Jim's Remote Git Adventure

Week 2: Collaborating with the Team

Previously on Jim's Journey

> *Last week, Jim learned local Git commands. He can now create commits, branch, merge, and resolve conflicts. But he's been working alone on his computer. Today, he joins the team's GitHub repository to collaborate with developers around the world!*

What Jim will learn today:

- Level 1: Connecting to the Cloud (Remote Repositories)
- Level 2: Sharing Work (Push)
- Level 3: Getting Updates (Fetch & Pull)
- Level 4: Team Conflicts (Merge Conflicts with Remote)
- Level 5: Advanced Collaboration (Pull Requests)
- Level 6: Temporary Drawer (Stash)
- Level 7: Force push (--force)
- Level 8: Best Practices for Teams

Level 1: Connecting to the Cloud

Monday Morning Meeting

> *"Welcome to week two, Jim!" says his manager. "Time to join our GitHub repository. You'll be working with Sarah in Cincinnati and Alex in New York. Here's our repository URL."*

Jim's new challenge:

- Connect his local repository to the team's GitHub
- Learn about **remote repositories**
- Understand the difference between local and remote

Jim Learns About Remotes

> *Sarah explains: "Think of a remote repository as a shared gallery in the cloud. Everyone on the team can contribute their snapshots there!"*

The Remote Repository Concept:

- **Local Repository**: Jim's personal gallery on his computer
- **Remote Repository**: Shared gallery on GitHub
- **Origin**: The default name for the main remote
- Multiple developers can push and pull from the same remote

Jim Adds His First Remote

> *Jim has been working on the user-profile feature locally. Now he needs to connect it to the team's repository.*

```
> git remote add origin https://github.com/company/webapp.git

> git remote -v
origin  https://github.com/company/webapp.git (fetch)
origin  https://github.com/company/webapp.git (push)
```

What happened:

- Added a remote named **origin**
- Can now **fetch** (download) from it
- Can now **push** (upload) to it

Jim Tries to Push

> *"Perfect! Now I can push my work!" Jim exclaims. But when he tries...*

```
$ git push origin main
Username for 'https://github.com': jim@company.com
Password for 'https://jim@company.com@github.com':
remote: Support for password authentication was removed on August 13, 2021.
remote: Please see https://github.blog/2020-12-15-token-authentication-requirements-for-git-operations/ for more information.
fatal: Authentication failed
```

The problem:

- GitHub no longer accepts passwords
- Jim needs proper authentication
- Even public repos need auth for push!

Understanding Repository Access

> *Sarah explains: "Let me break down repository permissions for you, Jim."*

Public Repositories:

- **Read** (clone/fetch): Anyone can do this
- **Write** (push): Need authentication + permission

Private Repositories:

- **Read** (clone/fetch): Need authentication + permission
- **Write** (push): Need authentication + permission

Authentication ≠ Authorization!

- **Authentication**: Proving who you are
- **Authorization**: Having permission to do something

Two Authentication Methods

> *"You have two main options," Sarah continues. "Personal Access Tokens for HTTPS, or SSH keys."*

Option 1: Personal Access Token (PAT)

- Works with HTTPS URLs
- Acts like a password
- Can set expiration and scope
- Easy to revoke if compromised

Option 2: SSH Keys

- Works with SSH URLs
- Public/private key pair
- More secure
- One-time setup per machine

Jim Sets Up a Personal Access Token

> *Jim decides to start with a Personal Access Token.*

Steps:

1. Go to GitHub → Settings → Developer settings
2. Personal access tokens → Tokens (classic)
3. Generate new token
4. Select scopes (at minimum: `repo` for full control)
5. Copy the token (you won't see it again!)

```
$ git push origin main
Username: jim@company.com
Password: gph_xxxxxxxxxxxxxxxxxxxx  # PAT
```

Now, we can use the PAT to access repositories, even private repositories.

```
git clone https://oauth2:gph_xx@github.com/company/webapp.git
```

It works — **but it's dangerous**.

