-in CI/CD pipelines, issue tracking, and is available as both a cloud service and a self-hosted solution.

- Azure Repos



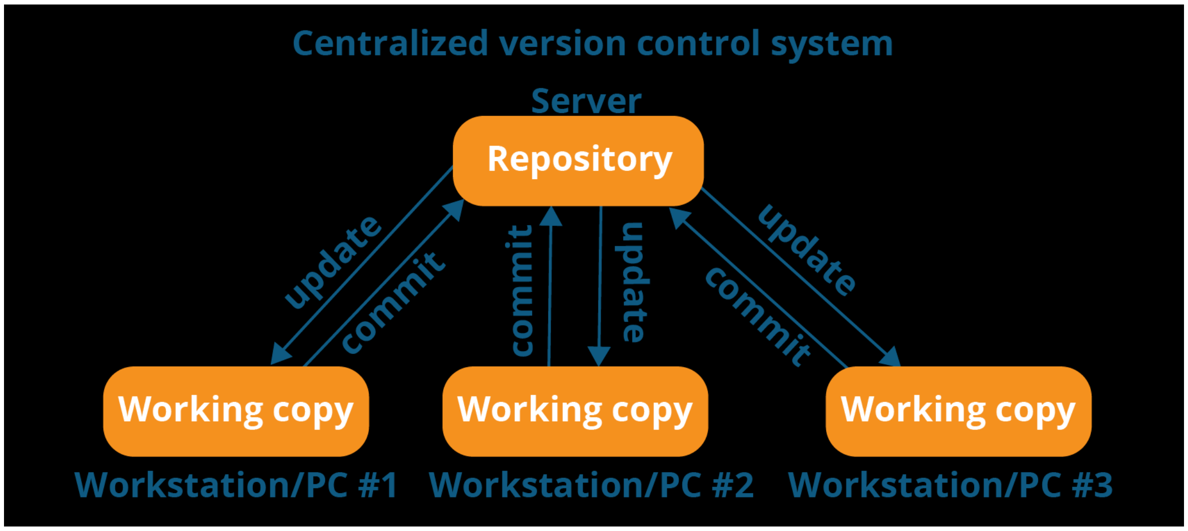
```
+ using System;
+ using System.Collections.Generic;
+ using System.Linq;
+ using System.Text;
+ using System.Threading.Tasks;
+
+ namespace FiberTest.NetworkSwitches
+ {
+     internal class NetworkSwitch
+     {
+         internal void Initialize()
+         {
+
+         }
+     }
+ }
+
```

```
  {
      class Initialization
      {
+         NetworkSwitch.Initialize();
      }
  }
```

1746388842969

Part of Microsoft's Azure DevOps suite, providing Git repositories with advanced security and collaboration features.

## Why use Git for your organization?



1746389039998

- **Collaboration**: Facilitates multiple developers working on the same project simultaneously.
- **Branching and Merging**: Allows for feature development, bug fixes, and experiments in isolated branches.
- **History Tracking**: Keeps a detailed history of changes, aiding in debugging and understanding project evolution.
- **Distributed Nature**: Each developer has a complete copy of the repository, enhancing reliability and speed

## How to install Git?

**Git can be installed on various operating systems:**

- **Windows**: Download the installer from git-scm.com (https://git-scm.com/download/win) and follow the setup instructions.
- **macOS**: Use Homebrew with the command `brew install git`, or download from git-scm.com (https://git-scm.com/download/mac).
- **Linux**: Use package managers like `apt` for Ubuntu (`sudo apt install git`) or `dnf` for Fedora.

# Getting Started

## Configuring git on our local machine

**Set your user name and email address, which will be associated with your commits:**

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

### Setting up a new repository

- `git init`: Initialize a new repository

  ```
  git init
  ```

- `git clone`: Clone an existing repository

  ```
  git clone <repository-url>
  ```

- **`git status`**: Shows the current status of the working directory (modified, staged files, etc.). Example :

### Saving changes a new file

#### 1. Initialize a Git Repository (if not already done)

If your project isn't already a Git repository, navigate to your project directory and run:

```
git init
```

This command initializes a new Git repository in your current directory.

#### 2. Create or Add a New File

Create a new file using your preferred text editor or the command line. For example, to create a new file named `example.txt` with some content:

```
git add example.txt
```

**3. Check the Status of Your Repository**

To see the current state of your repository and check for any untracked files, run:

```
git status
```

This command will display information about untracked files, changes not staged for commit, and changes to be committed.

**4. Stage the New File**

To add the new file to the staging area, use the `git add` command:

```
git add example.txt
```

This command stages the file, preparing it to be committed.

**5. Commit the Staged File**

Once the file is staged, commit it to the repository with a descriptive message:

```
git commit -m "Add example.txt file"
```

This command records the changes in the repository's history.

**6. Push the Commit to a Remote Repository (Optional)**

If you're working with a remote repository (e.g., on GitHub), push your commit to the remote server:

```
git push origin main
```

By following these steps, you can effectively save a new file in your Git repository.

## Tags and Versioning

- **git tag <tag_name>**: Adds a tag to mark a specific point in the history (e.g., for versioning). **Example**:

```
git tag v1.0.0
```

- **git push origin --tags**: Pushes all tags to the remote repository. **Example**:

```
git push origin --tags
```

## Undoing changes

**git clean :**

The `git clean` command is used to remove untracked files from your working directory. This is particularly useful when you have files that are not being tracked by Git and you want to clean up your working directory.

**Usage:**

- **Dry Run (Preview Files to Be Removed):**

  ```
  git clean -n
  ```

This command will list the files that would be removed without actually deleting them.

- **Remove Untracked Files:**

  ```
  git clean -f
  ```

This command will remove untracked files from your working directory.

- **Remove Untracked Files and Directories:**

  ```
  git clean -fd
  ```

This command will remove both untracked files and untracked directories.

- **Remove Ignored Files:**

```
git clean -fX
```

This command will remove all ignored files. Be cautious with this option as it will delete files listed in your `.gitignore`.

**Warning:** `git clean` permanently deletes files. Use it cautiously, especially if there are untracked files you may want to keep.

**git revert :**

The `git revert` command is used to create a new commit that undoes the changes made by a previous commit. This is useful for undoing changes in a public history without rewriting the commit history.

**Usage:**

```
git revert <commit-hash>
```

Replace `<commit-hash>` with the hash of the commit you want to revert.

This command will create a new commit that undoes the changes introduced by the specified commit.

**Example:**

```
git revert a1b2c3d
```

This will revert the changes introduced by commit `a1b2c3d`.

**git reset :**

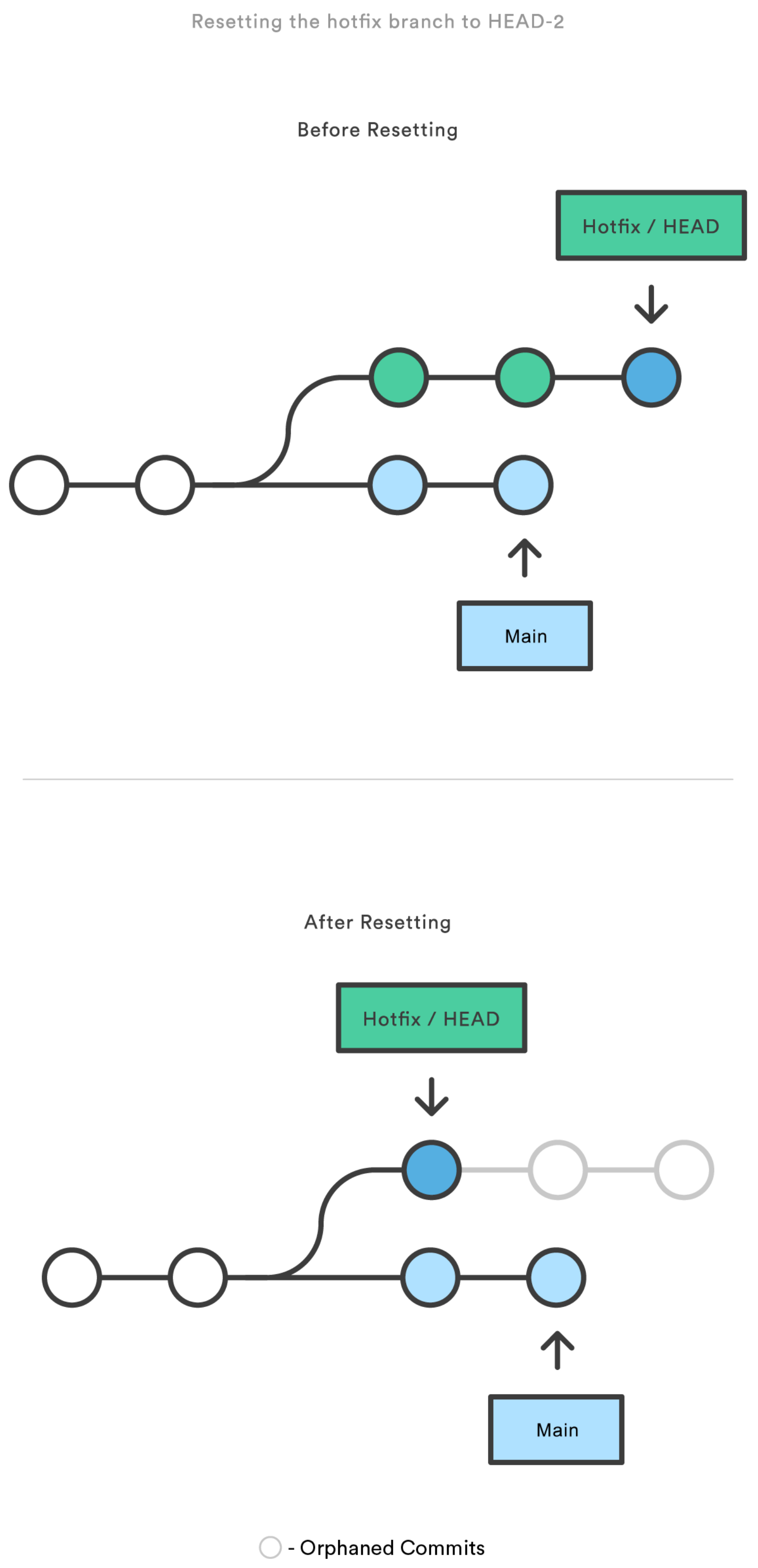
The `git reset` command is used to reset your current HEAD to a specified state. It has three primary modes:

- Reset a specific commit

  On the commit-level, resetting is a way to move the tip of a branch to a different commit. This can be used to remove commits from the current branch. For example, the following command moves the `hotfix` branch backwards by two commits.

```
git checkout hotfix git reset HEAD~2
```

The two commits that were on the end of `hotfix` are now dangling, or orphaned commits. This means they will be deleted the next time Git performs a garbage collection. In other words, you're saying that you want to throw away these commits. This can be visualized as the following:



Resetting the hotfix branch to HEAD-2

This usage of `git reset` is a simple way to undo changes that haven't been shared with anyone else. It's your go-to command when you've started working on a feature and find yourself thinking, "Oh crap, what am I doing? I should just start over."

In addition to moving the current branch, you can also get `git reset` to alter the staged snapshot and/or the working directory by passing it one of the following flags:

**Soft Reset (`--soft`)**: Moves HEAD to the specified commit and stages the changes.

```
git reset --soft <commit-hash>
```

- **Mixed Reset (`--mixed`)**: Moves HEAD to the specified commit and unstages the changes. This is the default mode.

```
git reset --mixed <commit-hash>
```

- **Hard Reset (`--hard`)**: Moves HEAD to the specified commit and resets the working directory and staging area to match. This discards all changes.

```
git reset --hard <commit-hash>
```

**Warning:** `git reset --hard` will discard all changes in your working directory and staging area. Use it with caution.

- **`git reset --soft HEAD~1`**: Undoes the last commit but keeps the changes staged.
- **`git reset --hard HEAD~1`**: Undoes the last commit and discards all changes.

**git rm :**

The `git rm` command is used to remove files from both your working directory and the staging area. This is useful when you want to delete a file from your repository.

**Usage:**

- **Remove a File:**

  ```
  git rm <file-name>
  ```

This command will remove the specified file from both the working directory and the staging area.

- **Remove a File from Staging Area Only:**

  ```
  git rm --cached <file-name>
  ```

This command will remove the specified file from the staging area but leave it in the working directory.

