4. Syncing Remote Branch List

If remote branches were created or deleted by others, update your local list using:

```
git fetch --all --prune
```

- `--all` fetches data from all remotes.
- `--prune` removes references to deleted remote branches.

5. Viewing Branch Details

To view more details, including the latest commit on each branch:

```
git branch -vv
```

Example Output:

```
* main                2f3b7c1 [origin/main] Update README
  feature/login-page  a8d9c4e Add authentication handler
```

Switching Branches (git checkout <branch> or git switch <branch>)

Switching branches in Git lets you move between different lines of development in your repository. This allows you to isolate changes, test new features, or return to previous work without affecting other branches.

1. Using git checkout

Traditionally, `git checkout` was used to switch branches.

```
git checkout <branch_name>
```

Example:

```
git checkout feature/login-page
```

This moves your working directory to the `feature/login-page` branch, updating your files to match its last commit.

2. Using git switch (Recommended)

Git introduced `git switch` in version 2.23 to make branch operations simpler and clearer.

```
git switch <branch_name>
```

Example:

```
git switch feature/dashboard
```

Both commands achieve the same goal, but `git switch` is easier to read and less error-prone.

3. Switching and Creating a New Branch

If the branch doesn't exist yet, you can create and switch to it in one step.
With `checkout`:

```
git checkout -b <new_branch_name>
```

With `switch`:

```
git switch -c <new_branch_name>
```

Example:

```
git switch -c feature/payment-api
```

4. Switching Back to the Previous Branch

You can return to the last branch you were on using:

```
git switch -
```

or

```
git checkout –
```

Example:

```
git switch –
```

This toggles between your current and last branch.

5. Safety Note

Before switching branches:

- Commit or stash any uncommitted changes to avoid losing work.
- Git won't allow you to switch if doing so would overwrite uncommitted modifications.

To stash changes temporarily:

```
git stash
```

Then switch branches safely.

Merging Branches (git merge <branch>)

Merging in Git combines the work from one branch into another, integrating changes so your project stays consistent and up to date. It's one of the key steps in collaborative development, especially when features are developed in separate branches.

1. Purpose of Merging

When you finish work in a feature branch (for example, `feature/login`), you typically merge it into the main branch (e.g., `main` or `develop`) to include your updates.

2. Basic Merge Process

Step 1: Switch to the Target Branch

First, move to the branch you want to merge **into** (usually `main`):

```
git switch main
```

Step 2: Merge the Source Branch

Then merge the feature branch into it:

```
git merge feature/login
```

Example Output:

```
Updating a3c5e3f..b4d2f67
Fast-forward
 src/login.js | 10 ++++++++++
 1 file changed, 10 insertions(+)
```

If no conflicting changes exist, Git performs a **fast-forward merge**, simply moving the branch pointer forward.

3. Types of Merges

- **Fast-forward merge:** Happens when the target branch hasn't diverged — the branch pointer just moves forward.
- **Three-way merge:** Occurs when both branches have new commits. Git creates a new "merge commit" combining both histories.

4. Handling Merge Conflicts

If the same file was changed differently in both branches, Git can't decide automatically — this causes a **merge conflict**.

Example Output:

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

To resolve:
1. Open the conflicted file.
2. Look for conflict markers like:

```
<<<<<<< HEAD
code from main
=======
code from feature/login
>>>>>>> feature/login
```

3. Edit the file to keep the desired version.
4. Mark conflict as resolved:

```
git add index.html
```

5. Complete the merge:

```
git commit
```

5. Aborting a Merge

If something goes wrong or you want to stop the merge process:

```
git merge --abort
```

This restores your branch to the state before the merge began.

6. Checking Merge History

To verify merges in your history:

```
git log --oneline --graph --decorate
```

You'll see a visual representation of branches and merge commits.

Fast-Forward vs Three-Way Merge

When you merge branches in Git, the way the merge occurs depends on how the branches' commit histories relate to each other. The two main merge types are **Fast-Forward** and **Three-Way Merge**. Understanding these helps you control your project's history and avoid confusion when collaborating.

1. Fast-Forward Merge

A **fast-forward merge** happens when the branch you're merging into has not diverged — meaning no new commits were made there since the source branch was created.