⚠️ Problems

- **Leaked secret** in shell history, logs, and process lists
- **Stored in** `.git/config` (plain text, easy to share by accident)
- **Hard to rotate** — must fix every repo manually
- **Policy violations** — URLs with tokens often blocked by scanners

Storing Credentials Safely

> *"I don't want to type my token every time!" Jim complains. Sarah shows him credential helpers.*

```
# On macOS
$ git config --global credential.helper osxkeychain

# On Windows
$ git config --global credential.helper wincred

# On Linux
$ git config --global credential.helper cache --timeout=3600
```

Now Jim only enters his token once!

⚠️ **Never commit tokens to your repository!**

Jim Learns SSH Authentication

> *"SSH is even better for long-term use," Alex chimes in. "Let me show you!"*

Step 1: Generate SSH key

```
$ ssh-keygen -t ed25519 -C "jim@company.com" # replace it with yours
Generating public/private ed25519 key pair.
Enter file in which to save the key (/home/jim/.ssh/id_ed25519): [Enter]
Enter passphrase (empty for no passphrase): [Enter passphrase]
```

Step 2: Add public key to GitHub

```
$ cat ~/.ssh/id_ed25519.pub
# Copy this output
```

Then: GitHub → Settings → SSH and GPG keys → New SSH key

Switching to SSH URL

> *"Now change your remote to use SSH instead of HTTPS," Alex instructs.*

```
# Check current remote
$ git remote -v
origin  https://github.com/company/webapp.git (fetch)
origin  https://github.com/company/webapp.git (push)

# Change to SSH
$ git remote set-url origin git@github.com:company/webapp.git

# Verify the change
$ git remote -v
origin  git@github.com:company/webapp.git (fetch)
origin  git@github.com:company/webapp.git (push)
```

No more username/password prompts!

Testing SSH Connection

> *"But wait, how do I know my SSH is working?" Jim asks.*

```
$ ssh -T git@github.com
Hi jim! You've successfully authenticated, but GitHub does not
provide shell access.

# Now push works without prompting!
$ git push origin main
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Writing objects: 100% (3/3), 284 bytes | 284.00 KiB/s, done.
```

Success! SSH authentication is set up.

Private Repository Access

> *Now, Jim can create private repositories and access them.*

1. Using PAT (HTTPS)

```
git clone https://oauth2:ghp_XXX@github.com/company/secret-project.git
```

2. Using SSH

```
git clone git@github.com:company/secret-project.git
```

Jim understands that he should be ready for both cases as some GIT tools require PAT and some SSH.

Private Team Repository Access

> *Later, Jim needs to clone a private repository. "I have authentication, but I still can't access it!" he says.*

```
$ git clone git@github.com:company/secret-project.git
Cloning into 'secret-project'...
ERROR: Repository not found.
fatal: Could not read from remote repository.
```

The issue:

- Authentication works (WHO you are)
- But Jim lacks authorization (WHAT you can do)
- Repository owner must grant access!

Getting Repository Access

> *The team lead explains: "For private repos, I need to add you as a collaborator."*

For organization repos:

1. Admin goes to repository settings
2. Access → Collaborators
3. Invites Jim by username/email
4. Jim accepts invitation

After accepting:

```
$ git clone git@github.com:company/secret-project.git
Cloning into 'secret-project'...
remote: Enumerating objects: 156, done.
# Success!
```

Understanding Remote Branches

> *"But wait," Jim asks, "how do I see what's on the remote?"*

```
$ git branch -a
* main
  feature/user-profile
  remotes/origin/main
  remotes/origin/feature/authentication
  remotes/origin/feature/payment
```

Jim discovers:

- Local branches: `main`, `feature/user-profile`
- Remote branches: Start with `remotes/origin/`
- Other teammates' work is visible!

