

Imagine you're working on a collaborative project, such as developing a task management app with your team. Each team member is responsible for a different feature, and you want to work on your part without interfering with your teammates' progress. This is where Git branching comes into play.

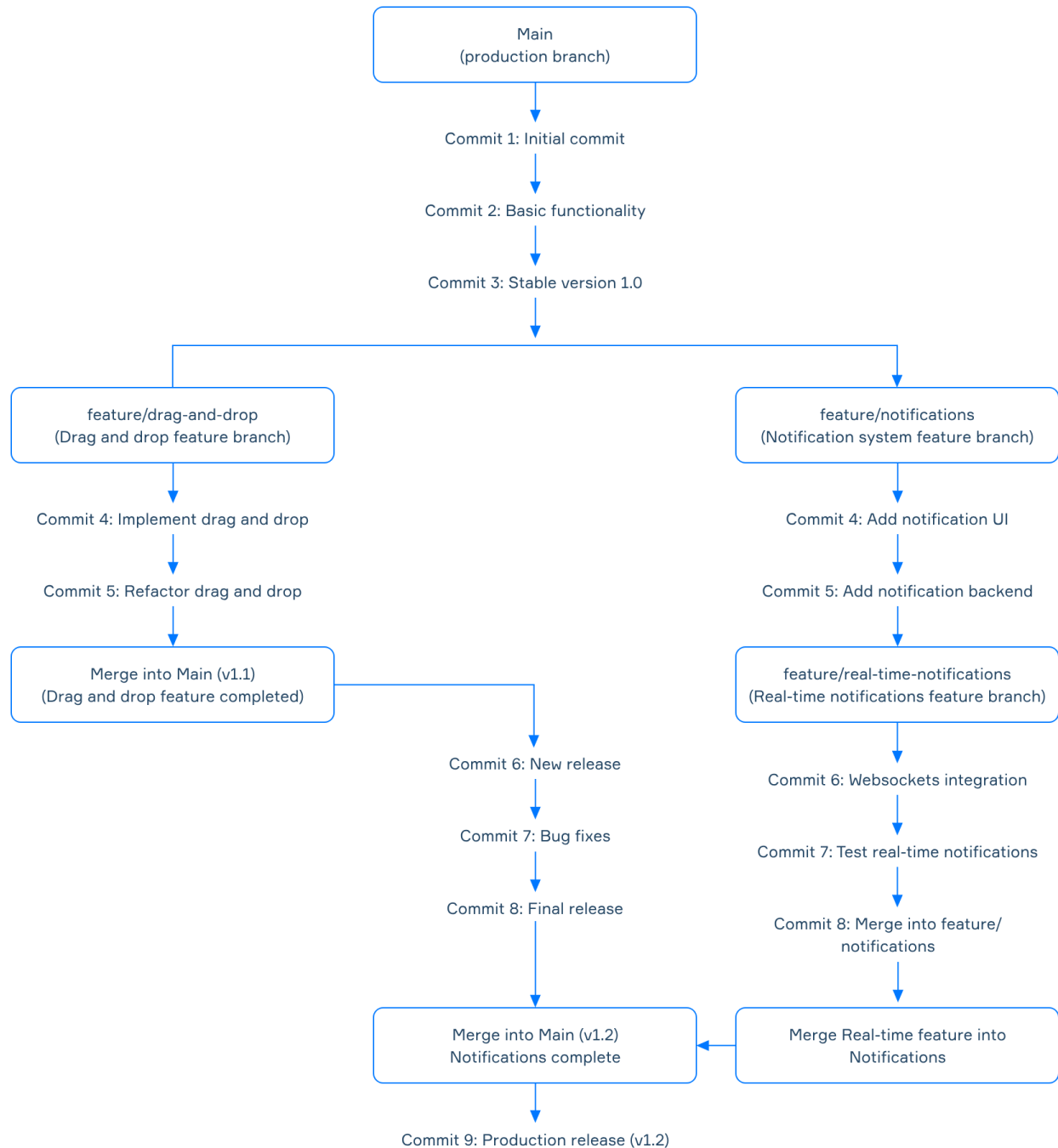
In this topic, we'll explore how branching in Git helps manage parallel workstreams, ensuring that your project remains stable while allowing for continuous development. We'll cover how to create and switch between branches, merge changes, stash changes, and resolve conflicts. By the end, you'll see how branching makes collaboration smoother and more efficient, keeping your project organized and your team productive.

