A Safe Workflow for Collaboration: Branch, Pull Request, Merge, Delete

Helene Wagner, University of Toronto

This tutorial assumes that you:

- have been invited to join a Github project,
- have installed git on your computer and made it work with RStudio,
- have installed GitKraken and know how to use the user interface,
- have cloned the repo to your computer as an R project with version control, and
- have mastered the basic workflow of pull - commit - push.

If that's not the case, please check the relevant sections under Installation (incl. how to clone a repo) or Workflows (see "Basic workflow tutorial").

Now we're ready to take it to the next level with a workflow that is recommended for collaborative coding. When you edit other team member's code, you want to let them know, and you don't want to make a change that is difficult to revert.

At this stage, we'll focus on these goals:

- Understand the workflow: what are the steps, and why are we doing this?
- Understand the implementation: how does GitHub "think"?
- Gain experience with a test file to keep things safe initially.
- Learn about some advanced git concepts.

The next tutorial "Surviving your first merge conflicts" will help you master your first merge conflicts and start building confidence around this dreaded event.

1. Challenges in collaborative coding

When team members collaborate on coding within the same file, a number of challenges are bound to arise:

- **Syncing**: in Google Docs, everyone sees the exact same file version at any given time and you can see changes made by team members in real time. In GitHub, every team member works on their local copy of a file, and these are not in sync automatically. We use "push" to make our changes available to team members, and "pull" to update our file with any changes they may have pushed.
- **Communication**: if one team member makes a change, the rest of the team needs to be alerted to that change.
- **Portability**: if the code depends on R objects that are available only in the local R session, it may not run on other team members' computers.
- **Redundancy**: each analysis step should be programmed only once. Otherwise, there is the risk that it will be changed in one place but not in another.

- **Consistency**: if one line of code is altered, other parts of the code may no longer run (or worse: they may still run without error messages but produce garbage).
- **Fall-back version**: if someone made a change that created a problem for other parts of the code (was it me?), you want to be able to identify and revert that change.

2. Important terms and concepts

Link to a series of GitKraken videos that explain many concepts (three lists: Beginner videos, Intermediate videos, Advanced videos): https://www.gitkraken.com/learn/git/tutorials

main	**The current "official" version of your code resides in the "main" branch of your repo.** ● In older repos, this branch may be labeled "master"
branch	**A branch is a parallel copy of the code that may be in a different state of development.** ● You can create, and delete, branches on the spot, and call them anything you want. ● GitHub pages used for developing R packages often have a branch called "dev" or "devel", where new features are being developed. ● You could name a branch by ownership (e.g. your first name or username), or by keyword (e.g., "data-import"). ● Make sure the branch name is a single word (no blanks).
merge	**The process of updating one branch (usually "main") with changes from another branch.** ● Merging "dev" into "main" means updating "main" with the changes made in "dev".
pull request	**Request a review of proposed changes and merging them into "main"** ● If you have "Maintainer" privilege for the repo, you can review and accept the changes yourself. You'll still need to let your team know about them. ● If you have "Contributor" status, a "Maintainer" will need to approve your pull request to merge it into "main".
merge conflict	**A merge conflict arises when the same line of code has changed in both branches since the time at which they separated.** ● Git will merge changes unless two users have committed changes to the same line of code. ● If the changes are on different lines, they will be merged, even if one change might create an error in another line of code. Git does not check for code consistency. ● For R Notebooks, code refers to the R code in the chunks, the text in the Notebook (which is programmed in R markdown language, thus it is also a form of code), and the yaml header.

remote vs. local	**Each branch can have a "remote" and a "local" version.** ● They can be synced with "pull" (from remote to local) and "push" (from local to remote). ● See the Basic workflow tutorial.

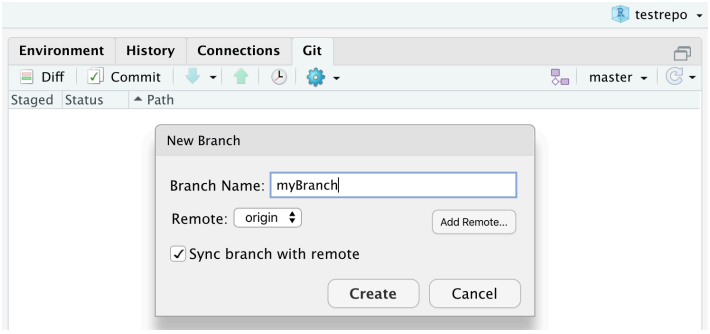
3. Workflow overview

When you start a programming session, you should do the following:

- Start with an up-to-date version (pull from the remote "main" branch)
- Isolate your work in progress (create a separate branch, both local and remote)
- Use the simple workflow as you go (pull from remote branch, commit, push to remote branch).
- To ensure that you will are pushing to the remote branch, not the “main”, double check that you are currently working in your new branch.
- When you are satisfied with the changes, ask your team to review them (create a pull request).
- If all is ok, update the "main" branch with your changes (merge branch into "main").