Level 2: Sharing Work (Push)

> *Jim finished his user profile feature. His manager says: "Great work! Push it to GitHub so Sarah can review it."*

Jim's first push challenge:

- Share his local commits with the team
- Make his branch available on GitHub
- Learn the push workflow

Jim's First Push

> *Jim takes a deep breath. "It's time to share my work with the world!"*

```
$ git push origin main
Enumerating objects: 15, done.
Counting objects: 100% (15/15), done.
Delta compression using up to 8 threads
Compressing objects: 100% (10/10), done.
Writing objects: 100% (15/15), 2.34 KiB | 1.17 MiB/s, done.
To https://github.com/company/webapp.git
   a1b2c3d..f4e5d6c  main -> main
```

Success! Jim's commits are now on GitHub!

Push Rejection

> *Later, Jim tries to push again but encounters an error!*

```
$ git push origin main
To https://github.com/company/webapp.git
 ! [rejected]        main -> main (fetch first)
error: failed to push some refs to 'https://github.com/company/webapp.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
```

What happened?

- Sarah pushed changes while Jim was working
- Remote has commits Jim doesn't have
- Git prevents accidental overwrites!

Pushing a New Branch

> *Jim creates a new feature branch and wants to share it with the team.*

```
$ git checkout -b feature/dark-mode-v2
$ # ... make some commits ...

$ git push origin feature/dark-mode-v2
Total 0 (delta 0), reused 0 (delta 0)
To https://github.com/company/webapp.git
 * [new branch]      feature/dark-mode-v2 -> feature/dark-mode-v2
```

New branch created on remote!

Setting Upstream Branch

> *"Typing 'origin feature/dark-mode-v2' every time is tedious," Jim complains. Sarah shows him a shortcut!*

```
$ git push -u origin feature/dark-mode-v2
Branch 'feature/dark-mode-v2' set up to track remote branch
'feature/dark-mode-v2' from 'origin'.

# Now Jim can just use:
$ git push
$ git pull
```

The `-u` flag:

- Sets the **upstream** branch
- Links local and remote branches
- Simplifies future commands

Clone sets up main's tracking automatically.

You only need -u for new branches you create locally!

Switching Between Branches

> *"I see," Jim says, "I set my feature branch to track origin, but when I checkout to main, I work with main's upstream settings!"*

```
# Currently on feature branch
$ git branch -vv
* feature/dark-mode-v2  f8g9h0 [origin/feature/dark-mode-v2] Feature work
  main                  c3d4e5 [origin/main] Main branch

# Switch to main
$ git checkout main
Switched to branch 'main'
Your branch is up to date with 'origin/main'.

# Now git commands use main's upstream
$ git pull  # Pulls from origin/main ✅
$ git push  # Pushes to origin/main ✅

# Switch back to feature
$ git checkout feature/dark-mode-v2
$ git pull  # Now pulls from origin/feature/dark-mode-v2 ✅
$ git push  # Now pushes to origin/feature/dark-mode-v2 ✅
```

> *Sarah smiles, "Exactly! Each branch remembers its own upstream. When you checkout, you're switching to that branch's settings. It's like each branch has its own remote control!"*

The key concept:

- Upstream is **per-branch**, not global
- `checkout` switches both your local branch AND which upstream you're using
- Each branch independently tracks its corresponding remote branch

Level 3: Getting Updates (Fetch & Pull)

> *Jim's Slack notification pops up: "Sarah: Just pushed the new authentication system! Alex: Added the payment module!"*
> *Jim realizes: "I need to get their changes!"*

Two ways to get updates:

1. **Fetch**: Download changes but don't merge
2. **Pull**: Download AND merge in one step

Jim Learns to Fetch

> *Jim wants to see what's new without affecting his current work.*

```
$ git fetch origin
remote: Enumerating objects: 25, done.
remote: Counting objects: 100% (25/25), done.
From https://github.com/company/webapp
   f4e5d6c..a9b8c7d  main       -> origin/main
   * [new branch]      feature/auth -> origin/feature/auth
```

What fetch does:

- Downloads new commits from remote
- Updates remote-tracking branches
- Does NOT change Jim's working files
- Safe to do anytime!