In this case, Git doesn't need to create a new merge commit; it simply **moves the branch pointer forward** to the latest commit of the source branch.

Example:

Before merge:

```
A --- B --- C (main)
             \
              D --- E (feature)
```

After merge (`git merge feature` while on `main`):

```
A --- B --- C --- D --- E (main)
```

Git simply "fast-forwards" `main` to point at commit `E`.

Command Example:

```
git switch main
git merge feature
```

Output:

```
Updating c3f5d2e..e7b1a9c
Fast-forward
 src/app.js | 12 ++++++++++++
 1 file changed, 12 insertions(+)
```

Advantages:

- Keeps the history clean and linear.
- No unnecessary merge commit.

Disadvantages:

- Loses visibility of the separate feature branch once merged (the branch pointer moves forward).

2. Three-Way Merge

A **three-way merge** occurs when both branches have diverged — meaning each has unique commits that the other doesn't.

Git uses **three snapshots** to perform the merge:

- The **common ancestor** commit.
- The **HEAD** commit of the current branch.
- The **tip** commit of the branch being merged.

Git then combines changes from both sides into a **new merge commit**.

Example:
Before merge:

```
A --- B --- C --- D (main)
       \
        E --- F (feature)
```

After merge:

```
A --- B --- C --- D --- G (main)
       \               /
        E --- F (feature)
```

Commit `G` is the new **merge commit** that combines both histories.

Command Example:

```
git switch main
git merge feature
```

Output:

```
Merge made by the 'recursive' strategy.
 src/app.js | 10 ++++++++++
 1 file changed, 10 insertions(+)
```

Advantages:

- Preserves full branch history.
- Clearly shows when and where branches diverged and merged.

Disadvantages:

- Creates extra merge commits, which can clutter history if used frequently.

3. Forcing a Merge Commit (Even if Fast-Forward is Possible)

If you want to preserve the fact that a branch was merged (for example, to retain feature branch context), you can force a three-way merge:

```
git merge --no-ff feature
```

This creates a merge commit even if a fast-forward merge is possible.

Tip:

- Use **fast-forward merges** for minor changes or solo work.
- Use **three-way merges** (with `--no-ff`) in collaborative workflows to clearly document when a feature branch was merged into the main branch.

Resolving Merge Conflicts

When two branches modify the same part of a file differently, Git can't automatically decide which changes to keep — this results in a **merge conflict**. You'll need to resolve it manually before completing the merge.

Steps to Resolve Merge Conflicts

1, Attempt the merge:

```
git merge <branch-name>
```

If there are conflicts, Git will pause the merge and show a message like:

```
CONFLICT (content): Merge conflict in file.txt
```

2. Check which files have conflicts:

```
git status
```

Files marked as "unmerged" are where conflicts exist.

3. Open and edit conflicted files:

Inside the file, Git adds conflict markers:

```
<<<<<<< HEAD
your current branch's content
=======
content from the branch being merged
>>>>>>> branch-name
```

- Keep the desired changes.
- Remove the conflict markers (`<<<<<<<`, `=======`, `>>>>>>>`).

4. Mark the conflict as resolved:

After editing and saving:

```
git add <filename>
```

5. Complete the merge:

```
git commit
```

Git automatically creates a merge commit unless you're in a rebase or squash workflow.

Deleting Branches (Local and Remote)

Cleaning up unused branches helps maintain a tidy repository. Git lets you delete branches both locally and remotely once they're merged or no longer needed.

1. Deleting a Local Branch

To delete a local branch that's **already merged** into the current branch:

```
git branch -d <branch-name>
```

Example:

```
git branch -d feature/login
```

If Git prevents deletion because the branch isn't fully merged, and you're sure you want to remove it, use the **force delete** option:

```
git branch -D <branch-name>
```

2. Deleting a Remote Branch

To delete a branch from the remote repository (e.g., GitHub, GitLab):

```
git push origin --delete <branch-name>
```

Example:

```
git push origin --delete feature/login
```

Alternatively (older syntax):

```
git push origin :<branch-name>
```

3. Verifying Deletion

List local branches:

```
git branch
```

List remote branches:

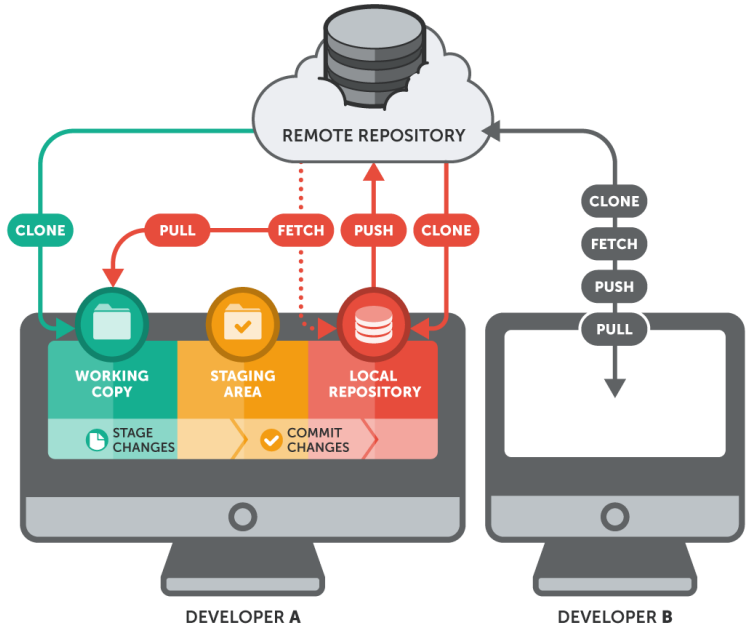
```
git branch -r
```

Remove references to deleted remote branches:

```
git fetch -p
```

(`-p` or `--prune` removes stale branch references that no longer exist on the remote.)

Working with Remotes



Adding a Remote Repository (git remote add origin <url>)

A *remote repository* is a version of your project hosted on a server (e.g., GitHub, GitLab, Bitbucket). Adding a remote allows you to push and pull code between your local and online repositories.

1. Add a Remote Repository

Use the following command to connect your local repository to a remote:

```
git remote add origin <url>
```

Example

```
git remote add origin
https://github.com/username/repository.git
```

Example (SSH):

```
git remote add origin
git@github.com:username/repository.git
```

Here:

- `origin` — is the conventional name for the main remote repository.
- `<url>` — is the link to your remote repository.

2. Verify the Remote

After adding the remote, confirm it's correctly configured:

```
git remote -v
```

Output example:

```
origin  https://github.com/username/repository.git (fetch)
origin  https://github.com/username/repository.git (push)
```

3. Changing or Removing a Remote

If you ever need to **change** the remote URL:

```
git remote set-url origin <new-url>
```

To **remove** a remote:

```
git remote remove origin
```

4. Initial Push to Remote

After adding the remote, push your main branch for the first time:

```
git push -u origin main
```

The `-u` flag sets `origin/main` as the default upstream branch for future pushes and pulls.

Viewing Remote Repositories (git remote -v)

Remote repositories are external versions of your project hosted on servers like GitHub, GitLab, or Bitbucket. Git allows you to easily view and manage these remotes.

1. View Configured Remotes

To list all remotes associated with your repository:

```
git remote -v
```

Example output:

```
origin  https://github.com/username/repository.git (fetch)
origin  https://github.com/username/repository.git (push)
```

Here:

- `origin` is the name of the remote.
- The URLs show where Git fetches and pushes data.

2. View Only Remote Names

If you only want to see the names of the configured remotes:

```
git remote
```

Example output:

```
origin
upstream
```

3. View Detailed Information

For more details about a specific remote, use:

```
git remote show origin
```

This command displays:

- The remote URL
- The branches fetched and pushed
- The upstream tracking branches

- Merge and push configuration

Example:

```
* remote origin
  Fetch URL: https://github.com/username/repository.git
  Push  URL: https://github.com/username/repository.git
  HEAD branch: main
  Remote branches:
    main tracked
  Local branches configured for 'git pull':
    main merges with remote main
```

4. When to Use This Command

- Use `git remote -v` when you:
- Clone a repository and want to confirm its remote.
- Add or update remote connections.
- Work on multiple remotes (e.g., origin, upstream).