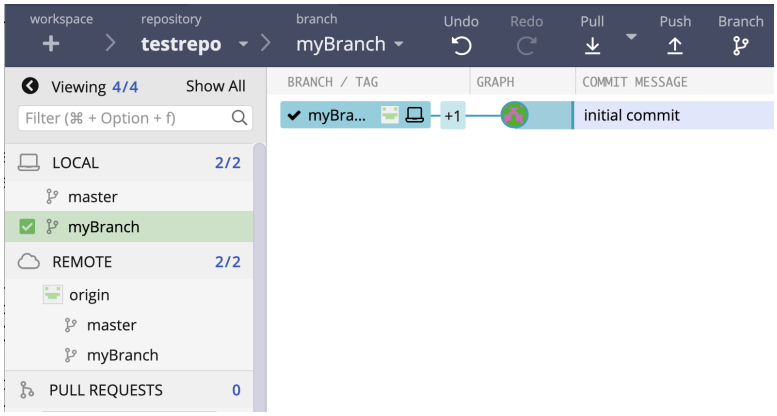
4. Step by step instructions

- Start with an up-to-date version (pull from the remote "main" branch)
- Isolate your work in progress (create a separate branch, both local and remote)



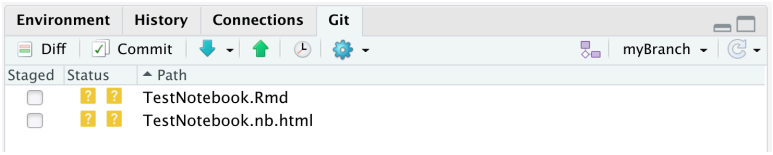
Go to GitKraken and have a look how this new branch shows up for now (note that your group's repo is likely to show a longer history already):

- Under "LOCAL", there is a new branch "myBranch"
- Same under "REMOTE", "origin"
- In the visual repo history, the symbols for "LOCAL" (laptop icon) and for "origin" (some other avatar) are at the same level. This indicates that your local branch is in sync with the remote branch "myBranch".

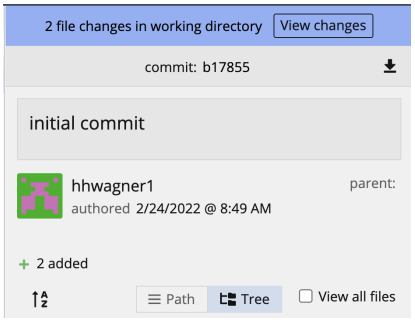


- Use the simple workflow as you go (pull from remote branch, commit, push to remote branch)
    - Create a new R Notebook, save it, don't commit yet. Then go to GitKraken to see what that looks like.

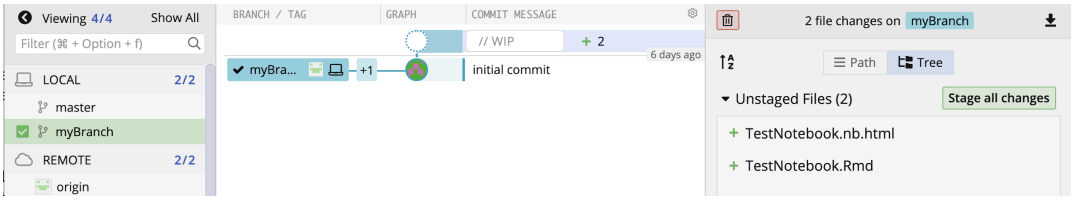
      In the git tab in RStudio:



