Introduction to Version Control and Project Management with Git and GitHub: Part II

Dmitri Svetlov

School of Biomedical and Chemical Engineering, Colorado State University

Dmitri.Svetlov@ColoState.edu

CM 515 Spring 2026

Agenda

Branching Model

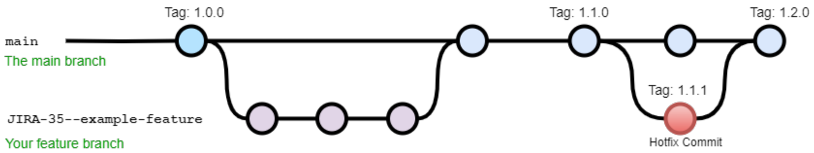
- Because of the intrinsic branching functionality of Git, it lends itself well to teams (and groups of teams) of developers.
- To be successful, any development team needs to adopt, and adhere to, a particular **branching model**:
  - What does each branch *mean*, philosophically and operationally?
  - Under what conditions can branches be created, deleted, and tagged?
  - Under what conditions can changes from one branch be merged into another?
  - How do the branches relate to other aspects of the team's/company's work, *e.g.* help-desk requests?

Naive Branching Models

- One approach is to simply have each developer own a "personal" branch, particularly if peers are working largely independently of one another and without much supervision. There are many downsides, especially at scale:
  - "Divergent evolution" to an extreme degree - merging across branches without conflicts will become prohibitively difficult.
  - The code will become highly idiosyncratic, with n developers adopting $\geq n$ ways of doing the same thing.
  - Someone ultimately has to call the shots - who will decide what is in the authoritative version of the code, and how?
- A better approach is to associate one branch with each intellectual change to the code, *i.e.* one bug fix or one new, standalone feature.
  - These tend to be called **feature branches** even when they are to fix a bug.

Trunk-and-Branch Model

- You can now see why `trunk` is often used for the name of the `main` branch - it is because of this model.
- This is very similar to what GitHub uses for their own development.
- A **tag** is what you'd expect: simply a human-readable reference to a particular commit. In this model, tags are used to identify versions for public (”production”) release.
- See also the following webpages: [1] and [2]

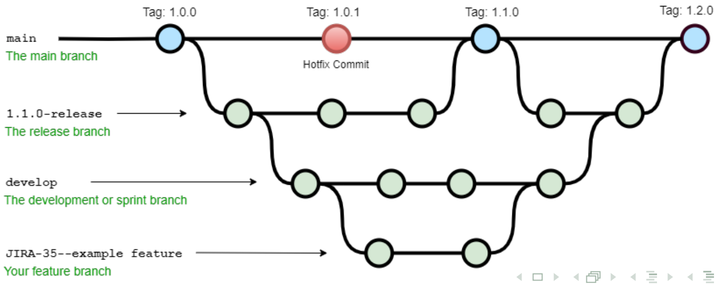


Toward More Complex Branching Models

- Several problems with having just one "mainline" branch:
  - `main` must **ALWAYS** be deployable, to whatever standard of correctness and reliability your code requires.
    - As a package grows in complexity and size (think millions of lines of code), this can become very difficult in practice.
  - You may need to simultaneously maintain, *i.e.* (customer) support, multiple versions that are very far apart historically.
  - You may have other constraints on information flow, the software development life cycle - *e.g.* for regulatory reasons, a public release might require validation that automated testing cannot handle.
- As a result, some workflows maintain multiple mainlines:
  - `stable`
  - `development`
  - `nightly`
  - `release`
  - *etc.*

A More Complex Gitflow

- This is for an Agile workflow, where releases are frequent (often, timed) and development accordingly occurs in ”sprints”; therefore, the branches have different interrelationships.
- There are other workflows that work, but how well suited they are is very situation-dependent.
- Find one that works for you - and stick to it!



What is GitHub?

- Partly, GitHub is a hosting service for Git repositories (and associated webpages, README files, wikis, *etc.*
    - There are competitors, *e.g.* GitLab and Bitbucket.
- But GitHub also provides additional functionality that is very useful.
    - Project management tools, *e.g.* issue tracking.
    - Software and hardware for automated testing.
    - Workflow-related tools - most notably, the **pull request**.