Inspecting Fetched Changes

> *"I fetched the changes, but where are they?" Jim wonders.*

```
$ git log --oneline main..origin/main
a9b8c7d Add user authentication system
b8c7d6e Fix security vulnerability

$ git diff main origin/main
# Shows all changes between local and remote main
```

Jim can now:

- See what's new before merging
- Review teammates' changes
- Decide when to integrate updates

Jim Pulls Updates

> *Jim reviewed the changes and they look good. Time to integrate them!*

```
$ git pull origin main
Updating f4e5d6c..a9b8c7d
Fast-forward
 auth.js     | 45 ++++++++++++++++++++++++
 security.js | 12 +++++++
 2 files changed, 57 insertions(+)
```

What happened:

- Git fetched the latest changes
- Then merged them into Jim's current branch
- "Fast-forward" = no conflicts!

Pull vs Fetch + Merge

> *Sarah explains: "Pull is just fetch + merge in one command!"*

```
# These two are equivalent:

# Option 1: Pull
git pull origin main

# Option 2: Fetch then merge
git fetch origin
git merge origin/main
```

When to use each:

- **Pull**: When you're ready to integrate immediately
- **Fetch + Merge**: When you want to review first

Level 4: Team Conflicts

> *Jim and Alex both modified the same configuration file. When Jim tries to pull Alex's changes, disaster strikes!*

```
$ git pull origin main
Auto-merging config.js
CONFLICT (content): Merge conflict in config.js
Automatic merge failed; fix conflicts and commit the result.
```

Jim's first remote conflict!

Understanding the Conflict

> *Jim opens config.js and sees the familiar conflict markers:*

```
const config = {
<<<<<<< HEAD (Current Chanage)
    theme: 'dark',
    language: 'en-US',
=======
    theme: 'light',
    language: 'en-GB',
>>>>>>> a9b8c7d (Incoming Change)
    version: '2.0'
};
```

The conflict:

- Jim set theme to 'dark' and language to 'en-US'
- Alex set theme to 'light' and language to 'en-GB'
- Git can't decide which to keep!

Jim Resolves Diplomatically

> *Jim messages Alex: "Hey, we both changed the config. Should we support both themes?" Alex replies: "Good idea! Let's make it configurable!"*

```
const config = {
    theme: getUserPreference('theme', 'light'),
    language: getUserPreference('language', 'en-US'),
    version: '2.0'
};
```

```
git add config.js # resolve the issue
git commit -m "Merge with Alex's changes - make theme/language configurable"
git push origin main
```

Lesson learned: Communication prevents conflicts!

The Dreaded Merge Conflict Loop

> *Jim, Sarah, and Alex all push to main within minutes of each other. Jim pulls and resolves conflicts, but when he tries to push...*

```
$ git push origin main
 ! [rejected]        main -> main (non-fast-forward)
# Someone else pushed while Jim was resolving!

$ git pull origin main
# More conflicts!
```

The solution: Work on feature branches!

Level 5: Advanced Collaboration (Pull Request)

> *The team lead calls a meeting: "We're implementing branch protection and Pull Requests!"*

Jim's new collaborator status means:

- ✅ Can push to feature branches
- ✅ Can create pull requests
- ❌ Cannot push directly to main (protected!)
- ❌ Cannot merge without review

Branch Protection Setup

> *"Watch this," the team lead says, configuring the repository settings. "I'm protecting our main branch."*

Team Lead's Configuration:

1. Settings → Branches → Add rule
2. Branch pattern: `main`
3. Protections enabled:
    - ✅ Require pull request reviews
    - ✅ Require approved reviews: 1
    - ✅ Dismiss stale reviews
    - ✅ Include administrators

Now when Jim tries direct push:

```
$ git push origin main
remote: error: GH006: Protected branch update failed
remote: error: At least 1 approving review is required
! [remote rejected] main -> main (protected branch hook declined)
```

The Pull Request Workflow

> *"Don't worry," Sarah reassures Jim. "This protection helps us maintain code quality. Here's our new workflow!"*

The Pull Request Workflow:

1. Create feature branch
2. Push to remote
3. Open Pull Request (PR)
4. Team reviews code
5. Merge when approved

Jim's First Pull Request

> *Jim implements a new search feature on a separate branch.*
>
> *Since he's a collaborator, he can push feature branches!*

```
$ git checkout -b feature/search
$ # ... implement search feature ...
$ git add .
$ git commit -m "Add search functionality"

$ git push -u origin feature/search
remote: Create a pull request for 'feature/search' on GitHub by visiting:
remote: <https://github.com/company/webapp/pull/new/feature/search>
```

Next steps:

- Jim opens the URL
- Writes a description
- Requests review from Sarah

Keeping Feature Branch Updated

> *While Jim waits for review, the main branch gets new updates. Sarah comments: "Please update your branch with the latest main."*

```
$ git checkout main
$ git pull origin main

$ git checkout feature/search
$ git merge main
# Resolve any conflicts if they exist

$ git push origin feature/search
```

Sarah Reviews and Approves

> *Sarah receives a notification about Jim's pull request. "Let me review your search feature," she says, opening GitHub.*

Sarah's Review Process:

1. **Checks the PR description** - Clear explanation ✅
2. **Reviews the code changes** - Clean implementation ✅

3. **Tests locally** (if necessary):

```
git fetch origin pull/42/head:pr-42
git checkout pr-42
npm test
```

> Testing is one of the most important processes in software engineering; in this scenario, we assume Sarah just runs a quick test to approve. In a real world, we should run exhaustive tests always.

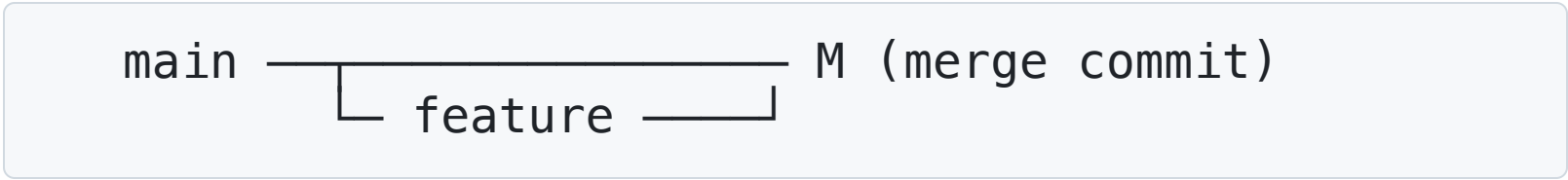
4. **Leaves feedback** - Suggests minor improvements
5. **Approves the PR** - "LGTM! (Looks Good To Me)"

The Merge Process

> *With Sarah's approval, Jim can now merge his PR. GitHub shows him merge options, but Sarah advises: "For now, just use the default 'Create a merge commit' - it's the safest!"*

The Default Merge Option:

Create a merge commit - Preserves full history



Why this is best for beginners:

- ✅ It's the default option
- ✅ Preserves all your commit history
- ✅ Works just like `git merge` locally
- ✅ Can't accidentally lose work
- ✅ Easy to understand what happened

> *"Later, when you're more experienced, you can explore squash and rebase options," Sarah adds. "But merge commits are perfectly fine for most situations!"*

Jim Completes His First PR

> *"I'll click the green 'Merge pull request' button," Jim says, following Sarah's advice to use the default option.*

On GitHub:

1. Jim clicks "Merge pull request"
2. Confirms by clicking "Confirm merge"
3. Sees "Pull request successfully merged and closed"
4. GitHub shows option to delete the feature branch

Back in terminal:

> *"Wait," Jim realizes, "the merge happened on GitHub's server. My local main branch doesn't have the changes yet!"*

```
$ git checkout main
$ git status
Your branch is behind 'origin/main' by 2 commits

$ git pull origin main
# NOW Jim's local main has the merged feature!

$ git branch -d feature/search  # Clean up local branch
Deleted branch feature/search (was f8g9h0).
```

> *"Congratulations on your first merged PR!" Sarah celebrates. "Your search feature is now live in the main branch!"*