Branching

Imagine you're working on a team project, like building a task management app called "TaskFlow." Each developer on the team is responsible for a different feature:

- Bob is adding drag-and-drop functionality.
- Alice is creating a notification system.
- Mike is implementing real-time updates.

They create separate branches for their features to avoid interfering with each other's work. This is where branching in Git becomes essential. The diagram below shows a visual representation of this:



A branch in Git is a way to work on a separate version of your project. It starts from the latest version of your main code, and as you make changes, the new branch keeps track of them. This allows you to develop modules separately, create alternative versions, or make fixes to your project without affecting the main codebase.

Unlike a literal copy of the project, a branch doesn't duplicate all the files; instead, it references the state of the project at a specific point in time. This is the latest commit on the branch. As you make new commits on the new branch, it will diverge from the main branch, creating its own separate history. Branches have their own commit history and changes isolated from each other until you decide to merge them.

There are several reasons this is crucial:

- The already working, stable version of the code is saved.
- Various new functions can be developed in parallel by different programmers.
- Developers can work with their own branches without the risk that the codebase will change due to someone else's changes.
- In case of doubt, you can develop different implementations of the same idea in different branches and then compare them.

In other words, branching is the ability to split versions of one project and work with them separately, allowing for flexibility and parallel development.

There are various branch management strategies depending on the team's workflow:

- Git flow: A structured model with separate branches for features, releases, and hotfixes.
- GitHub flow: A simpler model where all work is done in feature branches and merged into the main branch.
- GitLab flow: Combines Git Flow and GitHub Flow, with additional support for environment-specific branches (e.g., staging, production) to manage deployments.
- Trunk-based development: Developers frequently commit directly to the main branch, using short-lived feature branches.

Creating a branch

The main branch in each repository is typically called the

`main` branch. To create another branch, use the `git branch <name>` command:

```
$ git branch new_branch
```

This will create a new branch, which is, for now, a pointer to the same commit as the `main` branch. You can then switch to it, make changes, and get a new version of the code without affecting the `main`

branch. In case of errors or failures, you can switch back to the `main` branch and create another branch.

There is no limitation on the number of branches you can have.

When creating a branch, it's essential to use clear and descriptive names. This helps in identifying the purpose of the branch and makes collaboration easier. For example:

- Feature branches:
 - `feature/drag-and-drop`
- Bug fixes:
 - `bugfix/fix-drag-drop`
- Hotfixes:
 - `hotfix/urgent-fix`

Naming branches according to their purpose helps maintain a clean and organized repository, especially in larger projects.

After cloning a repository, you can list all branches, including remote ones, using

`git branch` command. To also view remote ones, use the

`-a` flag:

```
$ git branch -a
  feature/drag-and-drop
* main
remotes/origin/HEAD -> origin/main
remotes/origin/main
```

In this output, `main` is the currently active branch, indicated by the asterisk (*). To create a local copy of a remote branch, you can check it out:

```
$ git checkout -b local_branch_name origin/remote_branch_name
```

We'll discuss the `git checkout` command in detail in the next section. To keep your local repository up to date with the remote repository, you can use:

- `git fetch`
- : downloads changes from the remote repository but does not apply them to your local branches.
- `git pull`
- : fetches changes from the remote repository and merges them into your current branch.

To delete branches, you can use the

`-d` flag: `git branch -d new_branch`.

To force delete a branch, even if it hasn't been merged, use the `-D` flag. This can lead to data loss, so use it with caution.

Switching between branches

Now that you have several branches, you need to switch between different them to make changes to your files. There are two commands that allow you to do this:

- `git checkout`;
- `git switch`.

