Practical Git & GitHub Workflow

A closing lesson that ties together everything taught across the course in one continuous, realistic Spring Boot scenario: clone, feature branch, commit/amend, push, Pull Request, review feedback, rebasing onto main, resolving a conflict, force-with-lease, and squash and merge.

Intermediate 30 min

Last updated: August 27, 2026
Read online: https://www.learnforgex.com/en/topics/practical-git-github-workflow

---

Every command in this course so far has been taught in isolation. This closing lesson puts all of it together into one continuous, realistic scenario — the kind of day a Java/Spring Boot developer actually has. There's no new syntax here; the goal is to see how everything you've learned connects into a single, natural workflow.

The Scenario

You're adding a due-date field to a `Task` entity in a small Spring Boot task-tracking API. The work is small enough for one feature branch, but real enough to involve a Pull Request, a round of review feedback, and a conflict with changes someone else merged into `main` while you were working.

Cloning and Starting a Feature Branch

You're joining this work for the first time, so you clone the repository, then start a feature branch off `main`:

```
git clone https://github.com/your-team/task-tracker.git
cd task-tracker
git switch -c add-task-due-date
```

Making and Reviewing Changes

You edit the entity, then check exactly what you changed before staging anything:

```
# ... edit Task.java to add a dueDate field ...
git status
git diff
```

Staging and Committing the Work

Satisfied with the diff, you stage and commit — and a moment later realize you forgot the matching database migration, so you amend instead of creating a separate "oops" commit:

```
git add Task.java
git commit -m "Add dueDate field to Task entity"

# ... realize the migration file is missing ...
git add V42__add_task_due_date.sql
git commit --amend --no-edit
```

You continue with a couple more focused commits as the feature progresses:

```
git add TaskController.java TaskDto.java
git commit -m "Expose dueDate in the Task API response"

git add TaskControllerTest.java
git commit -m "Add tests for dueDate serialization"
```

Pushing and Opening a Pull Request

The feature is ready for review, so you push it — using `-u` since this is the branch's first push — and open a Pull Request on GitHub, targeting `main`.

```
git push -u origin add-task-due-date
```

Responding to Review Feedback

A teammate reviews the PR and requests a change: the date should be serialized in ISO-8601 format explicitly, rather than relying on the default. You make the fix directly on the same branch and push again — no new PR needed, GitHub updates the existing one automatically:

```
# ... adjust the @JsonFormat annotation on Task.dueDate ...
git add Task.java
git commit -m "Serialize dueDate as ISO-8601"
git push
```

Catching Up With main

While your PR was under review, a teammate merged an unrelated change into `main`. Before merging, you want your branch tested against the latest `main` — so you fetch and rebase:

```
git fetch origin
git rebase origin/main
```

Resolving a Conflict During Rebase

The rebase hits a conflict — the other change also touched `Task.java`, on a nearby line. Git pauses mid-rebase:

```
#!/bin/sh
# Rebasing a feature branch onto an updated main, hitting a conflict in
# Task.java, and resolving it -- the middle of the full end-to-end scenario.

git fetch origin
git rebase origin/main
# Output:
# Auto-merging Task.java
# CONFLICT (content): Merge conflict in Task.java
# error: could not apply 4f5a6b7... Add dueDate field to Task entity
# hint: Resolve all conflicts manually, mark them as resolved with
# hint: "git add/rm <conflicted_files>", then run "git rebase --continue".

# Task.java now contains conflict markers:
#
# public class Task {
#     private String title;
# <<<<<<< HEAD
#     private Priority priority;
# =======
#     private LocalDate dueDate;
# >>>>>>> 4f5a6b7 (Add dueDate field to Task entity)
#     // ...
# }
#
# Both fields are needed -- someone else added `priority` on main while this
# branch added `dueDate`. The correct resolution keeps both.

# ... edit the file to keep both fields, remove all three marker lines ...

git add Task.java
git rebase --continue
# Output:
# Successfully rebased and updated refs/heads/add-task-due-date.

git log --oneline -1
# Output:
# 9e8f7a6 Add dueDate field to Task entity
```

Updating the Pull Request

Because the rebase rewrote your branch's commit hashes, and this branch was already pushed once, a plain `git push` is rejected. Since nobody else has pulled this feature branch, a force push is safe — and `--force-with-lease` is the safe way to do it:

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

GitHub's PR page updates automatically to show the rebased commits.

Completing the Pull Request

The reviewer approves. On GitHub, you choose **Squash and merge** — the branch's five work-in-progress commits (including the amend and the review-feedback fix) become a single, clean commit on `main`: "Add dueDate field to Task entity." GitHub offers a one-click "Delete branch" button on the merged PR page, which you use immediately.

Back in your terminal, you clean up your now-merged local branch and sync up with the new `main`:

```
git switch main
git pull
git branch -d add-task-due-date
```

The feature is live on `main`, the branch is gone (locally and remotely), and your local repository is fully caught up — exactly where you'd start the next feature from.

Key Takeaways

- This entire workflow is built from commands you already know individually — nothing here is new syntax, only a realistic order of operations.
- Amending, staging deliberately, and writing focused commits all happen *before* anything is shared — once pushed and under review, new work becomes new commits, not rewrites of what's already visible to the reviewer.
- Rebasing your own not-yet-merged feature branch onto an updated `main` — and force-pushing with `--force-with-lease` afterward — is a completely normal, safe part of working with a team, precisely because nobody else has pulled that branch.
- Squash and merge, then delete the branch, closes the loop cleanly: `main` gets one focused commit, and nothing about the branch lingers afterward.