**Undoing a `git rm` Operation:**

- **Before Committing:** If you haven't committed the removal yet, you can restore the file using:

  ```
  git restore <file-name>
  ```

- **After Committing:** If you've already committed the removal, you can revert the commit using:

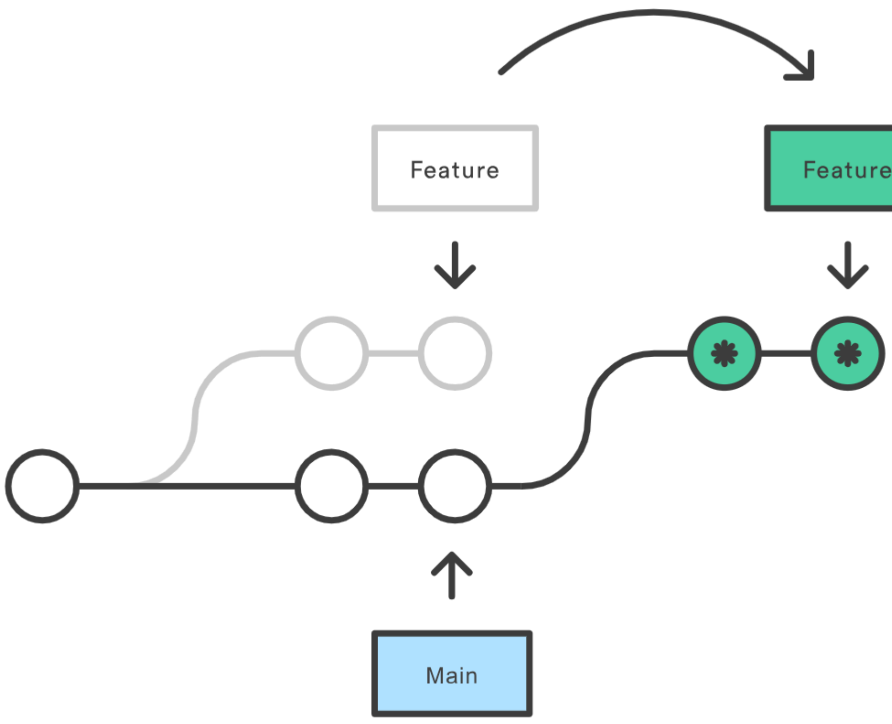
  ```
  git revert <commit-hash>
  ```

Replace `<commit-hash>` with the hash of the commit that removed the file.

**Note:** Always ensure you have backups or have committed your changes before using commands like `git reset --hard` or `git clean -f`, as they can permanently delete data.

## Rewritting History

- `git rebase`: Rebasing is the process of moving or combining a sequence of commits to a new base commit. Rebasing is most useful and easily visualized in the context of a feature branching workflow. The general process can be visualized as the following:



1746393331502

**Don't rebase public history**

You should never rebase commits once they've been pushed to a public repository. The rebase would replace the old commits with new ones and it would look like that part of your project history abruptly vanished.

# Collaborating workflows (syncing)

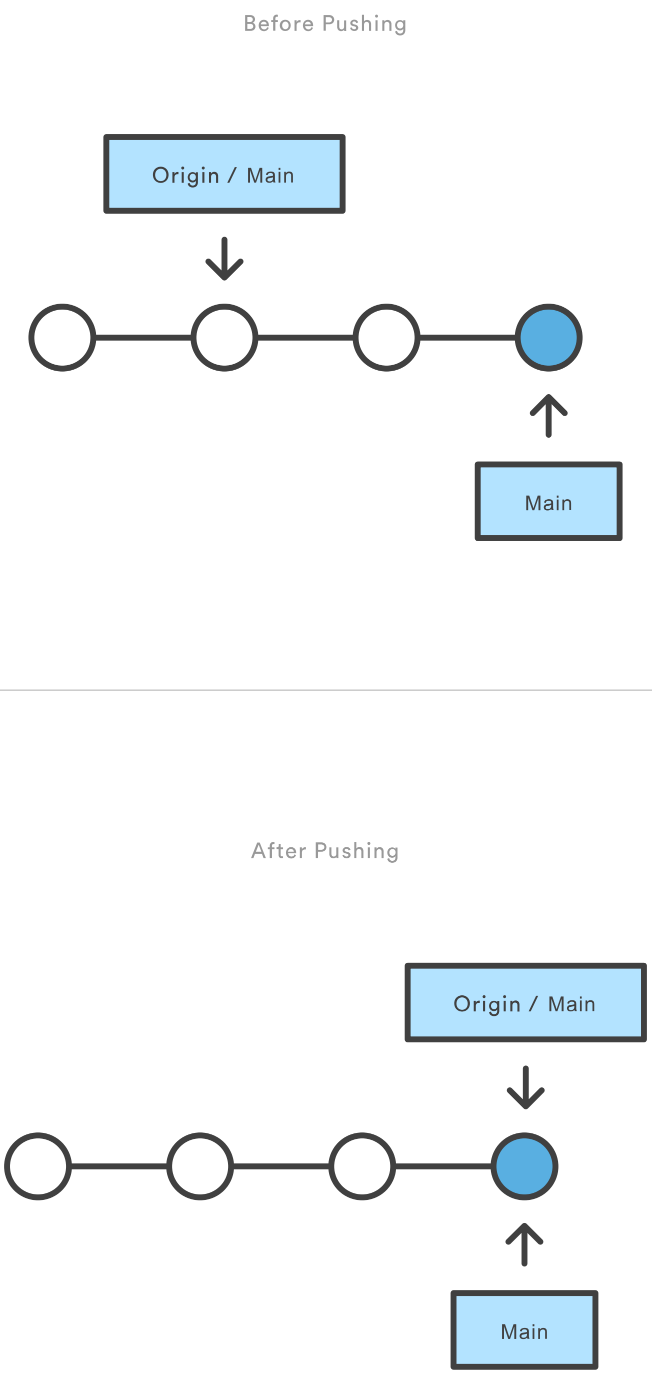
## git fetch :

The `git fetch` command downloads commits, files, and refs from a remote repository into your local repo

- `git fetch <remote>`: Fetch all of the branches from the repository. This also downloads all of the required commits and files from the other repository.
- `git fetch <remote> <branch>`: Same as the above command, but only fetch the specified branch.
- `git fetch --all`: A power move which fetches all registered remotes and their branches

## git push :

is most commonly used to publish an upload local changes to a central repository. After a local repository has been modified a push is executed to share the modifications with remote team members.



Using git push to publish changes

The above diagram shows what happens when your local `main` has progressed past the central repository's `main` and you publish changes by running `git push origin main`

The `git push` command is used to upload local repository content to a remote repository

- `git push <remote> <branch>`: Push the specified branch to , along with all of the necessary commits and internal objects. This creates a local branch in the destination repository.
- `git push <remote> --force`: Same as the above command, but force the push even if it results in a non-fast-forward merge
- Deleting a remote branch or tag : The fully delete a branch, it must be deleted locally and also remotely.

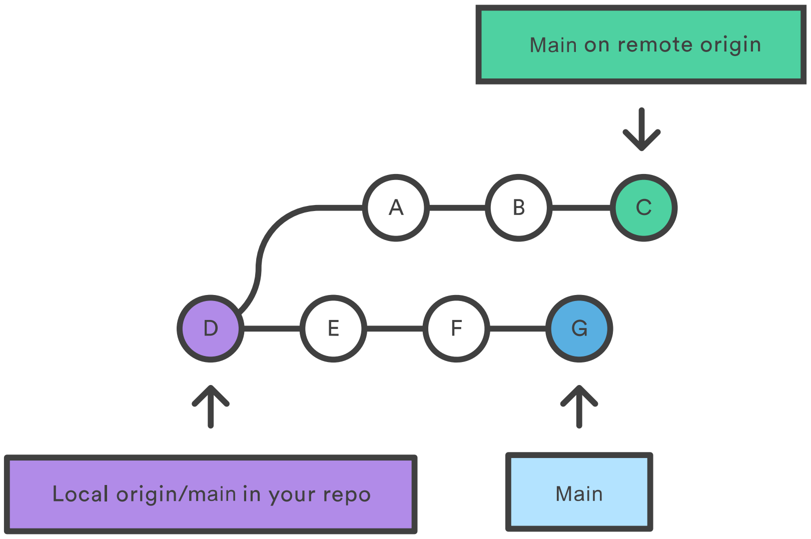
  ```
  git branch -D branch_name
  git push origin :branch_name
  ```

## git pull :

The `git pull` command is used to fetch and download content from a remote repository and immediately update the local repository to match that content.

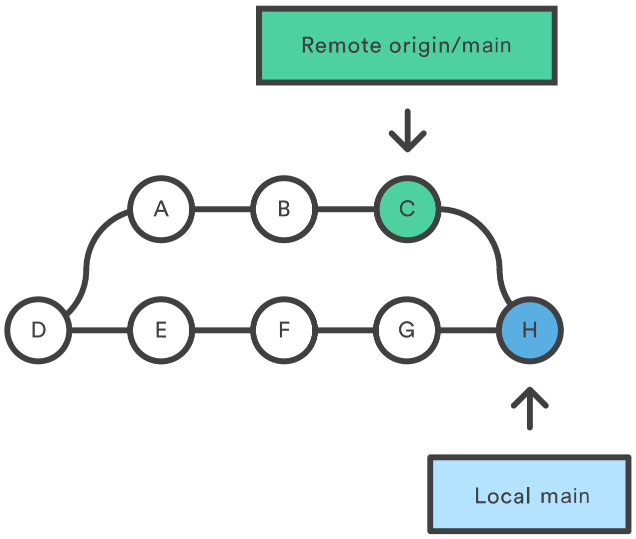
### How it works

The `git pull` command first runs `git fetch` which downloads content from the specified remote repository. Then a `git merge` is executed to merge the remote content refs and heads into a new local merge commit. To better demonstrate the pull and merging process let us consider the following example. Assume we have a repository with a main branch and a remote origin.



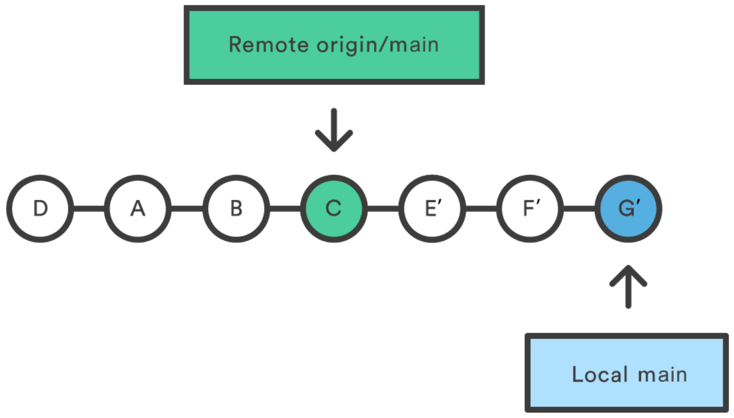
01 bubble diagram 01

In this scenario, git pull will download all the changes from the point where the local and main diverged. In this example, that point is E. git pull will fetch the diverged remote commits which are A-B-C. The pull process will then create a new local merge commit containing the content of the new diverged remote commits.



img

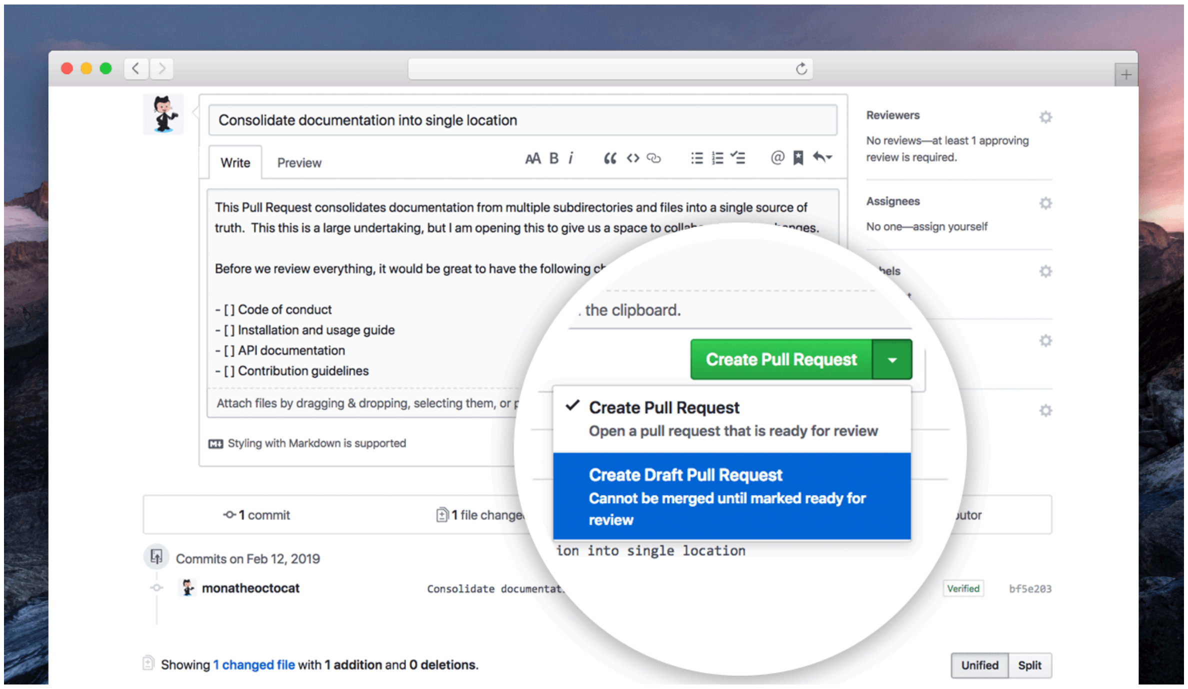
In the above diagram, we can see the new commit H. This commit is a new merge commit that contains the contents of remote A-B-C commits and has a combined log message. This example is one of a few `git pull` merging strategies. A `--rebase` option can be passed to `git pull` to use a rebase merging strategy instead of a merge commit. The next example will demonstrate how a rebase pull works. Assume that we are at a starting point of our first diagram, and we have executed `git pull --rebase`.