Traditionally, the `git checkout`

command has been used to switch between branches. To switch to `another_branch`, you would run:

```
$ git checkout another_branch
```

This command changes the active branch to `another_branch`, allowing you to work on its contents. Additionally, `git checkout` has a useful flag, `-b`

, which allows you to create a new branch and switch to it simultaneously:

```
$ git checkout -b new_branch
```

```
Switched to branch 'another_branch'
```

The `git switch` command is more focused and only deals with changing branches, unlike `git checkout`, which has multiple functions (e.g., switching branches, checking out files, etc.). To switch to `another_branch` using `git switch`, you would run:

```
$ git switch another_branch
```

```
Switched to branch 'another_branch'
```

`git switch` also has the `-c`

flag, which is similar to `git checkout -b`. It allows you to create a new branch and switch to it in one step:

```
$ git switch -c new_branch
```

```
Switched to branch 'new_branch'
```

Here are some differences between `git checkout` and `git switch`:

- Purpose: `git checkout` is a more general-purpose command that can switch branches, check out files, and more.
- `git switch`, on the other hand, is specifically designed for switching branches, making it a more intuitive option for this task. Syntax:
- `git switch`
- has a simpler and more focused syntax, which can be easier to understand and use for branch management.

In summary, while `git checkout` is still widely used and versatile, `git switch` is a more modern and straightforward option for branch-switching tasks. Depending on your Git version and personal preference, you can choose either command to manage your branches effectively.

Merging branches

So, you've switched to a new branch; now, you can start working on it. For example, Bob can add and commit the drag-and-drop functionality for the task board:

```
$ git add drag_and_drop.js
$ git commit -m "Added drag-and-drop functionality"
[feature/drag-and-drop 4a16dac] Added drag-and-drop
functionality

1 file changed, 1 insertion(+)
```

Since the changes are completed, Bob can switch back to the `main` branch:

```
$ git switch main
```

```
Switched to branch 'main'
```

Now, if we open our project in the file manager, we will not see the

`drag_and_drop.js` file because we switched back to the `main` branch where such a file does not exist. To put the files from a new version into the `main` branch, we use the `git merge` command. Merging just means applying changes from the new branch to the `main` version of the project.

```
$ git merge new_branch
```

Already up to date.

When you merge branches in Git, you're combining the changes from one branch into another. Depending on the state of the branches, Git will handle the merge in one of two primary ways: a fast-forward merge or a merge with a new commit.

A fast-forward merge happens when the branch you're merging into has not diverged from the branch you're merging. In this case, Git simply moves the branch pointer forward to the latest commit on the branch being merged into. This type of merge is straightforward and doesn't create a new commit.

```
$ git merge feature_branch
```

If the `main` branch has not diverged from `feature_branch`, Git will move the `main` branch pointer forward to include the commits from `feature_branch`. The history remains linear, with no additional commits created.

If the branches have diverged—meaning there are commits on both branches that the other branch doesn't have—Git will perform a merge that will result in a new commit. This new commit represents the combination of changes from both branches and ties their histories together. Using the `--no-ff` flag forces this behavior even if a fast-forward merge is possible. This is useful for preserving the history of feature branches.

```
$ git merge feature_branch
```


In this scenario, if `main` and `feature_branch` have both progressed independently, Git will create a new commit that incorporates changes from both branches. This new commit will have two parent commits: one from each branch, effectively merging their histories.

Resolving merge conflicts

Conflicts can occur when merging branches. The most common situation is when one file has been committed twice. Let's look at a version conflict in more detail using our TaskFlow app as an example.

Scenario: Bob and Mike both made changes to the same file,

`TaskBoard.js`. Bob added drag-and-drop functionality, while Mike added real-time updates. When they try to merge their branches into

`main`, Git detects a conflict because both branches modified the same lines in `TaskBoard.js`.