Centralized VCSes Revisited

- Remember that Git is a **decentralized** VCS. How then can we establish an authoritative Git repository, for a single point/source of truth?
- We can't do this *technically*, so we have to do it *socially*: we designate *by convention* "one repository to rule them all."
- All members in a team mutually agree to consider a particular repository to be authoritative. In most situations, this will be one hosted on a service like GitHub and/or deposited into a dedicated archive.
- This "contract" only holds as long as everyone abides by it!

Issue Tracking

- In project management, large and complex projects are decomposed into multiple constituent **tasks** interrelated in logical and/or sequential ways.
- In computing terms, an **issue** is a data structure representing and encapsulating a task and its defining information.
- Each issue has its own properties: assignee(s), milestone(s), priority, labels, related issue(s), dependencies, *etc*.
- This concept generalizes: if you've ever emailed an IT help-desk, your help-desk "ticket" was almost certainly an issue in that desk's instance of their ticketing system, *e.g.* ETS uses FreshService.
- Each ticket receives a unique, alphanumeric ID, *e.g.* `JIRA-35`.

Issue Tracking in Software Development and GitHub

- In software development, the best practice is that any new feature or bug fix will have a corresponding issue.
    - If the work is customer-originated, *e.g.* a customer reported the bug, the fixing issue will be "linked" to the help-desk issue (even if that's in a different system) so that the customer can be notified when the fix is made.
    - In some workflows, a feature branch's name will contain the issue ID (recall the mentions of `JIRA-35` in the diagrams above).
- Issues are an integral feature of GitHub repositories:
    - All of the fundamental issue-tracking functionality is provided, **plus**
    - Each issue has a webpage, where the comments can reference contributors, branch(es), and commits for the issue.
    - You can also reference - and even resolve! - issues in commit messages. If those commits are initially made to a local repository, the issues will get updated when you `push`.

Pull Requests

- Think back to our discussion of pushes and pulls.
- There are two common situations where you would like to push but cannot or should not:
  - You don't have write access, *e.g.* you want to make a contribution to an open-source repository but are not a full member.
  - You are reintegrating a feature branch, *i.e.* seeking to commit to `main`.
- Since you cannot *push*, you can only request that someone on the other end perform a *pull* (the same activity, remember, but the perspective is reversed).
- GitHub invented a formalization of this process: the **pull request**.

A Word about Forks

- In the first scenario above (lack of membership in a repository), the common solution is the following:
  1. Create (on GitHub) a **fork** of the repository. This will clone a specified branch of the repository and all upstream branches (up to and including `main`).
  2. Make changes on a branch of the forked repository (which you will own).
  3. When satisfied, create a pull request from the branch of your repository to the desired upstream branch of the upstream repository.
- In the second scenario, you should not create a fork. The pull request is created in the same way, but entirely within the one repository.

Anatomy of a Pull Request

- A pull request (PR) will contain some or all of the following, with its own GitHub webpage:
  - A discussion.
  - A list of all included commits and `diff` of all changes - this allows for integrated code review.
  - The results of any CI checks that were run as part of the PR. In particular, if this included pass/fail tests, their success or failure will be clearly visually indicated.
  - A resolution.
    - Using certain keywords, *e.g.* “close” and “resolve”, you can simultaneously approve the PR, merge the feature branch into the upstream branch, delete the feature branch, and successfully resolve the associated issue!
- The best practice is that any merge into `main` will only occur as the result of an approved PR, and GitHub will even enforce this requirement if you choose.

Commits

- Each commit should be as **atomic** as possible: it should exactly match an intellectual change to all the code and data changes needed to effect it.
- Beyond that, commit early and often!
- Write clear and descriptive commit messages!

Branches

- Like a commit, each branch should be as **atomic** as possible, but here the change is on a more permanent (and often larger) scale.
- Don't reintegrate upsteam (especially not to `main`) without a PR involving **both** code review **and** a perfect test run!
  - Test failures *can* be acceptable if their cause is understood and the implications of their failure on code correctness and reliability known.
- When working on GitHub, utilize the integrated functionality here: create branches directly from issue webpages and utilize PRs to handle both merging and subsequent feature branch deletion.