Central git repo to local git repo

In this diagram, we can now see that a rebase pull does not create the new H commit. Instead, the rebase has copied the remote commits A--B--C and rewritten the local commits E--F--G to appear after them them in the local origin/main commit history.

## Making a pull request(PR)


Consolidate documentation into single location

This Pull Request consolidates documentation from multiple subdirectories and files into a single source of truth. This this is a large undertaking, but I am opening this to give us a space to collab… …anges.

Before we review everything, it would be great to have the following c…

- [] Code of conduct
- [] Installation and usage guide
- [] API documentation
- [] Contribution guidelines

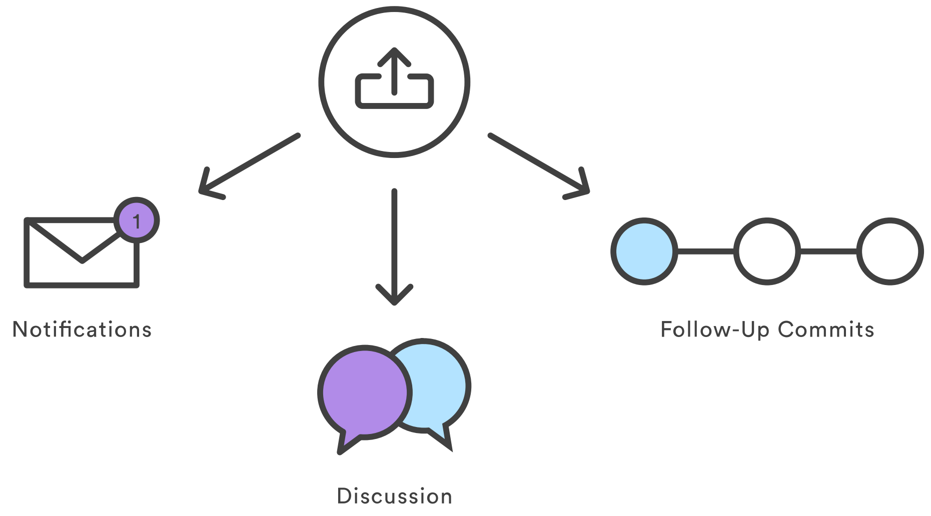
Write
Preview
Attach files by dragging & dropping, selecting them, or p
Styling with Markdown is supported
Reviewers
No reviews—at least 1 approving review is required.
Assignees
No one—assign yourself
…bels
the clipboard.
Create Pull Request
Create Pull Request
Open a pull request that is ready for review
Create Draft Pull Request
Cannot be merged until marked ready for review
1 commit
1 file change
…butor
Commits on Feb 12, 2019
monatheoctocat
Consolidate documentat… …ion into single location
Verified
bf5e203
Showing 1 changed file with 1 addition and 0 deletions.
Unified
Split

1746394977375

Pull requests are a feature that makes it easier for developers to collaborate using Bitbucket, GitHub, Azure Repos and Gitlab. They provide a user-friendly web interface for discussing proposed changes before integrating them into the official project.

In their simplest form, pull requests are a mechanism for a developer to notify team members that they have completed a feature. Once their feature branch is ready, the developer files a pull request via their Bitbucket account. This lets everybody involved know that they need to review the code and merge it into the `main` branch.

But, the pull request is more than just a notification—it's a dedicated forum for discussing the proposed feature. If there are any problems with the changes, teammates can post feedback in the pull request and even tweak the feature by pushing follow-up commits. All of this activity is tracked directly inside of the pull request.



Git workflows: Activity inside a pull request

Compared to other collaboration models, this formal solution for sharing commits makes for a much more streamlined workflow.

## How PR works

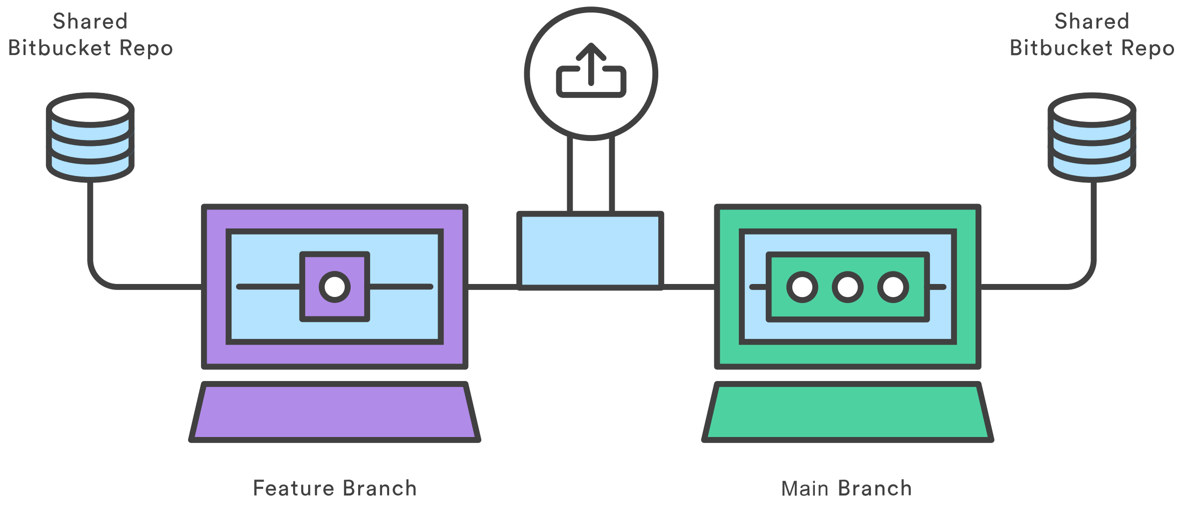
Pull requests can be used in conjunction with the Feature Branch Workflow, the Gitflow Workflow, or the Forking Workflow. But a pull request requires either two distinct branches or two distinct repositories, so they will not work with the Centralized Workflow. Using pull requests with each of these workflows is slightly different, but the general process is as follows:

1. A developer creates the feature in a dedicated branch in their local repo.
2. The developer pushes the branch to a public Bitbucket repository.
3. The developer files a pull request via Bitbucket.
4. The rest of the team reviews the code, discusses it, and alters it.
5. The project maintainer merges the feature into the official repository and closes the pull request.

The rest of this section describes how pull requests can be leveraged against different collaboration workflows.

### Feature Branch Workflow With Pull Requests

The Feature Branch Workflow uses a shared Bitbucket repository for managing collaboration, and developers create features in isolated branches. But, instead of immediately merging them into `main`, developers should open a pull request to initiate a discussion around the feature before it gets integrated into the main codebase.



Feature branch workflow

There is only one public repository in the Feature Branch Workflow, so the pull request's destination repository and the source repository will always be the same. Typically, the developer will specify their feature branch as the source branch and the `main` branch as the destination branch.

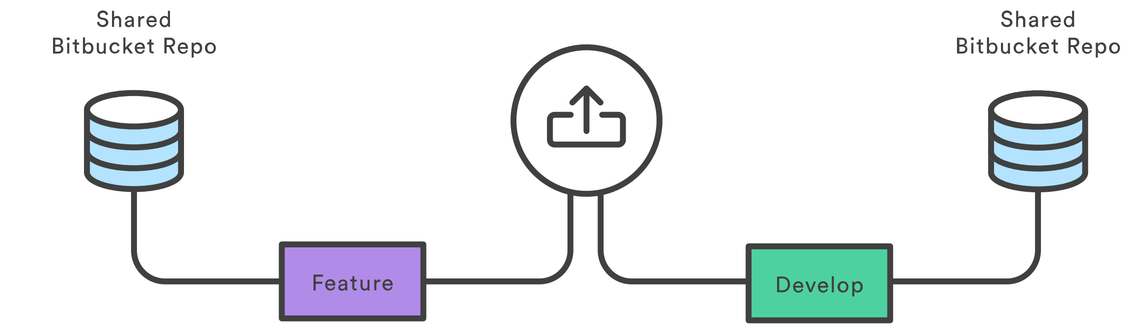
After receiving the pull request, the project maintainer has to decide what to do. If the feature is ready to go, they can simply merge it into `main` and close the pull request. But, if there are problems with the proposed changes, they can post feedback in the pull request. Follow-up commits will show up right next to the relevant comments.

It's also possible to file a pull request for a feature that is incomplete. For example, if a developer is having trouble implementing a particular requirement, they can file a pull request containing their work-in-progress. Other developers can then provide suggestions inside of the pull request, or even fix the problem themselves with additional commits.

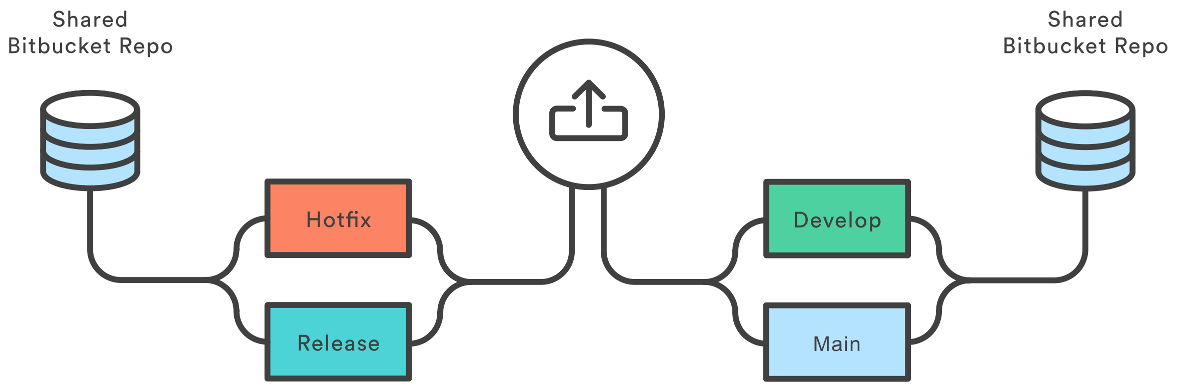
### Gitflow Workflow With Pull Requests

The Gitflow Workflow is similar to the Feature Branch Workflow, but defines a strict branching model designed around the project release. Adding pull requests to the

Gitflow Workflow gives developers a convenient place to talk about a release branch or a maintenance branch while they're working on it.



Gitflow workflow



Gitflow workflow

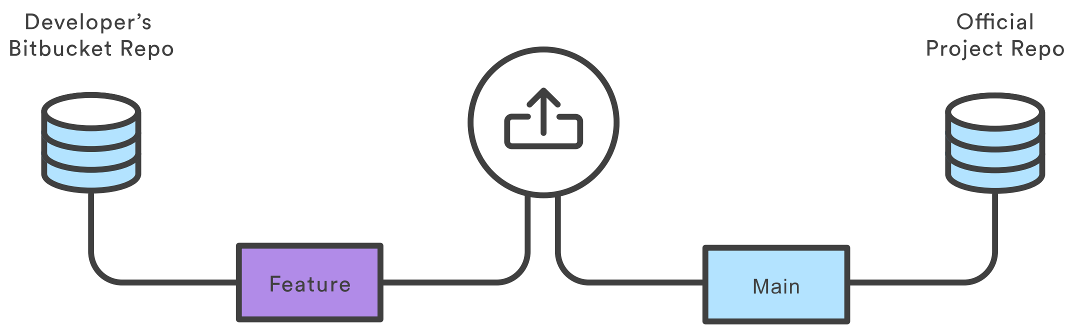
The mechanics of pull requests in the Gitflow Workflow are the exact same as the previous section: a developer simply files a pull request when a feature, release, or hotfix branch needs to be reviewed, and the rest of the team will be notified via Bitbucket.

Features are generally merged into the `develop` branch, while release and hotfix branches are merged into both `develop` and `main`. Pull requests can be used to formally manage all of these merges.

### Forking Workflow With Pull Requests

In the Forking Workflow, a developer pushes a completed feature to *their own* public repository instead of a shared one. After that, they file a pull request to let the project maintainer know that it's ready for review.

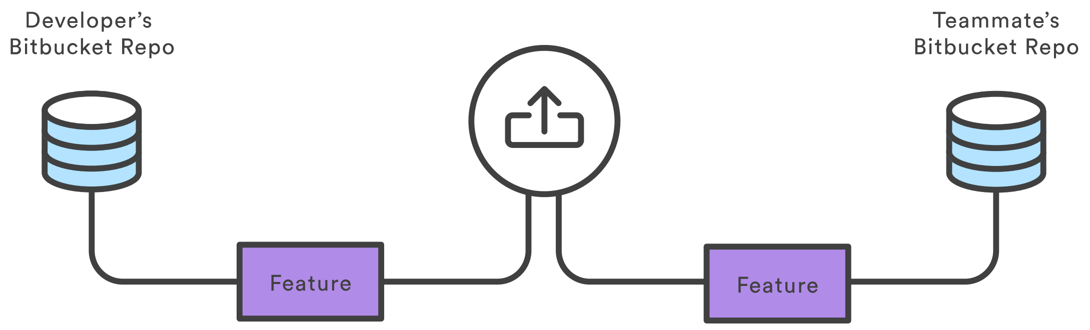
The notification aspect of pull requests is particularly useful in this workflow because the project maintainer has no way of knowing when another developer has added commits to their Bitbucket repository.



