Rebase & Squash

What rebase is, how it differs from merge, squashing/reordering/rewording commits with git rebase -i, resolving rebase conflicts, the danger of rewriting shared history, and the difference between git push --force and --force-with-lease.

Intermediate 35 min

Last updated: August 27, 2026
Read online: https://www.learnforgex.com/en/topics/rebase-and-squash

---

You already know one way to bring a branch's changes together with `main`: merging. This lesson covers a second way — rebasing — which produces a cleaner, linear history but comes with real risk if used carelessly on shared work. By the end, you'll know exactly when rebase is the right tool, when it isn't, and how it connects to squashing commits into a clean, reviewable unit.

What Is Rebase?

Rebasing takes the commits from one branch and replays them, one by one, on top of a different starting point — typically the latest `main`. The commits themselves get new hashes (they're technically brand-new commits with the same changes and messages), but the *content* of your work is preserved exactly.

Merge vs Rebase

Both solve the same underlying problem — "bring `main`'s new commits into my feature branch" — but produce different history shapes:

```
Merge:                          Rebase:

main    A---D---------M         main    A---D
             \       /                       \
feature       B---C--                 feature  B'---C'   (replayed on top of D)
```

- **Merge** preserves exactly what happened, including a merge commit showing when the branches came together — history is a true, if sometimes messy, record.
- **Rebase** rewrites your branch's commits as if they'd been written starting from the latest `main` all along — history is a clean, linear story, at the cost of no longer perfectly reflecting what actually happened.

Neither is universally "correct" — many teams merge into shared branches like `main`, but rebase their own feature branches to keep them clean before opening a Pull Request.

git rebase

```
git switch add-password-reset
git rebase main
```

This replays every commit unique to `add-password-reset` on top of `main`'s current tip. If `main` hasn't changed since you branched, there's nothing to do. If it has, each of your commits gets reapplied one at a time.

Rebasing a Feature Branch onto main

```
#!/bin/sh
# Rebasing a feature branch onto the latest main, then resolving a straightforward
# conflict during the replay.

git switch add-password-reset
git rebase main
# Output:
# Auto-merging PasswordService.java
# CONFLICT (content): Merge conflict in PasswordService.java
# error: could not apply e4f5a6b... Add password validation
# hint: Resolve all conflicts manually, mark them as resolved with
# hint: "git add/rm <conflicted_files>", then run "git rebase --continue".

# ... open PasswordService.java, resolve the conflict markers ...
git add PasswordService.java
git rebase --continue
# Output:
# Successfully rebased and updated refs/heads/add-password-reset.

# The branch's commits now sit on top of main's latest commit, with new hashes.
git log --oneline -3
# Output:
# 9d8e7f6 Add password validation
# 8c7d6e5 Add password field to User entity
# a1b2c3d Add rate limiting to login endpoint   (main's latest, now the base)

# The branch was already pushed once before this rebase, so a normal push is
# rejected -- the rebased commits don't match what's on the remote.
git push
# Output:
# ! [rejected]        add-password-reset -> add-password-reset (non-fast-forward)
# error: failed to push some refs

# force-with-lease pushes the rebased history, but would refuse if origin had
# changed unexpectedly since our last fetch.
git push --force-with-lease
# Output:
# + e4f5a6b...9d8e7f6 add-password-reset -> add-password-reset (forced update)
```

Interactive Rebase — git rebase -i

`git rebase -i` (interactive rebase) opens an editor listing your branch's commits and lets you decide what happens to *each one* — not just replay them, but reorder, combine, edit, or drop them:

```
git rebase -i HEAD~4    # interactively rebase the last 4 commits
```

Git opens something like this:

```
pick a1b2c3d Add password field to User entity
pick e4f5a6b Add password validation
pick 7c8d9e0 Fix typo in validation message
pick 1f2a3b4 Add tests for password validation

# Commands:
# p, pick <commit> = use commit
# s, squash <commit> = use commit, but meld into previous commit
# f, fixup <commit> = like squash, but discard this commit's message
# r, reword <commit> = use commit, but edit the commit message
# d, drop <commit> = remove commit
```

This is genuinely a text file you edit — change the words on each line, save, and Git carries out your instructions.

Squashing Commits

Change `pick` to `squash` (or its shorthand `s`) on any commit to merge it into the commit *above* it:

```
pick a1b2c3d Add password field to User entity
squash e4f5a6b Add password validation
squash 7c8d9e0 Fix typo in validation message
squash 1f2a3b4 Add tests for password validation
```

After saving, Git combines all four commits into one and lets you write a single, clean commit message for the result — turning a messy work-in-progress history into one focused commit. `fixup` does the same merge but silently discards the fixed-up commit's message, which is exactly what you want for commits like "fix typo" that add nothing to the final message.