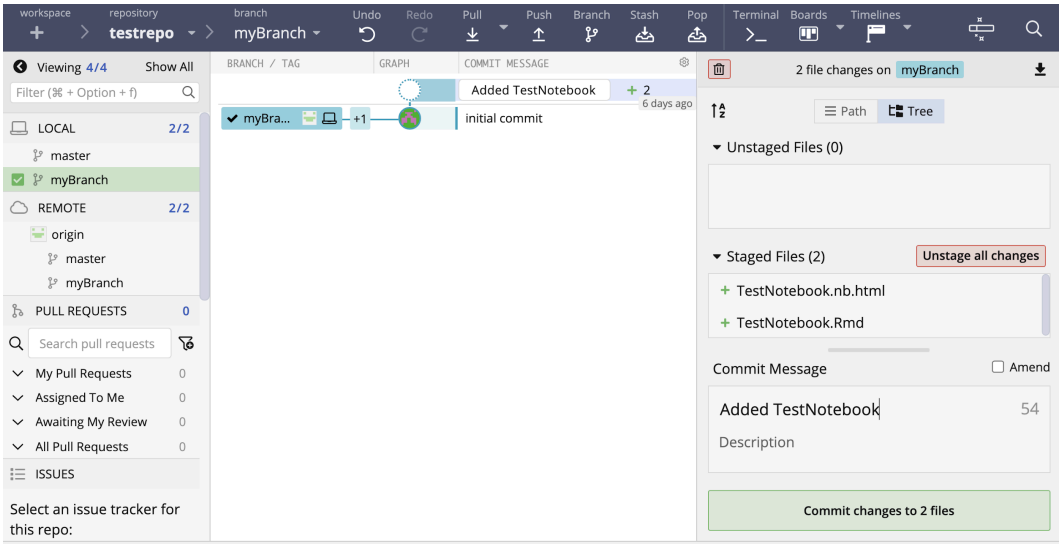
      In GitKraken, right-hand side:



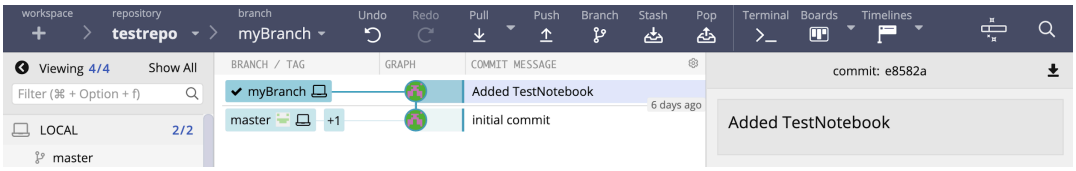
      Click on "View changes" to get this view: the new files are listed as "Unstaged"



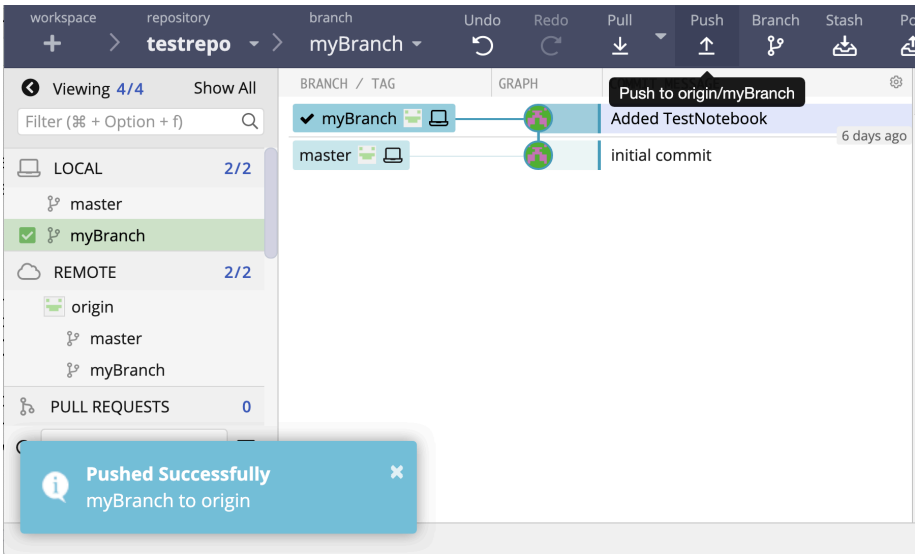
- You can now either click on "Stage all changes" here in GitKraken, add a commit message below, and commit (as shown below), or commit in RStudio.