Forking workflow

Since each developer has their own public repository, the pull request's source repository will differ from its destination repository. The source repository is the developer's public repository and the source branch is the one that contains the proposed changes. If the developer is trying to merge the feature into the main codebase, then the destination repository is the official project and the destination branch is `main`.

Pull requests can also be used to collaborate with other developers outside of the official project. For example, if a developer was working on a feature with a teammate, they could file a pull request using the *teammate's* Bitbucket repository for the destination instead of the official project. They would then use the same feature branch for the source and destination branches.



Pull requests: Forking workflow

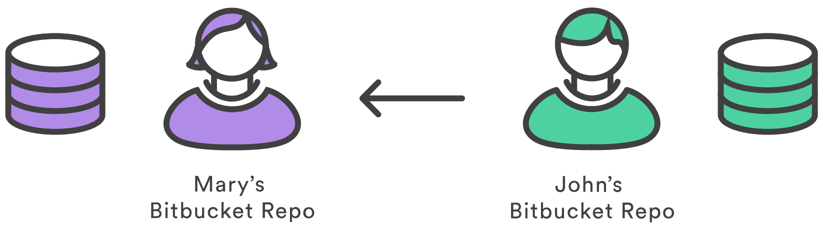
The two developers could discuss and develop the feature inside of the pull request. When they're done, one of them would file another pull request asking to merge the feature into the official main branch. This kind of flexibility makes pull requests very powerful collaboration tool in the Forking workflow.

**Example:**

The example below demonstrates how pull requests can be used in the Forking Workflow. It is equally applicable to developers working in small teams and to a third-party developer contributing to an open source project.

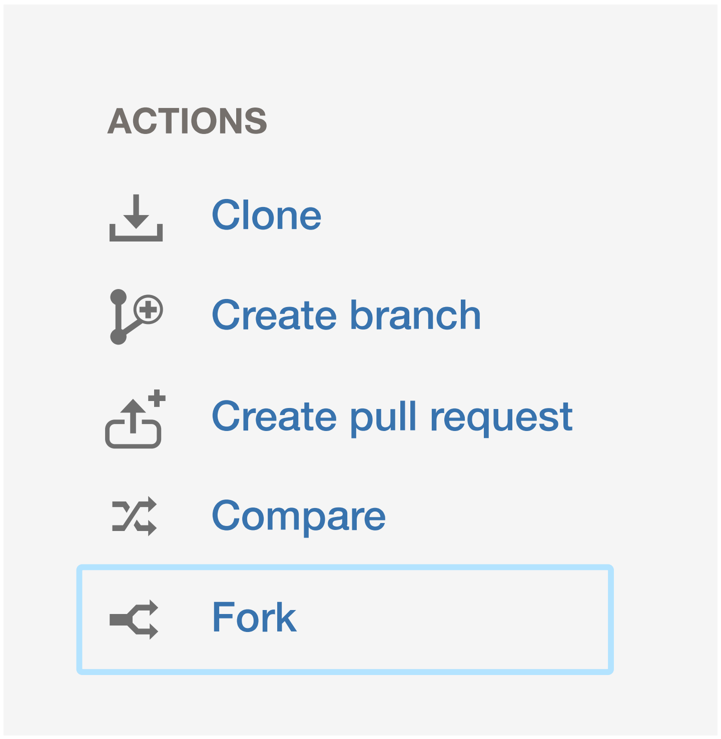
In the example, Mary is a developer, and John is the project maintainer. Both of them have their own public Bitbucket repositories, and John's contains the official project.

**Mary forks the official project**



Fork the project

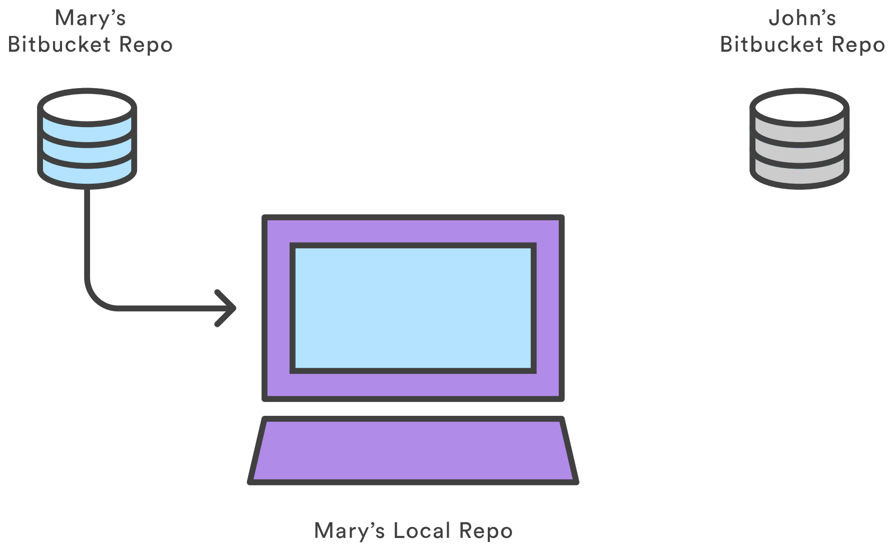
To start working in the project, Mary first needs to fork John's Bitbucket repository. She can do this by signing in to Bitbucket, navigating to John's repository, and clicking the *Fork* button.



Fork in bitbucket

After filling out the name and description for the forked repository, she will have a server-side copy of the project.

#### Mary clones her Bitbucket repository



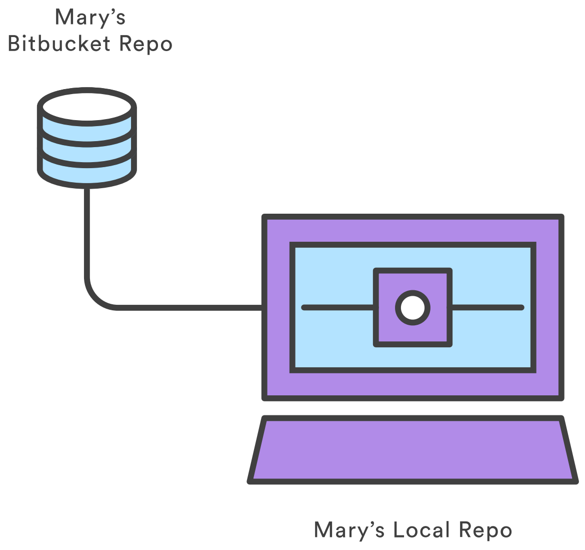
Clone the Bitbucket repo

Next, Mary needs to clone the Bitbucket repository that she just forked. This will give her a working copy of the project on her local machine. She can do this by running the following command:

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

Keep in mind that `git clone` automatically creates an `origin` remote that points back to Mary's forked repository.

#### Mary develops a new feature



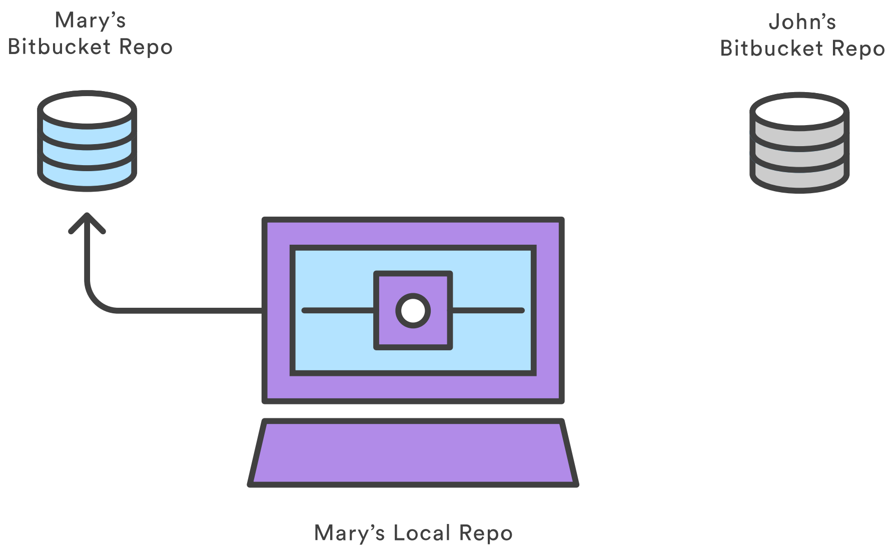
Develop a new feature

Before she starts writing any code, Mary needs to create a new branch for the feature. This branch is what she will use as the source branch of the pull request.

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

Mary can use as many commits as she needs to create the feature. And, if the feature's history is messier than she would like, she can use an interactive rebase to remove or squash unnecessary commits. For larger projects, cleaning up a feature's history makes it much easier for the project maintainer to see what's going on in the pull request.

**Mary pushes the feature to her Bitbucket repository**



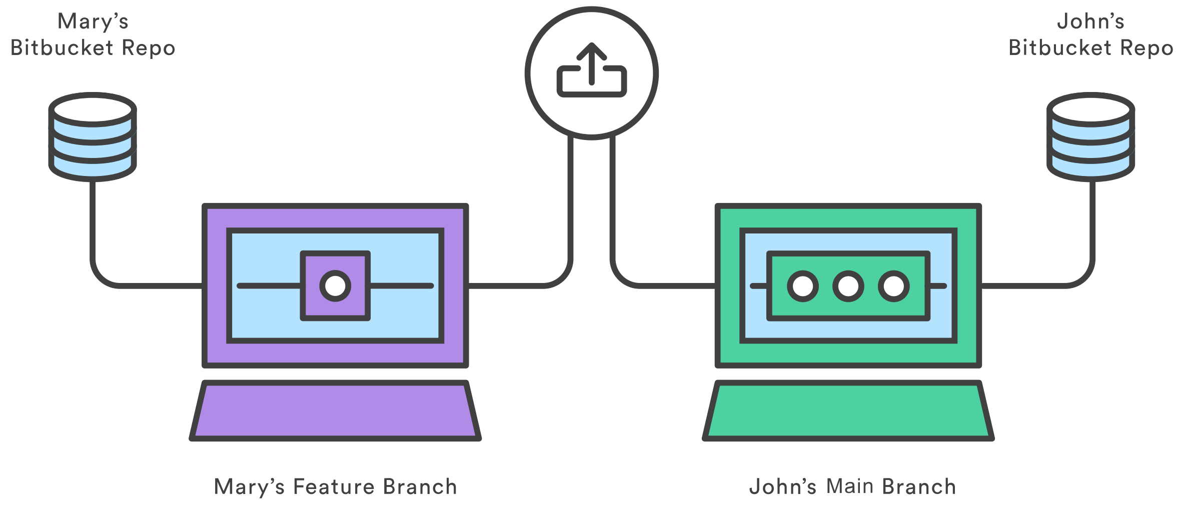
Push feature to Bitbucket repository

After her feature is complete, Mary pushes the feature branch to her own Bitbucket repository (not the official repository) with a simple `git push`:

```
git push origin some-branch
```

This makes her changes available to the project maintainer (or any collaborators who might need access to them).

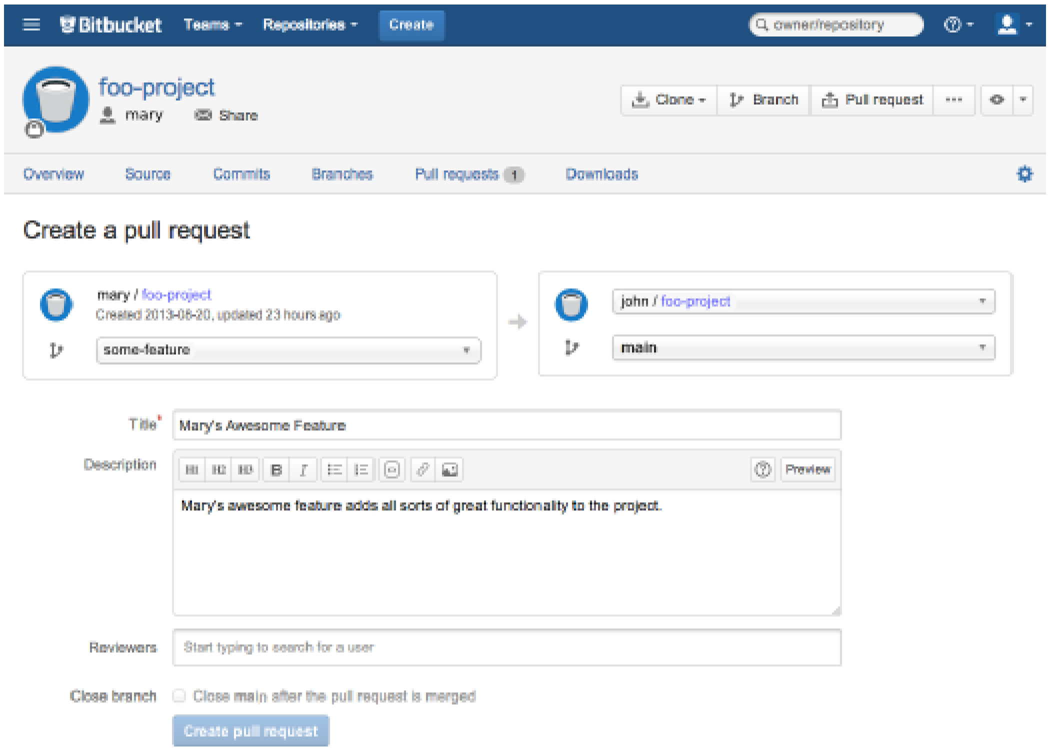
**Mary creates the pull request**



Create a pull request

After Bitbucket has her feature branch, Mary can create the pull request through her Bitbucket account by navigating to her forked repository and clicking the *Pull request* button in the top-right corner. The resulting form automatically sets Mary's repository as the source repository, and it asks her to specify the source branch, the destination repository, and the destination branch.