Fetching Updates (git fetch)

The `git fetch` command downloads the latest commits, branches, and tags from a remote repository — **without merging or modifying your local code**. It's a safe way to check for updates before integrating them.

1. Basic Fetch Command

To fetch all updates from the default remote (usually `origin`):

```
git fetch
```

This updates your local copy of remote branches, but your current branch remains unchanged.

2. Fetch from a Specific Remote

If you have multiple remotes configured:

```
git fetch <remote-name>
```

Example:

```
git fetch upstream
```

This pulls updates only from the `upstream` remote.

3. Fetch a Specific Branch

To fetch updates from a particular branch:

```
git fetch origin <branch-name>
```

4. Fetch and Prune Stale References

Over time, deleted branches on the remote may still appear locally. To remove these outdated references:

```
git fetch -p
```

or

```
git fetch --prune
```

5. Viewing Fetched Changes

After fetching, you can review what's new using:

```
git log HEAD..origin/main
```

This shows commits that exist on `origin/main` but not in your current branch.

6. Why Use git fetch?

- To **check remote updates** before merging or pulling.
- To **inspect changes** made by teammates without altering your local work.
- To **keep your local branch references** in sync with the remote repository.

Pushing Changes (git push origin main)

The `git push` command uploads your local commits to a remote repository, allowing others (or remote services like GitHub) to access your latest changes.

1. Basic Push Command

To push your local branch (e.g., `main`) to the remote repository (`origin`):

```
git push origin main
```

Here's what happens:

- `origin` refers to the remote repository name.
- `main` is the local branch you want to push.
- Git transfers your commits to the corresponding remote branch.

2. Pushing for the First Time

If your local branch is new and doesn't yet exist on the remote:

```
git push -u origin main
```

The `-u` (or `--set-upstream`) flag links your local branch with the remote one, so future pushes can be done simply with:

```
git push
```

3. Pushing All Branches

To push all local branches to the remote:

```
git push --all
```

4. Pushing Tags

To share all tags with the remote:

```
git push --tags
```

5. Force Pushing (Use with Caution)

If you need to overwrite changes on the remote (for example, after a rebase):

```
git push --force
```

or safer:

```
git push --force-with-lease
```

Use force pushing carefully, as it can rewrite history and remove others' commits.

6. Verifying the Push

After pushing, confirm that your branch is up to date with:

```
git status
```

or view your changes on the remote platform (GitHub, GitLab, etc.).

7. Common Use Cases

- Pushing new commits to share progress.
- Syncing local work with a remote repository.
- Updating tags or branches after merging locally.

Pulling Updates (git pull origin main)

The `git pull` command downloads and integrates changes from a remote repository into your current local branch. It's essentially a combination of two commands:

```
git fetch + git merge
```

1. Basic Pull Command

To fetch and merge updates from the remote `main` branch into your current local branch:

```
git pull origin main
```

Here:

- `origin` is the name of the remote repository.
- `main` is the branch you want to update from.
- Git will automatically merge the fetched changes into your working branch.

2. Default Remote Tracking

If you previously set an upstream branch using:

```
git push -u origin main
```

you can simply run:

```
git pull
```

Git will know which remote and branch to pull from automatically.

3. Handling Merge Conflicts

If changes in your local branch conflict with updates from the remote, Git will pause the merge and mark the conflicts.

To resolve them:

1. Open the conflicted files and edit them manually.

2. Stage the resolved files:

```
git add <filename>
```

3. Complete the merge:

```
git commit
```

4. Using Rebase Instead of Merge

If you prefer a cleaner history (without extra merge commits):

```
git pull --rebase origin main
```

This applies your local commits **on top** of the fetched commits instead of merging them.

5. Pulling All Branches

You can also update all branches by running:

```
git pull --all
```

6. Checking Changes After Pull

To review what new commits were added:

```
git log origin/main..HEAD
```

Or check the summary of file changes:

```
git diff HEAD@{1} HEAD
```

7. When to Use git pull

- To stay in sync with your teammates' latest commits.
- After pushing to ensure no conflicts exist.
- Before creating new features or merging branches.

Removing or Renaming Remotes

Managing your remotes helps keep your repository clean and organized. Git allows you to easily **remove** outdated remotes or **rename** existing ones without affecting your local branches or commits.