- Note in the screenshot below how a new line has been added in the visual repo history. Also note how "LOCAL" is now ahead of "origin":

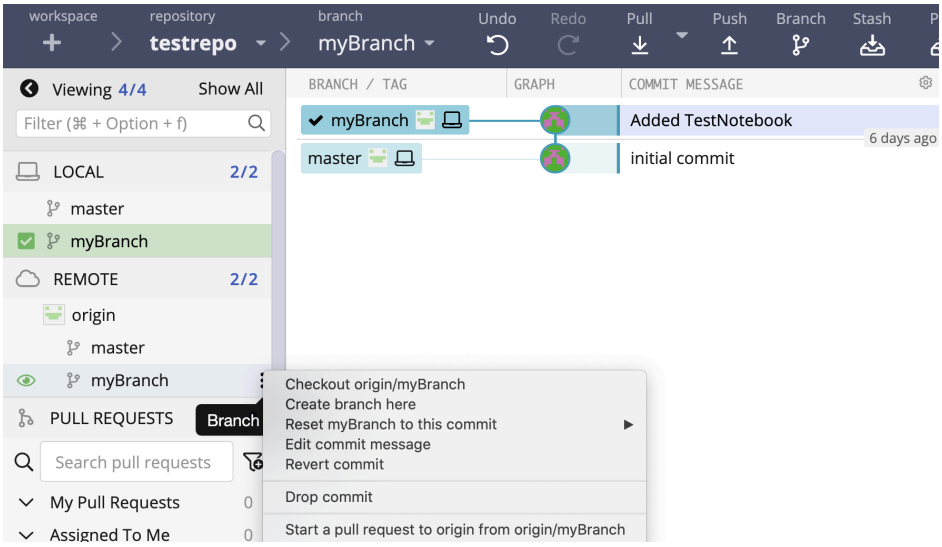


- Don't forget to push to the remote. This will bring "origin" of "myBranch" to the same stage as "LOCAL". You'll get confirmation that the push has been successful, and the remote branch is now in sync with the local branch. However, the "main" or "master" branches, both "LOCAL" and "REMOTE", are lagging behind.



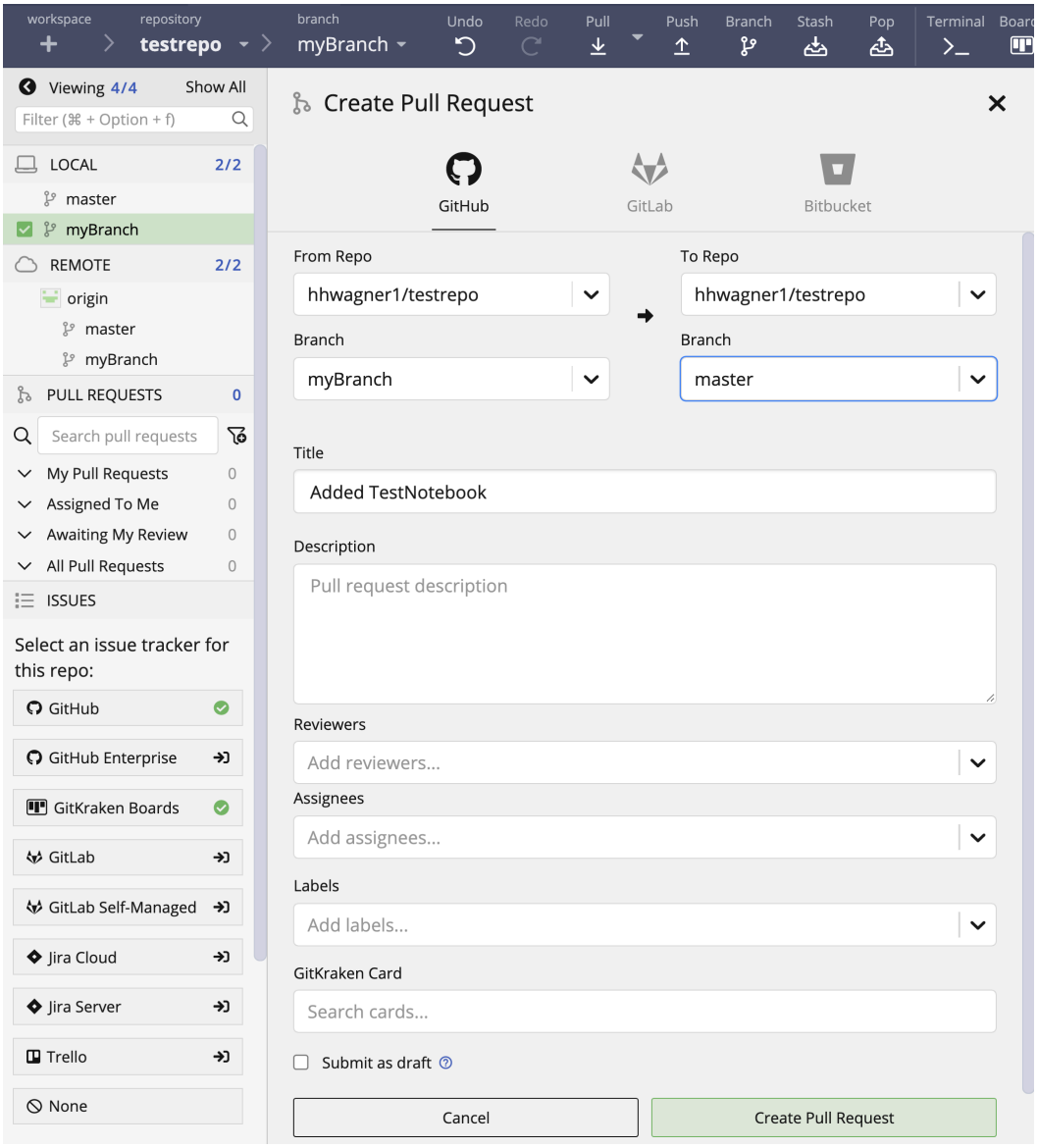
- When you are satisfied with the changes, ask your team to review them (create a pull request)
    - Start the pull request in GitKraken. With a pull request, you ask for the changes from the branch to be merged into another branch, usually "main"/"master".