Mary wants to merge her feature into the main codebase, so the source branch is her feature branch, the destination repository is John's public repository, and the destination branch is `main`. She'll also need to provide a title and description for the pull request. If there are other people who need to approve the code besides John, she can enter them in the *Reviewers* field.



mary / foo-project
Created 2013-08-20, updated 23 hours ago
some-feature

john / foo-project
main

Title* Mary's Awesome Feature

Description: Mary's awesome feature adds all sorts of great functionality to the project.

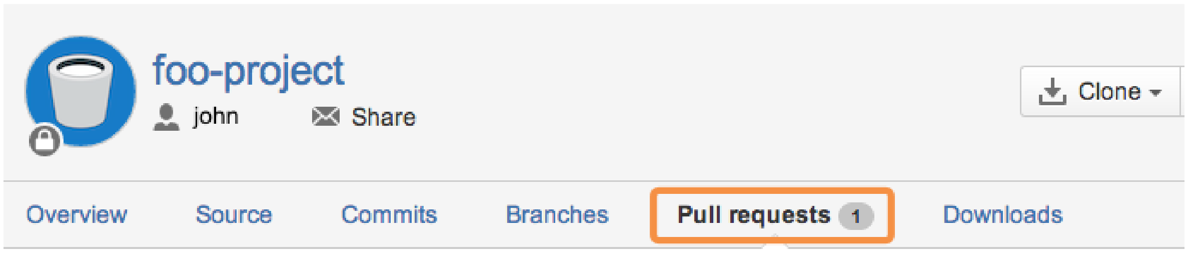
Reviewers: Start typing to search for a user

Close branch: ☐ Close main after the pull request is merged

Pull request within Bitbucket

After she creates the pull request, a notification will be sent to John via his Bitbucket feed and (optionally) via email.

**John reviews the pull request**

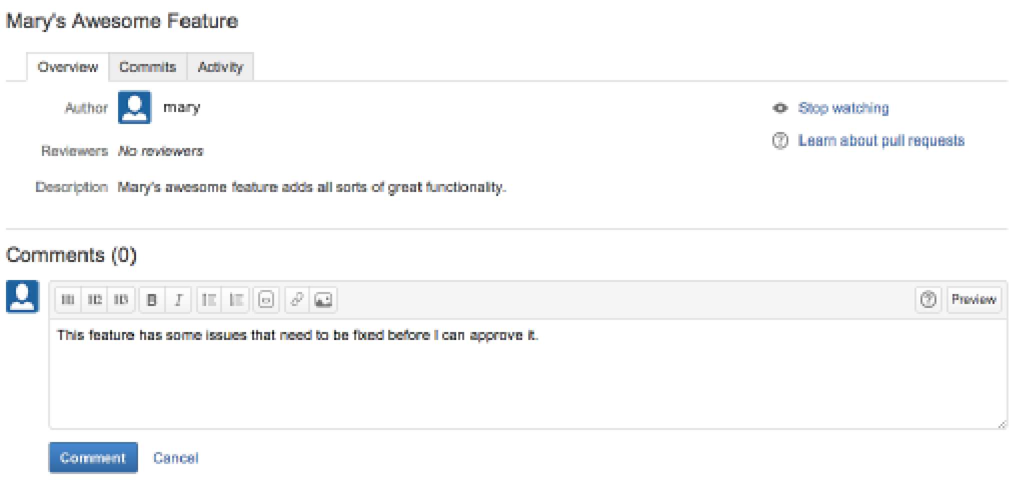


Bitbucket pull request

John can access all of the pull requests people have filed by clicking on the *Pull request* tab in his own Bitbucket repository. Clicking on Mary's pull request will show him a description of the pull request, the feature's commit history, and a diff of all the changes it contains.

If he thinks the feature is ready to merge into the project, all he has to do is hit the *Merge* button to approve the pull request and merge Mary's feature into his `main` branch.

But, for this example, let's say John found a small bug in Mary's code, and needs her to fix it before merging it in. He can either post a comment to the pull request as a whole, or he can select a specific commit in the feature's history to comment on.



Author: mary
Reviewers: No reviewers
Description: Mary's awesome feature adds all sorts of great functionality.

This feature has some issues that need to be fixed before I can approve it.

Bitbucket comment within pull request

### Mary adds a follow-up commit

If Mary has any questions about the feedback, she can respond inside of the pull request, treating it as a discussion forum for her feature.

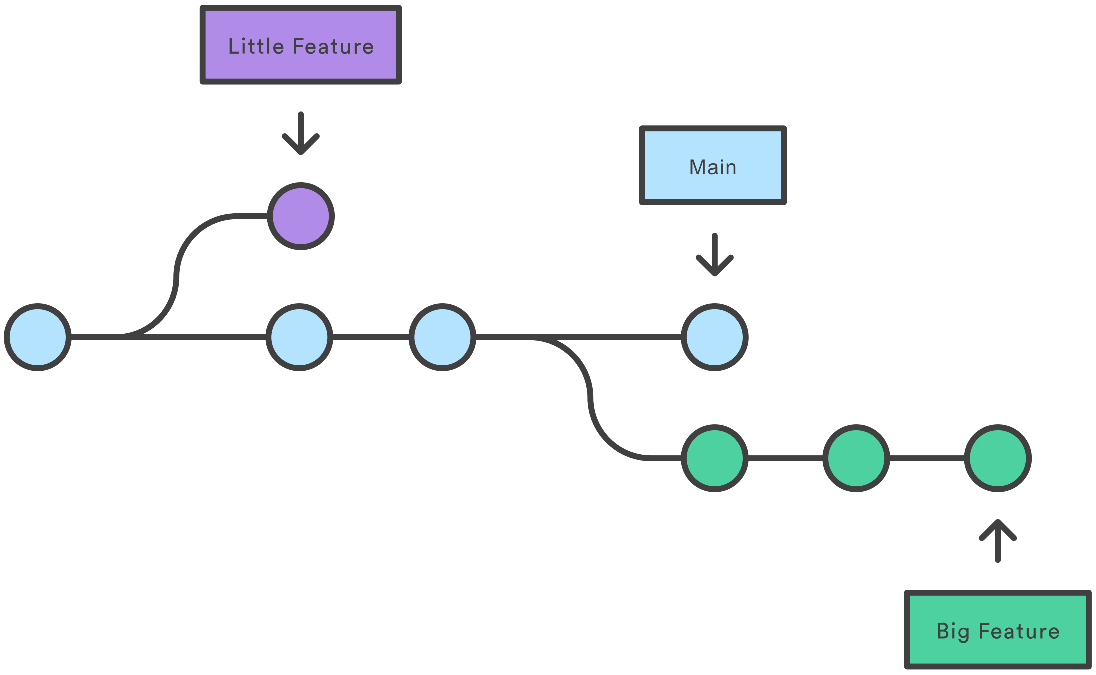
To correct the error, Mary adds another commit to her feature branch and pushes it to her Bitbucket repository, just like she did the first time around. This commit is automatically added to the original pull request, and John can review the changes again, right next to his original comment.

**John accepts the pull request**

Finally, John accepts the changes, merges the feature branch into main, and closes the pull request. The feature is now integrated into the project, and any other developers working on it can pull it into their own local repositories using the standard `git pull` command.

## Using branches

Git branches are effectively a pointer to a snapshot of your changes. 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 it harder for unstable code to get merged into the main code base, and it gives you the chance to clean up your future's history before merging it into the main branch.



Git branch

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 `main` branch free from questionable code.

- `git branch`: List all of the branches in your repository. This is synonymous with `git branch --list.`
- `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>`: Force delete the specified branch, even if it has unmerged changes. This is the command to use if you want to permanently throw away all of the commits associated with a particular line of development.
- `git branch -m <branch>`: Rename the current branch to `< branch >`
- `git branch -a`: List all remote branches.
- `git checkout <branch_name>` : Switches to the specified branch.
- `git checkout -b <branch_name>`: Creates and switches to a new branch in one step.
- **`git merge <branch_name>`**: Merges the specified branch into the current branch.
  **Example**:

```
git merge feature/add-header
```

## Creating remote branches

So far these examples have all demonstrated local branch operations. The `git branch` command also works on remote branches

```
git remote add new-remote-repo https://bitbucket.com/user/repo.git
# Add remote repo to local repo config
git push <new-remote-repo> crazy-experiment~
# pushes the crazy-experiment branch to new-remote-repo
```

## Merge Conflicts

Version control systems are all about managing contributions between multiple distributed authors ( usually developers ). Sometimes multiple developers may try to edit the same content. If Developer A tries to edit code that Developer B is editing a conflict may occur. To alleviate the occurrence of conflicts developers will work in separate isolated branches.

### Creating a merge conflict

In order to get real familiar with merge conflicts, the next section will simulate a conflict to later examine and resolve. The example will be using a Unix-like command-line Git interface to execute the example simulation.

```
$ mkdir git-merge-test
$ cd git-merge-test
$ git init .
$ echo "this is some content to mess with" > merge.txt
$ git add merge.txt
$ git commit -am"we are commiting the inital content"
[main (root-commit) d48e74c] we are commiting the inital content
1 file changed, 1 insertion(+)
create mode 100644 merge.txt
```

This code example executes a sequence of commands that accomplish the following.

- Create a new directory named `git-merge-test`, change to that directory, and initialize it as a new Git repo.
- Create a new text file `merge.txt` with some content in it.
- Add `merge.txt` to the repo and commit it.

Now we have a new repo with one branch `main` and a file `merge.txt` with content in it. Next, we will create a new branch to use as the conflicting merge.

```
$ git checkout -b new_branch_to_merge_later
$ echo "totally different content to merge later" > merge.txt
$ git commit -am"edited the content of merge.txt to cause a conflict"
```

```
[new_branch_to_merge_later 6282319] edited the content of merge.txt to
cause a conflict
1 file changed, 1 insertion(+), 1 deletion(-)
```

The proceeding command sequence achieves the following:

- create and check out a new branch named `new_branch_to_merge_later`
- overwrite the content in `merge.txt`
- commit the new content

With this new branch: `new_branch_to_merge_later` we have created a commit that overrides the content of `merge.txt`

```
git checkout main
Switched to branch 'main'
echo "content to append" >> merge.txt
git commit -am"appended content to merge.txt"
[main 24fbe3c] appended content to merge.tx
1 file changed, 1 insertion(+)
```

This chain of commands checks out the `main` branch, appends content to `merge.txt`, and commits it. This now puts our example repo in a state where we have 2 new commits. One in the `main` branch and one in the `new_branch_to_merge_later` branch. At this time lets `git merge new_branch_to_merge_later` and see what happen!

```
$ git merge new_branch_to_merge_later
Auto-merging merge.txt
CONFLICT (content): Merge conflict in merge.txt
Automatic merge failed; fix conflicts and then commit the result.
```

BOOM 💥. A conflict appears. Thanks, Git for letting us know about this!

## How to identify merge conflicts

As we have experienced from the proceeding example, Git will produce some descriptive output letting us know that a CONFLICT has occcured. We can gain further insight by running the git status command

```
$ git status
On branch main
You have unmerged paths.
(fix conflicts and run "git commit")
(use "git merge --abort" to abort the merge)

Unmerged paths:
(use "git add <file>..." to mark resolution)

both modified:   merge.txt
```

The output from `git status` indicates that there are unmerged paths due to a conflict. The `merge.text` file now appears in a modified state. Let's examine the file and see whats modified.

```
$ cat merge.txt
<<<<<<< HEAD
this is some content to mess with
content to append
=======
totally different content to merge later
>>>>>>> new_branch_to_merge_later
```

Here we have used the `cat` command to put out the contents of the `merge.txt` file. We can see some strange new additions

- `<<<<<<< HEAD`
- `=======`
- `>>>>>>> new_branch_to_merge_later`

Think of these new lines as "conflict dividers". The `=======` line is the "center" of the conflict. All the content between the center and the `<<<<<<< HEAD` line is content that exists in the current branch main which the `HEAD` ref is pointing to. Alternatively all content between the center and `>>>>>>> new_branch_to_merge_later` is content that is present in our merging branch.

## How to resolve merge conflicts using the command line

The most direct way to resolve a merge conflict is to edit the conflicted file. Open the `merge.txt` file in your favorite editor. For our example lets simply remove all the conflict dividers. The modified `merge.txt` content should then look like:

```
this is some content to mess with
content to append
totally different content to merge later
```

Once the file has been edited use `git add merge.txt` to stage the new merged content. To finalize the merge create a new commit by executing:

```
git commit -m "merged and resolved the conflict in merge.txt"
```

Git will see that the conflict has been resolved and creates a new merge commit to finalize the merge.

# Advanced Git Operations

Have you gotten accustomed to the basics of `git`, but the advanced concepts make you *scratch your head*?

hard-things

## 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 `main` branch. This encapsulation makes it easy for multiple developers to work on a particular feature without disturbing the main codebase. It also means the `main` branch will never contain broken code, which is a huge advantage for continuous integration environments.