Auto-merging TaskBoard.js

CONFLICT (content): Merge conflict in TaskBoard.js

Automatic merge failed; fix conflicts and then commit the result

Git will insert conflict markers in the file like this:

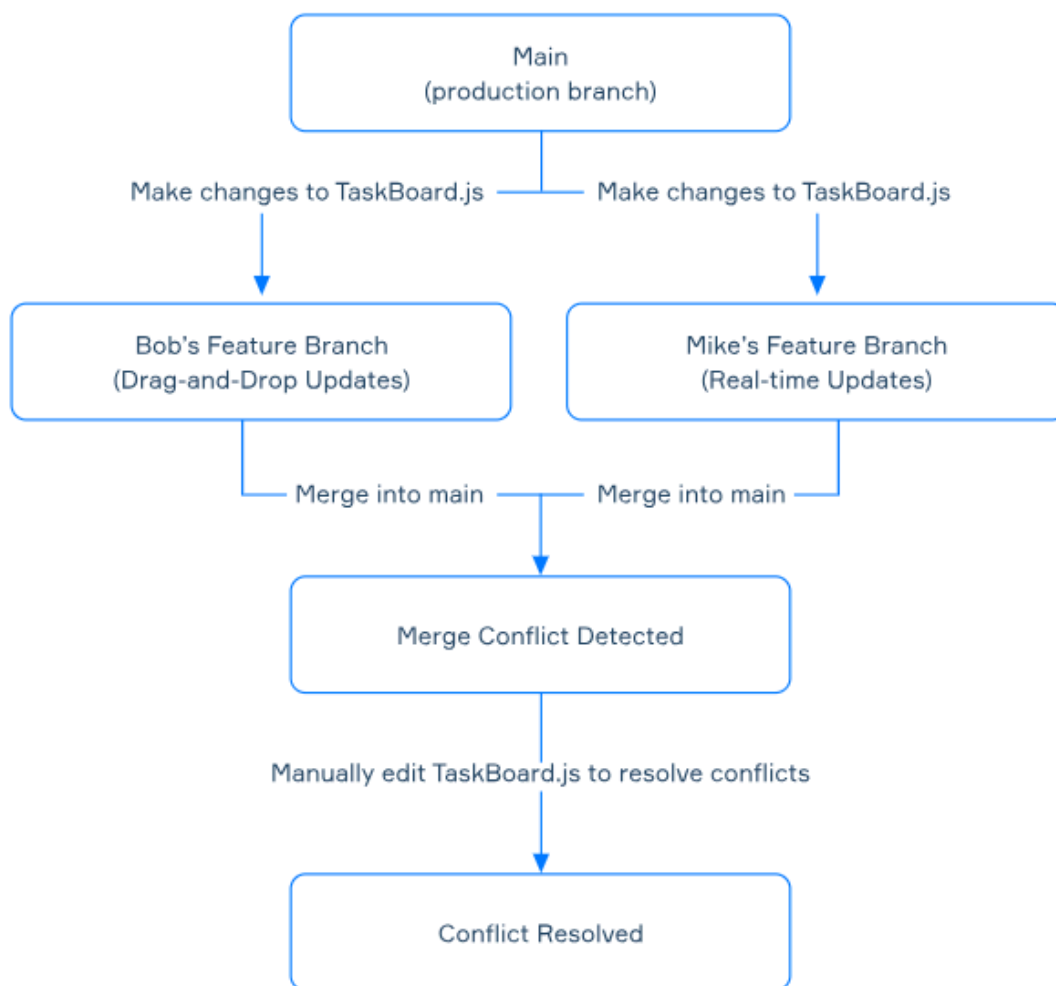
```
<<<<<<< HEAD
function updateTaskBoard() {
    // Bob's drag-and-drop code
}
=====
function updateTaskBoard() {
    // Mike's real-time update code
}
>>>>>>> mike-real-time
```

To resolve the conflict, Bob must manually edit the file to combine both his and Mike's changes, then remove the conflict markers and commit the resolved file:

```
$ git add TaskBoard.js
```

```
$ git commit -m "Resolved merge conflict between drag-and-drop  
and real-time updates"
```

Here is a visual representation of this setup:



Expand the section below to see a demo of a merge conflict as well as how it is resolved:

Resolving merge conflicts demo

```
$ # initialize a new git repository
$ git init conflict-demo
Initialized empty Git repository in /root/conflict-demo/.git/
$ cd conflict-demo
$ # create a new file, add some content, and commit
$ echo "This is the original content." > conflict-demo.txt
$ git add conflict-demo.txt
$ git commit -m "Initial commit"
[main (root-commit) 95784df] Initial commit
 1 file changed, 1 insertion(+)
 create mode 100644 conflict-demo.txt
$ # create a new feature branch, make changes, and commit
$ git checkout -b feature-branch
Switched to a new branch 'feature-branch'
$ echo "This is content from the feature branch." > conflict-demo.txt
$ git add conflict-demo.txt
$ git commit -m "Modified conflict-demo.txt on feature-branch"
[feature-branch d06059a] Modified conflict-demo.txt on feature-branch
 1 file changed, 1 insertion(+), 1 deletion(-)
$ # switch back to main and make conflicting changes
$ git switch main
Switched to branch 'main'
$ echo "This is content from the main branch." > conflict-demo.txt
$ git add conflict-demo.txt
$ git commit -m "Modified conflict-demo.txt on main"
[main 8bce1dd] Modified conflict-demo.txt on main
 1 file changed, 1 insertion(+), 1 deletion(-)
$ # try to merge the feature branch into main
$ git merge feature-branch
Auto-merging conflict-demo.txt
CONFLICT (content): Merge conflict in conflict-demo.txt
Automatic merge failed; fix conflicts and then commit the result.
```

```
$ # check file
```

```
$ cat conflict-demo.txt
```

```
<<<<<< HEAD
```

```
This is content from the main branch.
```

```
=====
```

```
This is content from the feature branch.
```

```
>>>>>> feature-branch
```

```
$ nvim conflict-demo.txt
```

```
$ git add conflict-demo.txt
```

```
$ git commit -m "Resolved merge conflicts"
```

```
$ git merge feature-branch
```

```
Already up to date.
```

```
1 $ # initialize a new git repository
2 $ git init conflict-demo
3 Initialized empty Git repository in /root/conflict-demo/.git/
4 $ cd conflict-demo
5 $ # create a new file, add some content, and commit
6 $ echo "This is the original content." > conflict-demo.txt
7 $ git add conflict-demo.txt
8 $ git commit -m "Initial commit"
9 [main (root-commit) 95784df] Initial commit
0   1 file changed, 1 insertion(+)
1   create mode 100644 conflict-demo.txt
2 $ # create a new feature branch, make changes, and commit
3 $ git checkout -b feature-branch
4 Switched to a new branch 'feature-branch'
5 $ echo "This is content from the feature branch." > conflict-demo.txt
6 $ git add conflict-demo.txt
7 $ git commit -m "Modified conflict-demo.txt on feature-branch"
8 [feature-branch d06059a] Modified conflict-demo.txt on feature-branch
9   1 file changed, 1 insertion(+), 1 deletion(-)
0 $ # switch back to main and make conflicting changes
1 $ git switch main
2 Switched to branch 'main'
3 $ echo "This is content from the main branch." > conflict-demo.txt
4 $ git add conflict-demo.txt
5 $ git commit -m "Modified conflict-demo.txt on main"
6 [main 8bce1dd] Modified conflict-demo.txt on main
7   1 file changed, 1 insertion(+), 1 deletion(-)
8 $ # try to merge the feature branch into main
9 $ git merge feature-branch
0 Auto-merging conflict-demo.txt
1 CONFLICT (content): Merge conflict in conflict-demo.txt
2 Automatic merge failed; fix conflicts and then commit the result.
3 $ # check file
4 $ cat conflict-demo.txt
5 <<<<<< HEAD
6 This is content from the main branch.
7 =====
8 This is content from the feature branch.
9 >>>>>> feature-branch
0 $ nvim conflict-demo.txt
1 $ git add conflict-demo.txt
2 $ git commit -m "Resolved merge conflicts"
3 $ git merge feature-branch
4 Already up to date.
```

As you can see, the process can be tedious and may become complex in larger projects. Many developers prefer to use GUI clients to resolve conflicts, but it isn't necessary for simple ones like the one we discussed here. Anyway, before doing anything, it is better to discuss it with your colleagues.