1. Viewing Existing Remotes

Before making any changes, list all configured remotes:

```
git remote -v
```

Example output:

```
origin   https://github.com/username/repository.git (fetch)
origin   https://github.com/username/repository.git (push)
upstream https://github.com/another/repo.git (fetch)
```

2. Removing a Remote

To delete a remote connection:

```
git remote remove <remote-name>
```

or the older syntax:

```
git remote rm <remote-name>
```

Example:

```
git remote remove upstream
```

This removes the remote reference from your repository configuration.

Note: This does *not* delete the remote repository itself — it only removes your local link to it.

3. Renaming a Remote

To change the name of a remote:

```
git remote rename <old-name> <new-name>
```

Example:

```
git remote rename origin github
```

After renaming, verify the change:

```
git remote -v
```

Output:

```
github  https://github.com/username/repository.git (fetch)
github  https://github.com/username/repository.git (push)
```

4. When to Remove or Rename a Remote

- You've migrated from one hosting service to another (e.g., GitHub → GitLab).
- The old remote is no longer needed or accessible.
- You want clearer naming conventions (e.g., renaming `origin` to `main-repo`).

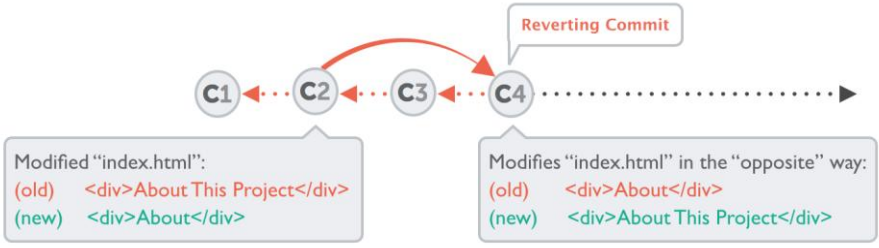
5. Verify Configuration Changes

After removing or renaming, you can double-check your `.git/config` file:

```
cat .git/config
```

This ensures the remote list reflects your updates.

Undoing Changes



Discarding Local Changes (git restore <file>)

The `git restore` command lets you safely **discard uncommitted changes** in your working directory — restoring files to their last committed state. It's especially useful when you want to undo accidental edits or reset specific files without affecting others.

1. Basic Syntax

To discard changes in a specific file:

```
git restore <file>
```

Example:

```
git restore index.html
```

This replaces the working copy of `index.html` with the version from the last commit.

2. Discard All Local Changes

To revert ALL modified files in your working directory:

```
git restore .
```

This resets every file to match the last commit on your current branch.

3. Restoring Files from a Specific Commit

You can restore a file to its state in a particular commit:

```
git restore --source <commit-hash> <file>
```

Example:

```
git restore --source a1b2c3d4 script.py
```

This replaces your working copy of `script.py` with the version from that specific commit.

4. Restoring Staged Files

If you've already staged a file (using `git add`) but want to unstage and discard its changes:

```
git restore --staged <file>
```

This removes the file from the staging area while also discarding its modifications.

5. Important Notes

`git restore` affects **only uncommitted changes**.

Once restored, discarded edits **cannot be recovered** unless you had them saved elsewhere.

For older Git versions (before 2.23), use:

```
git checkout -- <file>
```

6. When to Use git restore

- To revert files accidentally edited.
- Before committing, to remove unwanted changes.
- To recover a clean working directory matching your last commit.

Unstaging Files (git reset HEAD <file>)

Sometimes, you may accidentally stage a file using `git add` that you didn't intend to include in your next commit. The `git reset HEAD <file>` command lets you **unstage** that file — removing it from the staging area without deleting or modifying its changes in the working directory.

1. Basic Syntax

To unstage a specific file:

```
git reset HEAD <file>
```

Example:

```
git reset HEAD app.js
```

This moves `app.js` out of the staging area, but any edits you made remain in your working directory.

2. Unstaging Multiple Files

You can unstage several files at once:

```
git reset HEAD <file1> <file2> <file3>
```

Or simply unstage everything:

```
git reset HEAD
```

3. Confirming the Change

Check which files are still staged or unstaged:

```
git status
```