Reordering Commits

Since the rebase list is just lines in a file, reordering commits is as simple as reordering the lines — Git applies them top to bottom, in whatever order you leave them.

Editing Commit Messages

Change `pick` to `reword` (or `r`) on a commit to keep it exactly as-is, but pause the rebase so you can edit its message — useful for fixing a typo or clarifying an old commit's message without touching its actual changes.

Resolving Rebase Conflicts

Because rebase replays commits one at a time, a conflict can happen on *any* individual commit, not just once at the end. When it does, Git pauses mid-rebase:

```
# after resolving the conflicted file(s):
git add ConflictedFile.java
git rebase --continue
```

If a particular commit's conflict turns out to be more trouble than it's worth, or you decide rebasing wasn't the right move after all:

```
git rebase --abort    # cancel the whole rebase, return to how things were before
```

The full mechanics of resolving conflict markers are covered in the dedicated "Merge Conflicts" lesson — the process is the same whether the conflict came from a merge or a rebase.

When NOT to Rebase

The single most important rule about rebase: never rebase commits that other people have already pulled. Because rebase gives every replayed commit a new hash, anyone who already has the *old* versions of those commits now has a history that has diverged from yours — Git sees them as unrelated commits, not the "same work, moved." Their next pull will be confusing at best and produce duplicate, tangled history at worst.

The safe rule of thumb: rebase freely on your own feature branch, as long as you haven't pushed it yet (or nobody else has pulled your pushed version). Never rebase `main`, or any branch other people are actively working from.

Why Rebase Can Require Force Push

Since rebasing changes your commits' hashes, your local branch and its already-pushed remote counterpart now have completely different (though similar-looking) histories. A normal `git push` will be rejected — Git correctly sees this as "the remote has commits I don't have" and refuses, as a safety check. To push your rebased branch, you have to explicitly override that check.

git push --force vs git push --force-with-lease

Both push your rebased history over what's on the remote — but with an important safety difference:

```
git push --force origin add-password-reset
git push --force-with-lease origin add-password-reset
```

- `--force` overwrites the remote branch unconditionally — including any commits someone else may have pushed there since you last fetched, silently discarding them.

- `--force-with-lease` checks first: if the remote branch has changed since you last fetched (meaning someone else pushed something you haven't seen), it refuses and fails safely instead of overwriting their work.

Prefer `--force-with-lease` whenever a force push is genuinely necessary — it gives you all the same power with a real safety net. We cover this distinction in even more depth, along with when force-pushing is and isn't appropriate, in the dedicated "Force Push" lesson later in this course.

Common Mistakes

- **Rebasing a branch other people have already pulled** — this is the single most common way rebase causes real problems on a team.
- **Reaching for plain `--force` out of habit** when `--force-with-lease` would have caught an unexpected remote change.
- **Panicking and aborting mid-conflict without understanding what state you're in** — `git rebase --abort` is always safe and returns you to exactly where you started; use it freely if a rebase conflict gets confusing.

Best Practices

- Rebase your own, not-yet-shared feature branches to keep history clean before opening a Pull Request.
- Use `git rebase -i` to squash "fix typo"-style commits into their parent before requesting review.
- Default to `--force-with-lease`, not `--force`, any time you need to push a rebased branch.

Summary, Cheat Sheet, and Glossary

Summary

- Rebase replays your branch's commits on top of a new base (usually `main`), producing linear history at the cost of new commit hashes.
- `git rebase -i` lets you squash, reorder, reword, or drop commits before they're shared.
- Never rebase commits other people have already pulled — it diverges their history from yours.
- A rebased, already-pushed branch needs a force push; prefer `--force-with-lease` over plain `--force` since it refuses to overwrite unexpected remote changes.

Cheat Sheet

```
git rebase main                  # replay your branch's commits onto main
git rebase -i HEAD~4             # interactively edit the last 4 commits
git rebase --continue            # resume after resolving a conflict
git rebase --abort               # cancel, return to pre-rebase state
git push --force-with-lease      # safely push a rebased branch
```

Glossary

- **Rebase** — replaying a branch's commits on top of a different base commit, producing new hashes.
- **Interactive rebase** — a rebase where you choose pick/squash/fixup/reword/drop for each commit.
- **Force-with-lease** — a force push that refuses if the remote has changed unexpectedly since your last fetch.