In collaborative projects, it's common to set up branch protection rules to prevent direct commits to critical branches like `main`. These rules can enforce code reviews, require passing tests, or restrict who can push to the branch. While not a Git command, these rules are typically configured in your Git hosting platform (e.g., GitHub, GitLab) and help maintain code quality and stability.

Stashing changes

Sometimes, you may find yourself in the middle of working on a branch when you need to switch to another branch. However, you might not be ready to commit your changes yet. In this case, you can use

`git stash` to temporarily save your work without committing it.

```
$ git stash
```

This command saves your changes and reverts your working directory to match the last commit. You can switch branches, and when you're ready to continue working on your original branch, you can apply the stashed changes:

```
$ git stash apply
```

Expand the section below to see a demo of stashing changes and then restoring them:

`git stash demo`

```
$ # initialize a git repository
$ git init stash-demo
Initialized empty Git repository in /root/stash-demo/.git/
$ cd stash-demo
$ # create a file, add content, and commit
$ echo "<h1>Initial version</h1>" > index.html
$ git add index.html
$ git commit -m "Initial commit with index.html"
[main (root-commit) e0a10e1] Initial commit with index.html
1 file changed, 1 insertion(+)
create mode 100644 index.html
$ # create a new branch to start working on a feature
$ git checkout -b feature-branch
Switched to a new branch 'feature-branch'
$ echo "<p>Working on a new feature...</p>" >> index.html
$ # check the file's content
$ cat index.html
<h1>Initial version</h1>
<p>Working on a new feature...</p>
$ # stash your changes
$ git stash
Saved working directory and index state WIP on feature-branch: e0a10e1
Initial commit with index.html
$ # check the file's content and notice the feature branch changes are gone
$ cat index.html
<h1>Initial version</h1>
$ # switch back to the main branch to make some changes
$ git switch main
Switched to branch 'main'
$ echo "<p>Bug fix applied</p>" >> index.html
$ git add index.html
$ git commit -m "Fixed a bug in index.html"
[main 4a16dac] Fixed a bug in index.html
1 file changed, 1 insertion(+)
```

\$ # now go back to feature branch and restore stashed changes

\$ git switch feature-branch

Switched to branch 'feature-branch'

\$ git stash apply

On branch feature-branch

Changes to be committed:

(use "git restore --staged <file>..." to unstage)

modified: index.html

\$ cat index.html

<h1>Initial version</h1>

<p>Working on a new feature...</p>

\$ git commit


```
1 $ # initialize a git repository
2 $ git init stash-demo
3 Initialized empty Git repository in /root/stash-demo/.git/
4 $ cd stash-demo
5 $ # create a file, add content, and commit
6 $ echo "<h1>Initial version</h1>" > index.html
7 $ git add index.html
8 $ git commit -m "Initial commit with index.html"
9 [main (root-commit) e0a10e1] Initial commit with index.html
10 1 file changed, 1 insertion(+)
11 create mode 100644 index.html
12 $ # create a new branch to start working on a feature
13 $ git checkout -b feature-branch
14 Switched to a new branch 'feature-branch'
15 $ echo "<p>Working on a new feature...</p>" >> index.html
16 $ # check the file's content
17 $ cat index.html
18 <h1>Initial version</h1>
19 <p>Working on a new feature...</p>
20 $ # stash your changes
21 $ git stash
22 Saved working directory and index state WIP on feature-branch: e0a10e1 Initial
23 $ # check the file's content and notice the feature branch changes are gone
24 $ cat index.html
25 <h1>Initial version</h1>
26 $ # switch back to the main branch to make some changes
27 $ git switch main
28 Switched to branch 'main'
29 $ echo "<p>Bug fix applied</p>" >> index.html
30 $ git add index.html
31 $ git commit -m "Fixed a bug in index.html"
32 [main 4a16dac] Fixed a bug in index.html
33 1 file changed, 1 insertion(+)
34 $ # now go back to feature branch and restore stashed changes
35 $ git switch feature-branch
36 Switched to branch 'feature-branch'
37 $ git stash apply
38 On branch feature-branch
39 Changes to be committed:
40   (use "git restore --staged <file>..." to unstage)
41       modified:   index.html
42
43 $ cat index.html
44 <h1>Initial version</h1>
45 <p>Working on a new feature...</p>
46 $ git commit
```