### Start with the main branch

All feature branches are created off the latest code state of a project. This guide assumes this is maintained and updated in the `main` branch.

```
git checkout main
git fetch origin
git reset --hard origin/main
```

### Create the repository

This switches the repo to the `main` branch, pulls the latest commits and resets the repo's local copy of `main` to match the latest version.

### Create a new-branch

Use a separate branch for each feature or issue you work on. After creating a branch, check it out locally so that any changes you make will be on that branch.

```
git checkout -b new-feature
```

This checks out a branch called new-feature based on `main`, and the -b flag tells Git to create the branch if it doesn't already exist.

### Update, add, commit, and push changes

On this branch, edit, stage, and commit changes in the usual fashion, building up the feature with as many commits as necessary. Work on the feature and make commits like you would any time you use Git. When ready, push your commits, updating the feature branch on Bitbucket.

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

### Push feature branch to remote

It's a good idea to push the feature branch up to the central repository. This serves as a convenient backup, when collaborating with other developers, this would give them access to view commits to the new branch.

```
git push -u origin new-feature
```

This command pushes new-feature to the central repository (origin), and the -u flag adds it as a remote tracking branch. After setting up the tracking branch, `git push` can be invoked without any parameters to automatically push the new-feature branch to the central repository. To get feedback on the new feature branch, create a pull request in a repository management solution like bitbucket cloud or GitHub. From there, you can add reviewers and make sure everything is good to go before merging.

### Resolve feedback

Now teammates comment and approve the pushed commits. Resolve their comments locally, commit, and push the suggested changes to Bitbucket. Your updates appear in the pull request.

### Merge your pull request

Before you merge, you may have to resolve merge conflicts if others have made changes to the repo. When your pull request is approved and conflict-free, you can add your code to the `main` branch. Merge from the pull request in Bitbucket.

### Pull requests

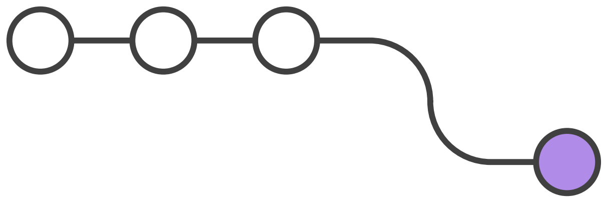
Once a pull request is accepted, the actual act of publishing a feature is much the same as in the Centralized Workflow. First, you need to make sure your local `main` is synchronized with the upstream `main`. Then, you merge the feature branch into `main` and push the updated `main` back to the central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

Pull requests can be facilitated by source code management solutions like Bitbucket Cloud.

## Example feature branching workflow

The following is an example of the type of scenario in which a feature branching workflow is used. The scenario is that of a team doing code review around on a new feature pull request. This is one example of the many purposes this model can be used for.

### Mary begins a new feature



Feature branch illustration

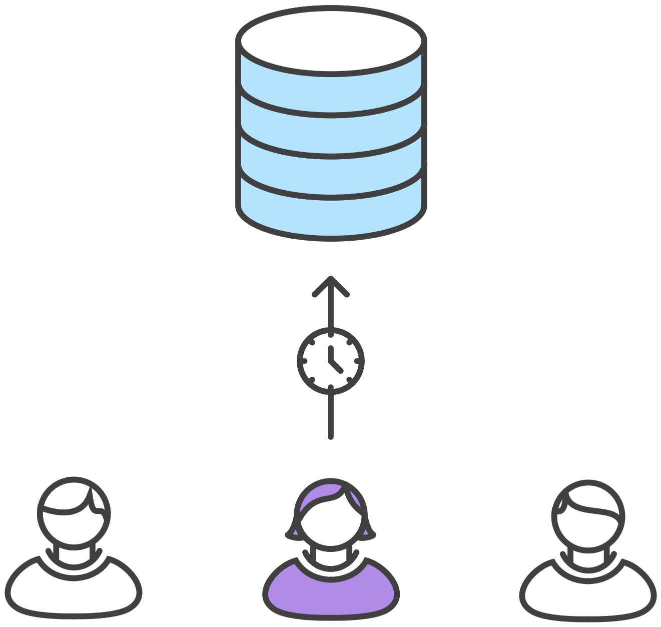
Before she starts developing a feature, Mary needs an isolated branch to work on. She can request a new branch with the following command:

```
git checkout -b marys-feature main
```

This checks out a branch called `marys-feature` based on `main`, 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



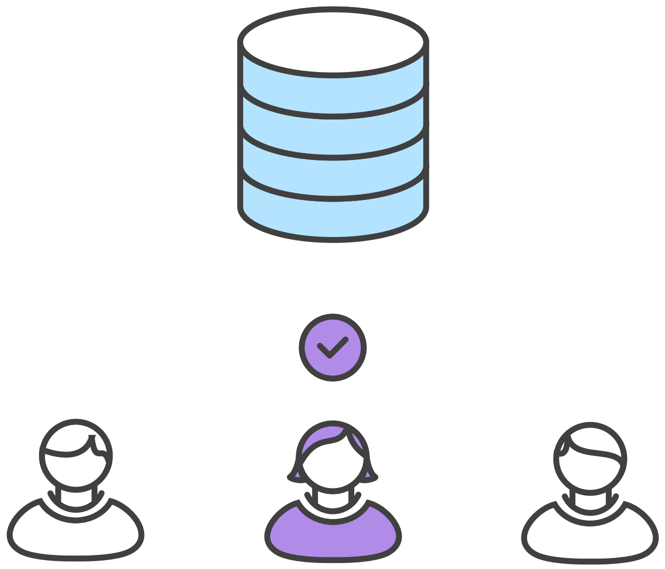
Push to central repository

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



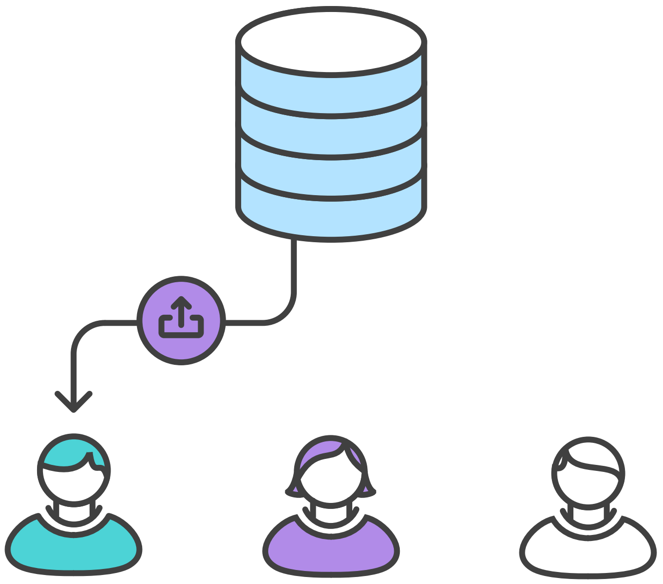
Pull request

When Mary gets back from lunch, she completes her feature. Before merging it into `main`, 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 `main`, 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.

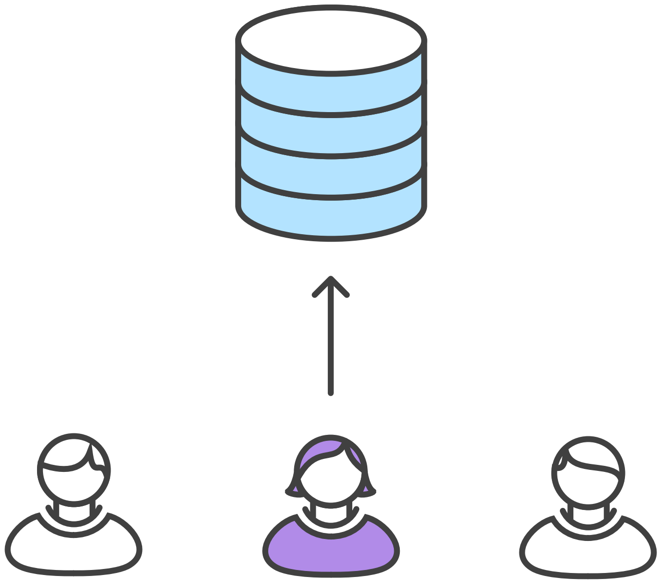
## Bill receives the pull request



Review pull request illustration

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

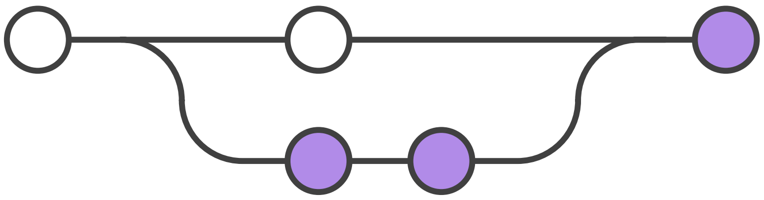


Pull request revisions

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


Publishing 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 main
git pull
git pull origin marys-feature
git push
```

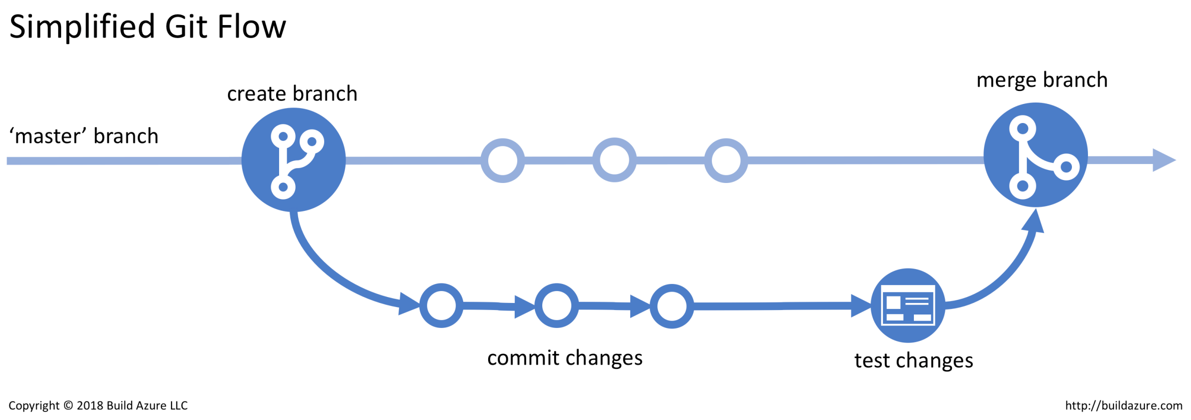
This process often results in a merge commit. Some developers like this because it's like a symbolic joining of the feature with the rest of the code base. But, if you're partial to a linear history, it's possible to rebase the feature onto the tip of `main` before executing the merge, resulting in a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

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.

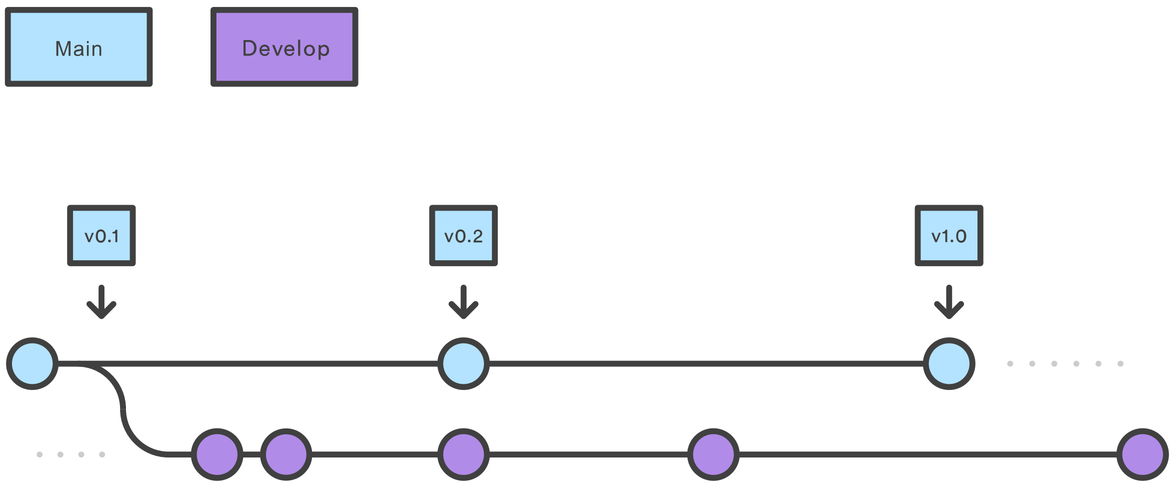
## Gitflow workflow



1746388779271

Gitflow is an alternative Git branching model that involves the use of feature branches and multiple primary branches. It was first published and made popular by Vincent Driessen at nvie. Compared to trunk-based development, Gitflow has numerous, longer-lived branches and larger commits

### How it works



Git workflow

### Develop and main branches

Instead of a single main branch, this workflow uses two branches to record the history of the project. The main 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 main branch with a version number.

The first step is to complement the default main 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 `main` will contain an abridged version. Other developers should now clone the central repository and create a tracking branch for `develop.`

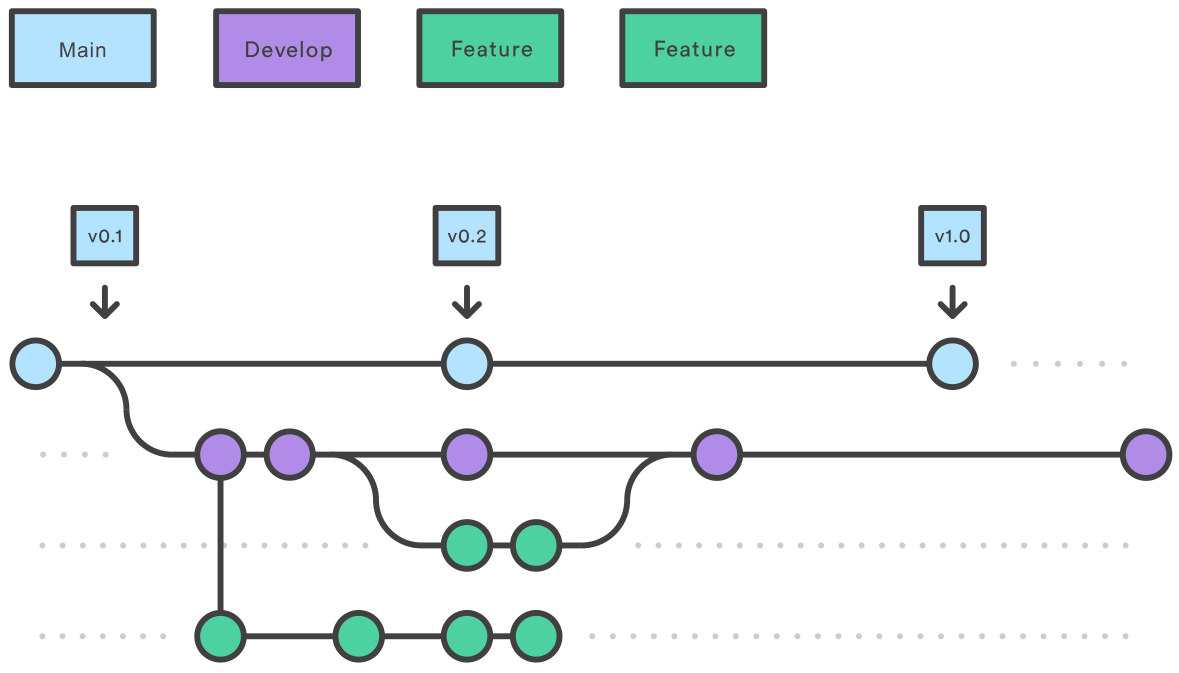
When using the git-flow extension library, executing `git flow init` on an existing repo will create the `develop` branch:

```
$ git flow init


Initialized empty Git repository in ~/project/.git/
No branches exist yet. Base branches must be created now.
Branch name for production releases: [main]
Branch name for "next release" development: [develop]


How to name your supporting branch prefixes?
Feature branches? [feature/]
Release branches? [release/]
Hotfix branches? [hotfix/]
Support branches? [support/]
Version tag prefix? []


$ git branch
* develop
 main
```

## Feature branches

### Step 1. Create the repository

Each new feature should reside in its own branch, which can be pushed to the central repository (https://www.atlassian.com/git/tutorials/syncing/git-push) for backup/collaboration. But, instead of branching off of main, feature branches use develop as their parent branch. When a feature is complete, it gets merged back into develop (https://www.atlassian.com/git/tutorials/using-branches/git-merge). Features should never interact directly with main.



Git workflow - feature branches

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.

Feature branches are generally created off to the latest develop branch.

**Creating a feature branch**

Without the git-flow extensions:

```
git checkout develop
git checkout -b feature_branch
```

When using the git-flow extension:

```
git flow feature start feature_branch
```

Continue your work and use Git like you normally would.

### Finishing a feature branch

When you're done with the development work on the feature, the next step is to merge the feature_branch into develop.

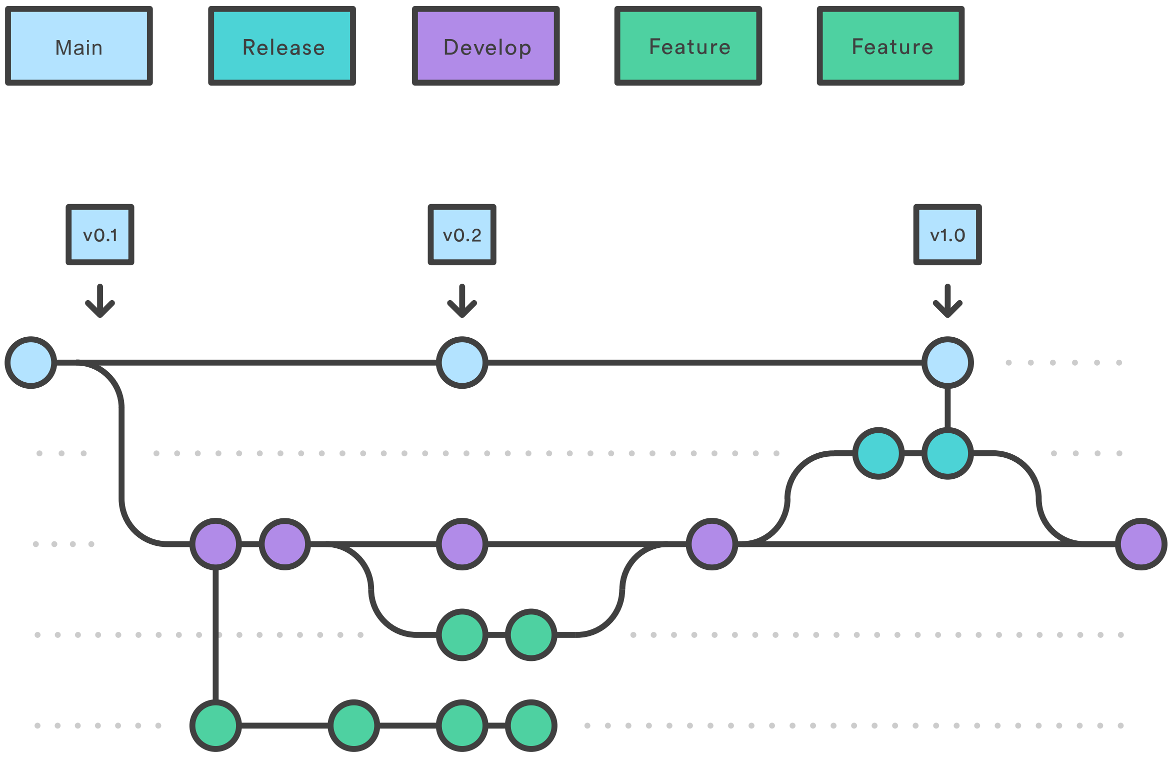
Without the git-flow extensions:

```
git checkout develop
git merge feature_branch
```

Using the git-flow extensions:

```
git flow feature finish feature_branch
```

## Release branches



Git workflow - 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 branch gets merged into main 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).

Making `release` branches is another straightforward branching operation. Like `feature` branches, `release` branches are based on the `develop` branch. A new `release` branch can be created using the following methods.

Without the git-flow extensions:

```
git checkout develop
git checkout -b release/0.1.0
```

When using the git-flow extensions:

```
$ git flow release start 0.1.0
Switched to a new branch 'release/0.1.0'
```

Once the release is ready to ship, it will get merged it into `main` and `develop`, then the `release` branch will be deleted. 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. If your organization stresses code review, this would be an ideal place for a pull request.

To finish a `release` branch, use the following methods:

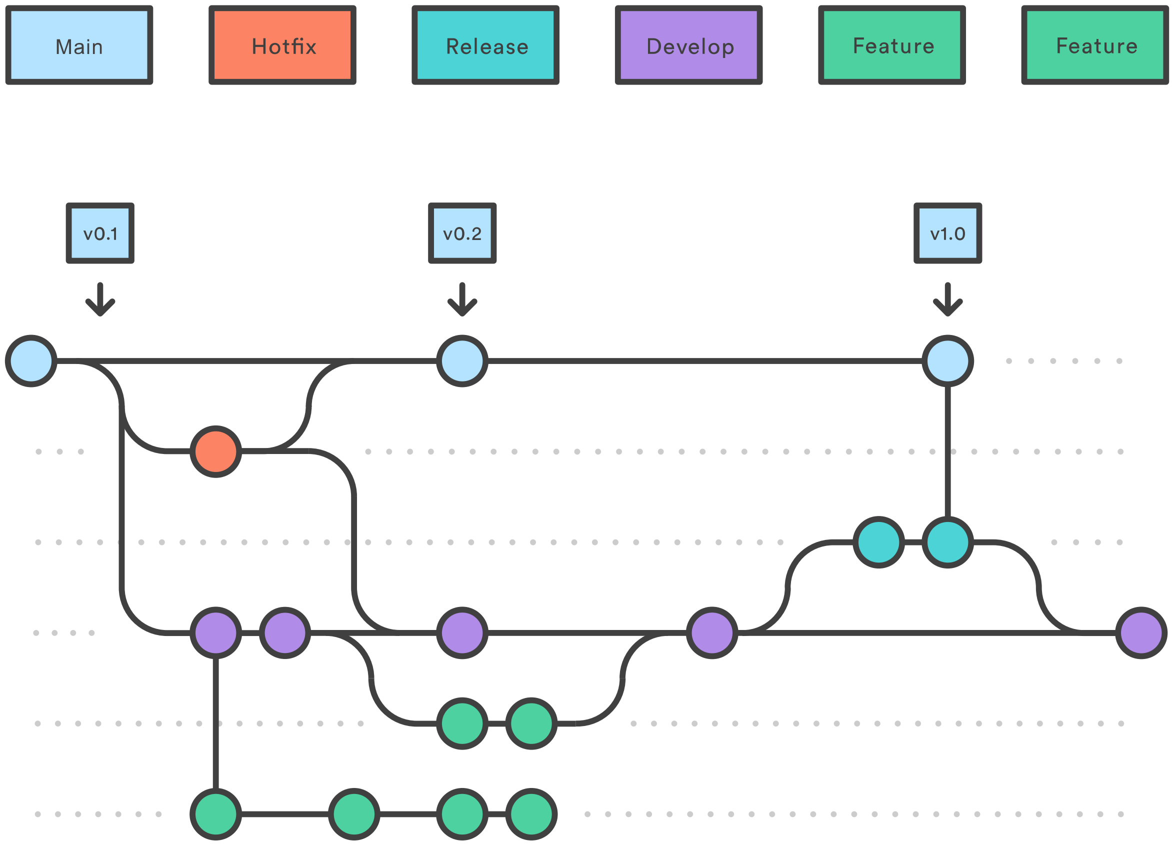
Without the git-flow extensions:

```
git checkout main
git merge release/0.1.0
```

Or with the git-flow extension:

```
git flow release finish '0.1.0'
```

## Hotfix branches



Hotfix branch within git workflow

Maintenance or "hotfix" branches are used to quickly patch production releases. Hotfix branches are a lot like release branches and feature branches except they're based on main instead of develop. This is the only branch that should fork directly off of main. As soon as the fix is complete, it should be merged into both main and develop (or the current release branch), and main 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 main. A hotfix branch can be created using the following methods:

Without the git-flow extensions:

```
git checkout main
git checkout -b hotfix_branch
```

When using the git-flow extensions:

```
$ git flow hotfix start hotfix_branch
```

Similar to finishing a `release` branch, a `hotfix` branch gets merged into both `main` and `develop.`

```
git checkout main
git merge hotfix_branch
git checkout develop
git merge hotfix_branch
git branch -D hotfix_branch
```