Jim Completes His First PR (For Advanced Users Only)

> *"Since this is a clean feature with 3 commits, I'll use 'Squash and merge' to keep history tidy," Jim decides.*

On GitHub:

1. Jim clicks "Squash and merge"
2. Edits the commit message
3. Confirms the merge
4. GitHub automatically deletes the feature branch

Back in terminal:

The merge happened on GitHub's server; so the local main should be updated too.

```
$ git checkout main
$ git pull origin main
# Jim's feature is now in main!

$ git branch -d feature/search  # Clean up local branch
Deleted branch feature/search (was f8g9h0).
```

Level 6: Temporary Drawer (Stash)

> *Jim is working on a new feature when suddenly his manager messages: "URGENT! Pull the latest main - there's a critical fix you need!"*

```
$ git status
Changes not staged for commit:
  modified:   profile.js
  modified:   utils.js

$ git pull origin main
error: Your local changes would be overwritten by merge.
Please commit your changes or stash them before you merge.
```

Jim's Manual Approach

> *"I'll just copy my files somewhere safe," Jim thinks, doing what many beginners try first.*

```
$ mkdir ~/tmp/backup
$ cp profile.js ~/tmp/backup/
$ cp utils.js ~/tmp/backup/

$ git checkout -- profile.js  # Discard changes
$ git checkout -- utils.js    # Discard changes

$ git pull origin main
# Success! But now...

$ cp ~/tmp/backup/profile.js .
$ cp ~/tmp/backup/utils.js .
# Manually restoring... tedious and error-prone!
```

Sarah Shows the Git Way

> *"Stop!" Sarah laughs. "Git has a built-in temporary drawer called stash. Let me show you the proper way!"*

```
$ git status
Changes not staged for commit:
  modified:   profile.js
  modified:   utils.js

$ git stash save "WIP: user profile improvements"
Saved working directory and index state

$ git pull origin main
# Clean pull - no conflicts!

$ git stash pop
# All changes perfectly restored!
```

> *"Much easier!" Jim exclaims. "No manual copying, no risk of forgetting files!"*

Level 7: Force push (--force)

When Normal Push Isn't Enough

> *"I can push fine usually," Jim says, "but sometimes Git refuses even though I'm a collaborator. Why?*

Normal push works when:

```
Local:  A—B—C—D—E (you added D,E)
Remote: A—B—C
# Push works! You're just ahead.
```

Force push needed when:

```
Local:  A—B—C' (you changed C)
Remote: A—B—C
# Push fails! Histories diverged.
```

Scenario 1: After Amending

> *Jim pushes a commit, then realizes he made a typo in the commit message.*

```
$ git push origin feature/login
# Success! Commit is now on GitHub

# Oh no, typo in commit message!
$ git commit --amend -m "Add login feature (not 'lofin')"

$ git push origin feature/login
! [rejected] feature/login -> feature/login (non-fast-forward)
# Git refuses! The commit changed.

$ git push --force origin feature/login
# Force overwrites the remote
```

Scenario 2: After Rebasing

> *Jim rebases his feature branch to get latest main changes.*

```
# Before rebase:
#     C—D (feature)
#    /
# A—B—E—F (main)

$ git rebase main
# After rebase:
#         C'—D' (feature – new commit hashes!)
#        /
# A—B—E—F (main)
```

```
$ git push origin feature/login
! [rejected] (non-fast-forward)

$ git push --force origin feature/login
# Must force because C,D became C',D'
```

Scenario 3: Removing Commits

> *"I accidentally committed a password file!" Jim panics.*

```
$ git log --oneline
f8g9h0 Fix styling
a1b2c3 Add password.txt (OH NO!)
e5d4c3 Add login form

# Remove the bad commit
$ git reset --hard e5d4c3
# Re-add the good commit
$ git cherry-pick f8g9h0

$ git push origin feature/login
! [rejected] (non-fast-forward)
$ git push --force origin feature/login
# Force push to rewrite history
```