Conclusion

Here's a recap of what you've learned:

- Branches in Git allow you to work on different versions of a project simultaneously. You can create a branch using
- `git branch <branch_name>`.
- To switch between branches, use `git switch` or the older `git checkout`.
- Merging branches applies changes from one branch to another. Use `git merge` for this.
- To delete a branch, use `git branch -d`.
- Be mindful of merge conflicts, which can happen when different branches modify the same parts of a file. Keep track of changes, especially in collaborative projects.
- You can set up branch protection rules to enforce best practices, like requiring code reviews or passing tests before merging.
- Use `git stash` to temporarily save your work if you need to switch branches without committing your changes.

By mastering Git branches, you can manage parallel development, avoid conflicts, and ensure smooth collaboration and integration in your projects.

A branch is a copy of the project

John has recently started using git. John created a branch called `my_branch` and tried to add `text.txt` file to this newly created branch. But, after the `git add text.txt` command, the file was instead added to the `main` branch. What command `did` John forget to use? `git switch my_branch`

Experimental branch with awesome routine ⓘ

Sort the git commands so that the following points are carried out one after another.

1. Check the branches you have,
2. then make a new branch called "experimental",
3. switch to this new branch,
4. make changes by adding a file,
5. then commit these changes,
6. and switch back to the main **branch**.

Put the items in the correct order

⋮ git branch

⋮ git branch experimental

⋮ git checkout experimental

⋮ git add Tom-Paris-holodeck-routine.hs

⋮ git commit -m "Awesome routine! Finally done!"

⋮ git checkout main

In this repository, a file named `file.txt` is created with the content "Original content". This version is committed. Later, `file.txt` is modified so that it now contains: "Modified content".

Your task is to:

- Add the changes to the staging area.
- Stash the changes (which will remove them from the working tree).
- Then apply the stash so that
- `file.txt`
- once again reflects the modifications.

Write your Git command(s) below line by line.

Write your Git command(s) below line by line

git add .

git stash

git stash apply


Switch back and forth

Sort the commands so that the following actions take place one after another.

- First, check the branches you have,
- merge the "experimental" branch,
- switch to "redundant" branch,
- check the state of the repository and staging area,
- then switch back to the main branch,
- delete "redundant" branch.

Sort the commands so that the following actions take place one after another.

1. First, check the branches you have,
2. merge the "experimental" branch,
3. switch to "redundant" branch,
4. check the state of the repository and staging area,
5. then switch back to the main branch,
6. delete "redundant" branch.

 See hint

Put the items in the correct order

⋮ git branch

⋮ git merge experimental


⋮ git checkout redundant

⋮ git status

⋮ git checkout main

⋮ git branch -d redundant

Match the items from left and right columns

 Get un

merging

:: applying changes from one branch to another

switching

:: transitioning from one branch to another

branching

:: splitting project versions

stashing

:: temporarily saving uncommitted changes (both staged and unstaged) without committing them

Continue

Working with branches

[Report a typo](#)

In this repository, an initial commit has already been made on the main branch.

Your task is to:

- Create a new branch (**feature**) and switch to it;
- Create a new file named **new_feature.txt** and write the following content into it: "This is a new feature"; Stage and commit **new_feature.txt** with a commit message ("Add new feature").
- Switch back to the main branch and merge the feature branch into it;
- Do not delete the **feature** branch.

Write your Git command(s) below line by line.

Write your Git command(s) below line by line

```
git checkout -b feature
```

```
echo "This is a new feature" > new_feature.txt
```

```
git add ./new_feature.txt
```

```
git commit -m "Add new feature"
```

```
git switch main
```

```
git merge feature
```