```
$ git flow hotfix finish hotfix_branch
```

## Forking workflow

The Forking Workflow is fundamentally different than other popular Git workflows. Instead of using a single server-side repository to act as the “central” codebase, it gives every developer their own 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 Forking Workflow is most often seen in public open source projects.

### How it works

1. A developer 'forks' an 'official' server-side repository. This creates their own server-side copy.
2. The new server-side copy is cloned to their local system.
3. A Git remote path for the 'official' repository is added to the local clone.
4. A new local feature branch is created.
5. The developer makes changes on the new branch.
6. New commits are created for the changes.
7. The branch gets pushed to the developer's own server-side copy.

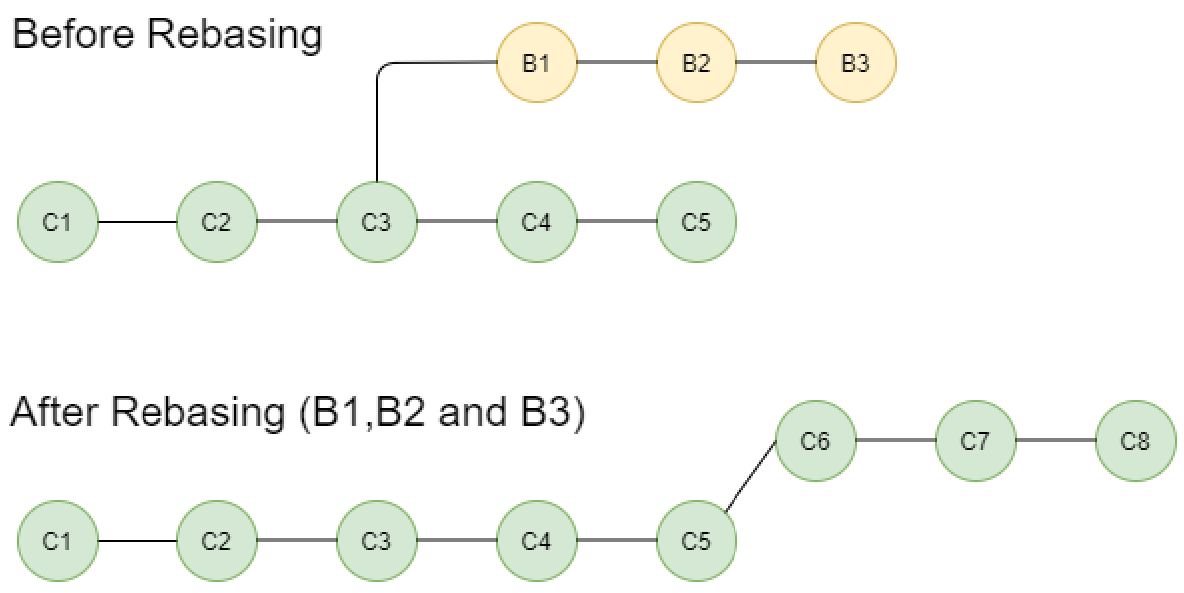
8. The developer opens a pull request from the new branch to the 'official' repository.
9. The pull request gets approved for merge and is merged into the original server-side repository

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 their local `main` branch, then pushes the `main` 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.

### Forking vs cloning

It's important to note that "forked" repositories and "forking" are not special operations. Forked repositories are created using the standard git clone command. Forked repositories are generally "server-side clones" and usually managed and hosted by a 3rd party Git service like Bitbucket. There is no unique Git command to create forked repositories. A clone operation is essentially a copy of a repository and its history.

## Git Rebasing



1746390511127

Git Rebase is a command used to move or combine a sequence of commits to a new base commit

## Types of Git Rebase

### 1. Interactive Rebase (git rebase -i)

- This allows you to edit, squash, reorder, or delete commits in your branch. It gives you full control over the commit history, making it useful for cleaning up commit messages or combining multiple commits into one.
- It squashing commits to combine them into a single commit.
- Git rebase reordering commits to reflect a more logical flow.
- Editing commit messages before pushing them to a remote repository

**Syntax:**

```
git checkout branch_x
git rebase -i master
```

**In this syntax**:

- This command lists all commits that are about to be moved and prompts you to edit or rearrange them based on your choices. It helps maintain a clean and structured project history.

### 2. Non-Interactive Rebase (Standard Rebase)

- This is the regular rebase command (git rebase `<branch>`), which simply applies your commits onto the target branch without allowing for manual intervention. It's ideal for straightforward rebasing where you don't need to modify or review individual commits.
- Updating your feature branch with the latest changes from the main branch.

**Syntax:**

```
git checkout <feature-branch>
git rebase <base-branch>
```

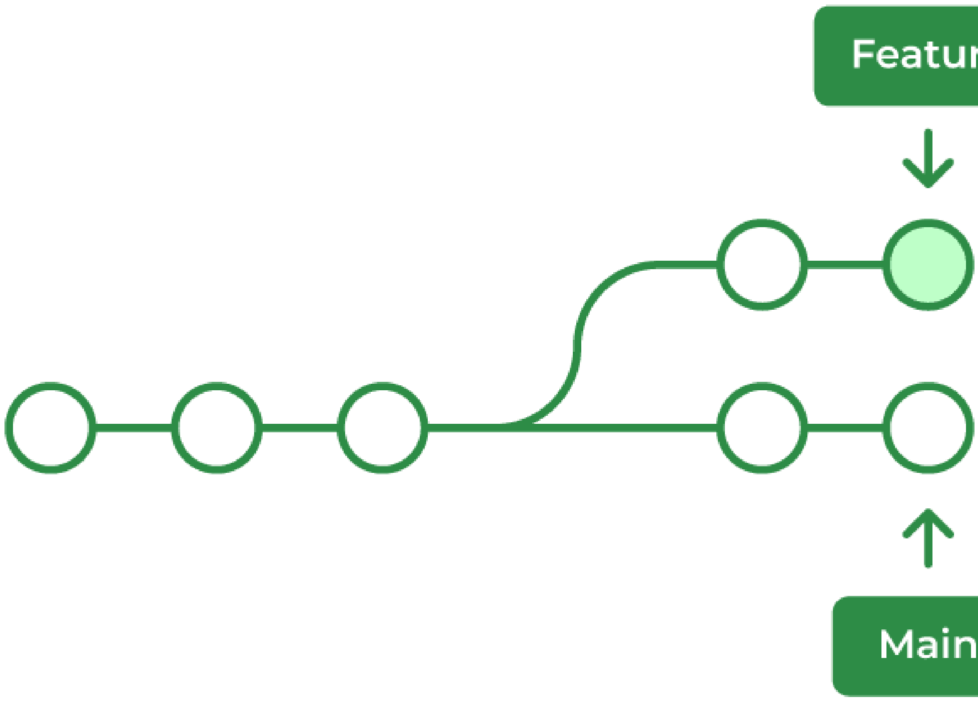
**In this syntax:**

- `<feature-branch>` is the branch with the changes you want to rebase.
- `<base-branch>` is the branch you want to rebase your changes onto, typically main or master.

## When to Use Git Rebase

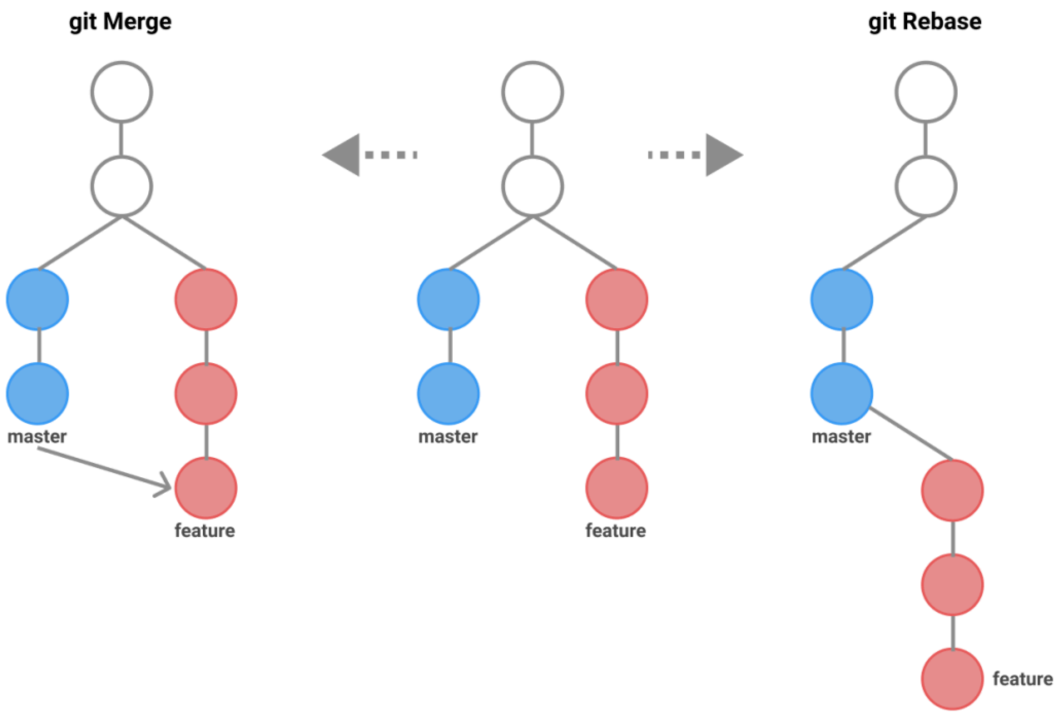
You can use git rebase in the following situations:

- **Clean up commit history:** If you've made multiple small commits or fixes that you want to combine into one commit for a cleaner history.
- **Stay up-to-date with the base branch**: If you're working on a feature branch and want to incorporate changes from the master branch into your branch without creating merge commits.
- **Prepare a branch for merging:** Before merging a feature branch into the master branch, you can use rebase to make sure your branch's changes are applied on top of the latest master branch.



1746390799863

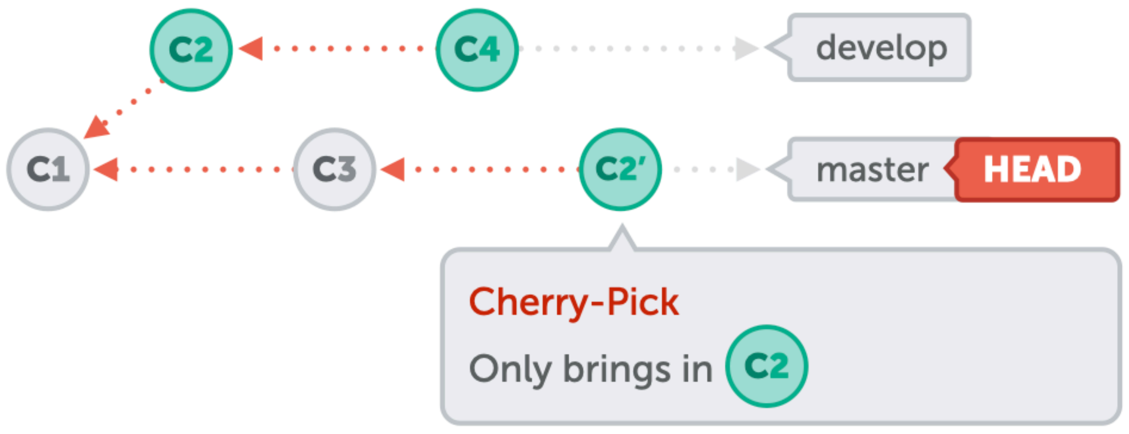
## Merging vs Rebasing



1746391592116

| Git Rebase | Git Merge |
|---|---|
| Rewrites commit history, leading to a cle aner, linear history. | Keeps commit history as is, leading to a m ore complex, branching history. |
| No merge commit is created, making the history easier to follow. | A merge commit is created, which can clu tter the history. |
| Can be used to update a feature branch with the latest changes from the base br anch. | Often used to integrate feature branches i nto the main branch. |
| Best for cleaning up commit history befo re merging. | Best for preserving history and when you want to maintain the exact sequence of e vents. |

## Git Cherry pick

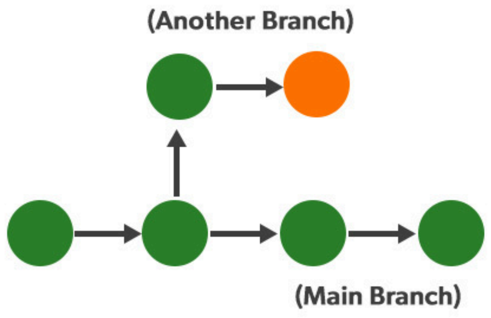


1746391404822

git **cherry-pick** in git means choosing a commit from one branch and applying it to another branch. This is in contrast with other ways such as **merge** and **rebases** which normally apply many commits into another branch.

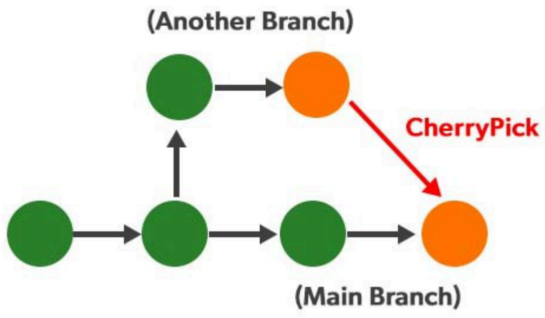
git cherry-pick is just like Rebasing an advanced concept and also a powerful command. It is mainly used if you don't want to merge the whole branch and you want some of the commits.

- Before cherry pick



1746389939141

- After Cherry pick



1746389948319

The command for Cherry-pick is as follows:

```
git cherry-pick<commit-hash>
```

## Some important Use Cases of Cherry-pick

The following are some common applications of Cherry-Pick:

1. If you by mistake make a commit in an incorrect branch, then using cherry-pick you can apply the required changes.
2. Suppose when the same data structure is to be used by both the frontend and backend of a project. Then a developer can use cherry-pick to pick the commit and use it to his/her part of the project.
3. At the point when a bug is found it is critical to convey a fix to end clients as fast as could be expected.
4. Some of the time a component branch might go old and not get converged into the main branch and the request might get closed, but since git never loses those commits, it can be cherry-picked and it would be back.

## How to use cherry-pick?

To demonstrate how to use `git cherry-pick` let us assume we have a repository with the following branch state:

```
a - b - c - d   Main
     \
       e - f - g Feature
```

`git cherry-pick` usage is straight forward and can be executed like:

```
git cherry-pick commitSha
```

In this example `commit` Sha is a commit reference. You can find a commit reference by using `git log`. In this example we have constructed lets say we wanted to use commit `f` in `main`. First we ensure that we are working on the `main` branch.

```
git checkout main
```

Then we execute the cherry-pick with the following command:

```
git cherry-pick f
```

Once executed our Git history will look like:

```
a - b - c - d - f   Main
     \
       e - f - g Feature
```

The f commit has been successfully picked into the main branch

Prepared with thanks by Kevin Comba