      Hover on either the local or the remote "myBranch" to see three vertical dots. Hover over the dots and click (or right-click) to get the context menu. Select "Start a pull request".

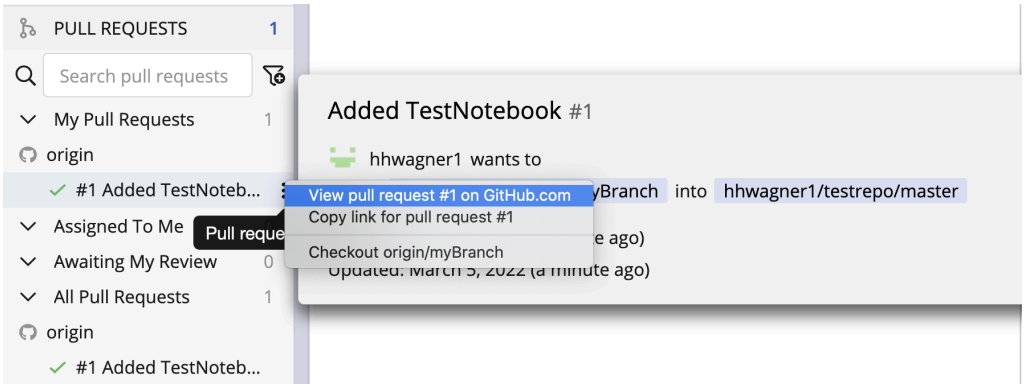


- Under "To Repo", "Branch" (on the right), select the "main"/"master" branch. Your last commit message will appear as Title.

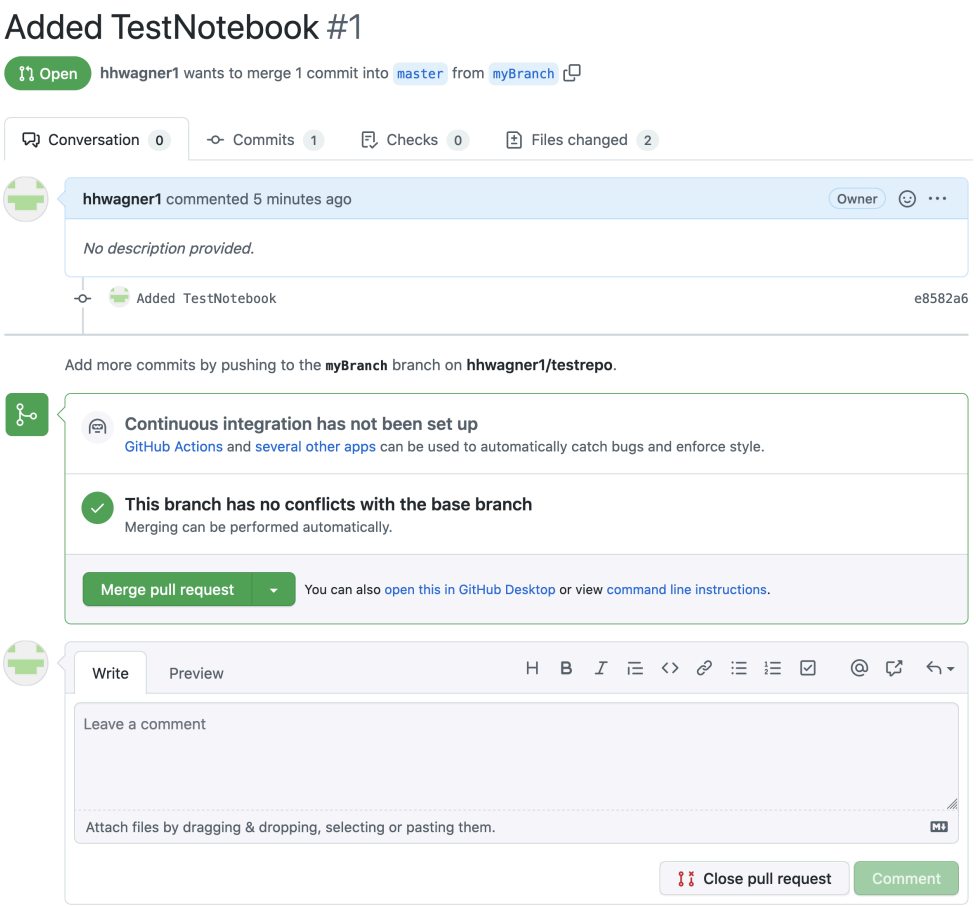
  You get options to assign collaborators as Reviewers. For now, just click "Create Pull Request".