Issues

- Here again, we're looking for atomicity, but we'll soften that ambition for the sake of *responsibility*.
- Ever assigned one task to six people? How did that go?
- The standard solution in project management is to create **dependencies**, *i.e.* sub-tasks (sub-issues), so that no issue is ever assigned to more than one person at any time.
- Again, utilize GitHub's integrated functionality here, *e.g.* notifications and mentions of users, related issues and PRs, *etc.*

Automated Testing in Software Development

- Testing, particularly automated testing, is an integral and indispensable best practice of software development.
- **Test-driven development** (TDD) actually makes tests primary, in a sense:
  - In this philosophy, a software package is defined not by its code but the functionality it implements.
  - Tests are the means by which functionality can be empirically, reliably verified.
  - Therefore, you write the test(s) for a new feature first, then the code to implement the feature.
  - Upon any changes to the code, all tests associated with a package are run, to ensure that the package is functioning as expected.
    - Industrial best practice is to additionally run all tests on a daily or nightly basis.

Continuous Integration and Deployment (CI/CD)

- The practice of frequently (re)integrating code changes from distributed repositories to a shared one, and then automatically testing the result, is called **continuous integration** (CI).
- Analogously, **continuous deployment** (CD) refers to the automated (re)deployment of (new) versions of software upon the success of the CI checks.
- These practices are not universal, *e.g.* highly regulated code will probably not use CD, because of the validation procedures required.
- Nonetheless, any move toward greater CI/CD catches errors soon, speeds up and eases debugging, and minimizes merge conflicts.

CI/CD in GitHub

- GitHub provides a wealth of CI/CD functionality, much of which is available for free for public (*i.e.* open-source) repositories.
- To any repository, workflows of GitHub **Actions** can be added, each of which is then executed by one or more **runners** in response to one or more **events**.
- That's pretty vague, so here's an example:
  - <u>Event</u>: A push is made to a repository containing Python code and utilizing some third-party packages.
  - <u>Action</u>: Install a given version of Python and the third-party packages on a machine. Execute all automated tests within the repository. Email the results to the committer of the push.
  - <u>Runner</u>: GitHub provides its own runners for the three major desktop OSes. (Additionally, you can "self-host" a runner wherever you choose.)

Guided Practice

1. Online, sync your fork of the course GitHub repository to get the `ROSTER.md` file.
2. Create an issue on your fork to update the `ROSTER.md` file with your personal information. From the issue page, assign the issue to yourself and create a feature branch for the issue.
3. On your local machine, checkout the new feature branch (Hint: Which Git operation(s) will you need to perform first?).
4. Copy and paste the contents of the `NAME.md` file you created earlier (where `NAME` should be replaced with your name) into the appropriate portion of the `ROSTER.md` file. Commit and push the changes.
5. Online, in your fork, create and complete a pull request to reintegrate your feature branch into `main`.

Guided Practice, continued

6. Online, in the course repository, you will have an issue assigned to you. Find and open its webpage (Hint: How might you do this? Are there any ways GitHub might have notified you that were assigned an issue?)
7. Create a PR to complete the issue. This will involve a merge between the `main` branches of the course repository and of your fork. (Hint: In which direction will the merge need to go?) Use a closing keyword and the issue number so that if the PR is accepted, the issue will automatically be closed successfully, *i.e.* as completed.
8. Select an instructor and both request a code review from and assign the PR to that person.
9. Perform any remaining tasks indicated by the instructor in the code review or subsequent comments.

General Advice

- Practice, practice, practice!
- Words of advice:
  - Don't invent a non-standard workflow unless none of the tried-and-true ones works for you.
  - Team members should all use the same workflow.
  - Always understand where you are and what you're trying to accomplish with your commit/push/pull request.
    - If you must, draw a workflow diagram like the ones above.
    - Graphical tools, *e.g.* Git GUI, will show you where you actually are, so compare that to where you think you are.
  - To minimize conflicts, frequently merge from `main` into all open feature branches. Keep them only ahead of `main`, never behind.
    - Industrial best practice: do this at the start of every workday and whenever an upstream branch changes significantly.
  - **DON'T REWRITE HISTORY!**
- Good luck!

Back to start

References

[1] Juan Benet. *a simple git branching model (written in 2013)*. URL: https://gist.github.com/jbenet/ee6c9ac48068889b0912. (accessed: January 11, 2024).

[2] End of Line Blog. *OneFlow: a Git branching model and workflow*. URL: https://www.endoflineblog.com/oneflow-a-git-branching-model-and-workflow.