The Golden Rules of Force push

> *Sarah shares the team's force push guidelines:*

✅ **SAFE to force push:**

- Your own feature branches
- Before anyone else pulls
- After communicating with team

❌ **NEVER force push:**

- To main/master branch
- To shared branches
- Without warning teammates

💡 **Safer alternative:**

```
$ git push --force-with-lease origin feature/login
# Fails if someone else pushed since you fetched
# Prevents accidentally overwriting others' work
```

Level 8: Best Practices

> *The team shares their hard-won wisdom about remote collaboration.*

The Team's Golden Rules:

1. **Always pull before you push**
2. **Work on feature branches, not main**
3. **Fetch regularly to stay updated**
4. **Communicate about conflicts**
5. **Review before merging**

Sarah's Fetch Habit

> *"I fetch every morning like reading the newspaper," Sarah shares. "It helps me see what the team is working on."*

```
# Sarah's morning routine
$ git fetch --all
$ git log --oneline --graph --all
$ git branch -vv  # See which branches are behind/ahead
```

Benefits:

- Stay informed without disrupting work
- Plan merges strategically
- Avoid surprise conflicts

Alex's Conflict Prevention

> *"I learned to check who's working on what files," Alex explains.*

```
# Before starting work on a file:
$ git fetch
$ git log origin/main -- file-to-edit.js
# See recent changes to this specific file

$ git blame file-to-edit.js
# See who last edited each line
```

Smart practices:

- Create smaller, focused commits
- Push frequently to signal activity

The Nuclear Option: Force Push

> *The team lead warns: "With great power comes great responsibility. Force push can rewrite history!"*

```
# DANGER! Only on YOUR feature branches
$ git push --force origin feature/my-branch

# Safer alternative:
$ git push --force-with-lease origin feature/my-branch
# Fails if others have pushed
```

⚠️ **NEVER force push to main!**

When it's okay:

- Your own feature branch
- After rebasing
- Team knows you're doing it

Pull Request Guideline (Make Commits Clean)

> *The team establishes PR guidelines for smooth collaboration. If you make multiple commits in your local repo, consider squash commits before requesting PR.*

Before pushing:

```
# Clean up your local commits
$ git rebase -i HEAD~3  # Squash related commits

# Before: Your feature branch
A—B (main)—C—D—E (feature/search - messy)
# After: Your feature branch (cleaned)
A—B (main)—C' (feature/search - one nice commit)
```

What if Main Moved Forward

```
# Your local state:
A—B————————C' (feature/search - your cleaned commit)
   \
    (main - outdated locally)

# But on GitHub:
A—B—X—Y—Z (origin/main - teammates added X,Y,Z!)
```

If you push now:

- Your PR will be behind main by 3 commits
- Might have conflicts
- Can't be merged cleanly

Do pull or pull --rebase

Option 1: `git pull origin main`:

```
$ git pull origin main  # Creates merge commit

A—B—X—Y—Z—┐
   \       \
    C'———————M (feature/search with ugly merge commit)
```

Option 2: `git pull --rebase origin main`:

```
$ git pull --rebase origin main  # Replays on top

A—B—X—Y—Z—C'' (feature/search - clean!)
```

NOW you can push

If you changed any remote history with `git rebase`.

```
> git push --force origin feature/search
```

If you don't change any remote history with `git rebase`.

```
> git push origin feature/search
```

Why this order matters:

1. Clean your commits first (easier to resolve conflicts once)
2. Then update with latest main
3. Result: Clean commits sitting on top of latest main
4. Perfect for PR review!

Good PR practices:

- Clear, descriptive titles
- Link to issue/ticket
- Add screenshots for UI changes
- Keep PRs focused and small

The Team's Workflow

> *Now, the team has established a smooth rhythm:*

Daily workflow:

```
# Morning
git fetch --all
git pull origin main

# Before starting new work
git checkout -b feature/new-feature

# Regular commits
git add . && git commit -m "Clear message"
git push -u origin feature/new-feature

# Open PR and collaborate!
```