- At this point, I like to go to GitHub to do the next steps. You can do so by responding to the success message (though that will be gone after a few seconds), or by using the context menu for the specific pull request (here: "#1 Added TestNotebook") and selecting "View pull request #1 on GitHub.com".



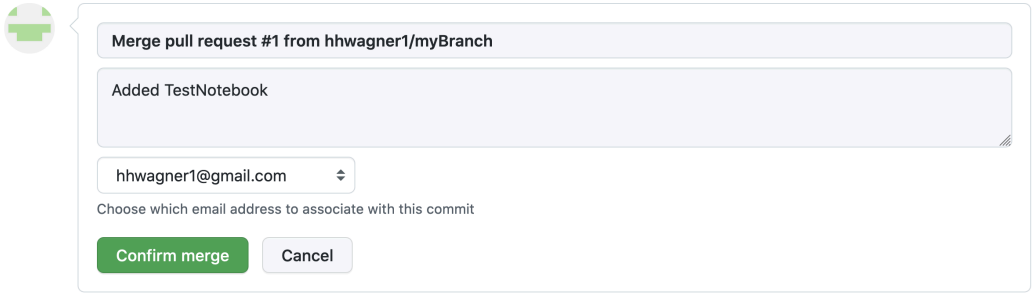
- If all is ok, update the "main" branch with your changes (merge branch into "main").
  - Ideally, on GitHub, it will look similar to this:



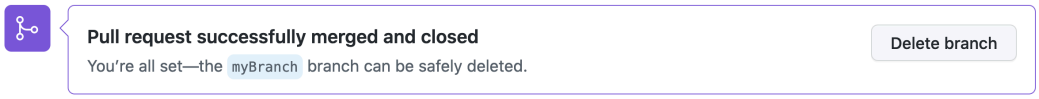
- If you get the message "This branch has no conflicts with the base branch", you can go ahead and click on "Merge pull request".

  You can leave comments for your team to explain in more detail what changes you made. Best discuss among your team the expectations and workflows with regard to alerting each other to changes. This may involve using git features (review, comments), sending a Slack message, or updating a Trello card, whatever works best for your group.

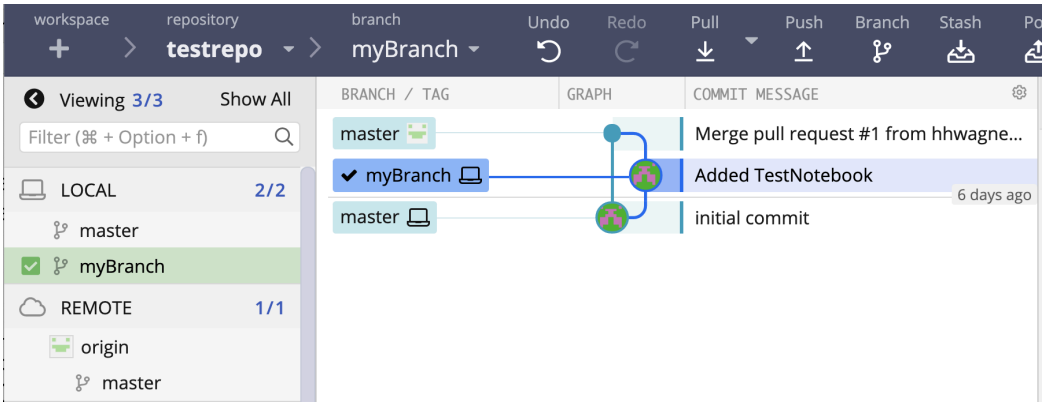
- After you clicked on "Merge pull request", you will be asked to confirm:



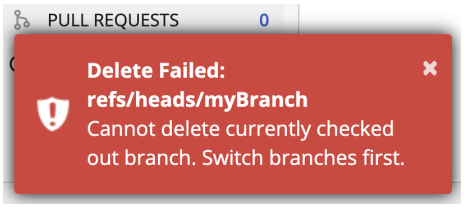
- With the confirmation, you get the option to delete the (remote) branch. Accept it (unless you have other, uncommited changes in it).



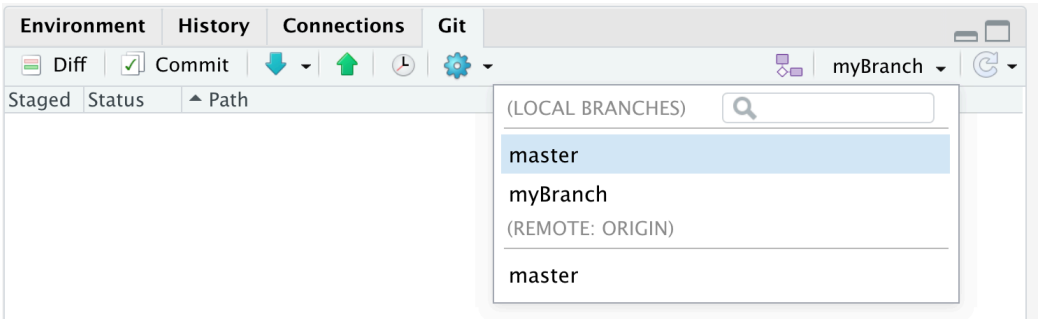
- Head over to GitKraken and check out the visual repo history. The remote branch has been merged into "main"/"master", but the local branch not yet.



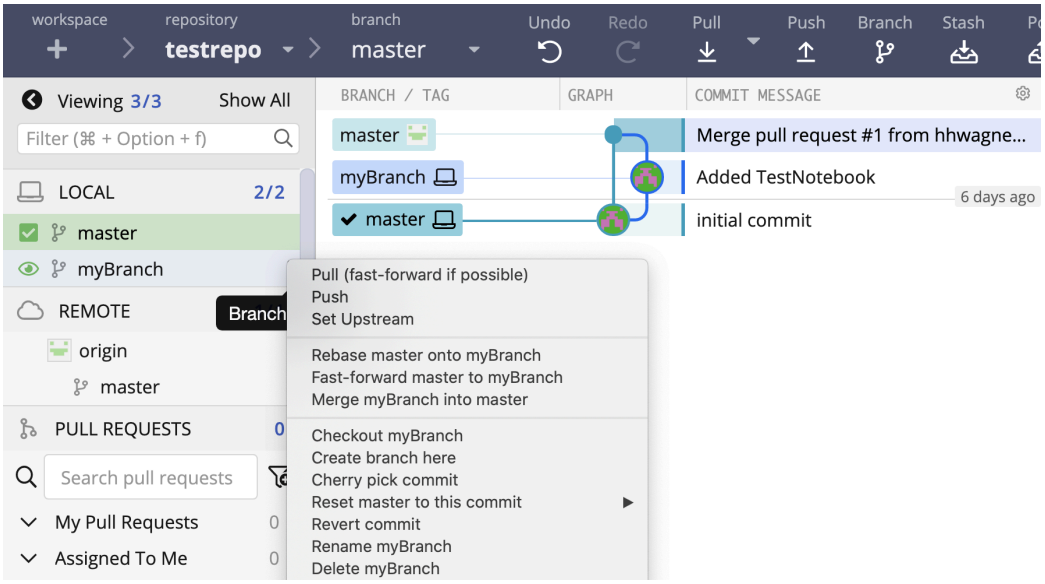
- If you now use the context menu of the "LOCAL" branch "myBranch" to delete it, you'll get an error message:



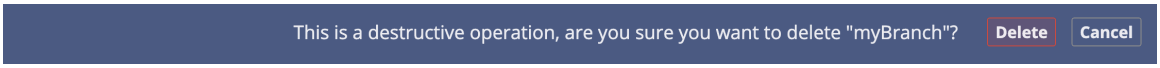
- In that case, go to RStudio and use the drop-down menu on the right-hand side of the git tab to switch branches to "main"/"master".



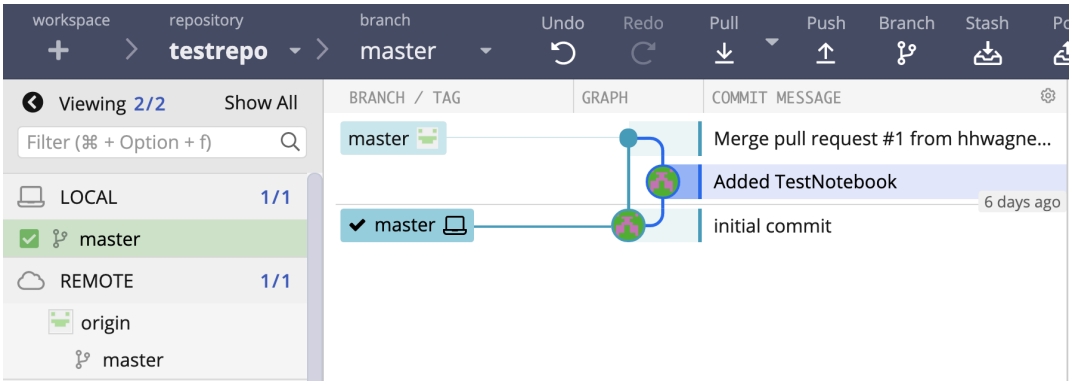
- Return to GitKraken and use the context menu of the "LOCAL" branch "myBranch" to delete it.



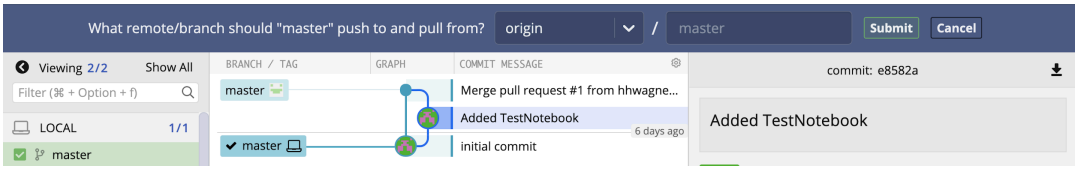
- You'll get a warning that this is a destructive operation. Click on "Delete".



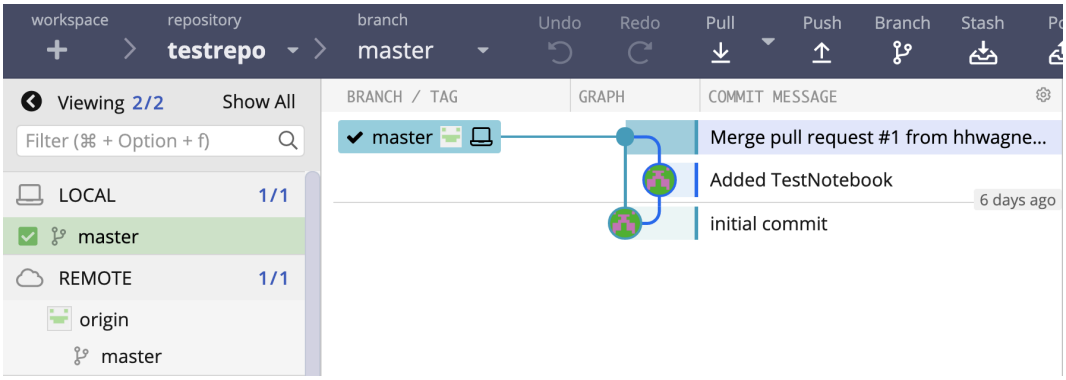
- In case you get an error message, refresh GitKraken and the branch shoudl be gone.



- The "LOCAL" branch "main"/"master" still lags behind! Pull to update it. Click "Submit".



- All's good. You can start the process again by creating a new branch, you could even use the same branch name again.



5. Notes on Portability

Any R Notebook that you merge into "main'' should be a safe, debugged version of code that can be knitted. "Knitting" is the process of rendering an R notebook. If you can knit it, this means that the code can stand alone and is portable, without referring to R objects that are held in memory during your local R session.

Consider this scenario:

- You defined the objects "a = 3" and "b = 2" in your Console, but not in the R Notebook.
- The R Notebook contains the code chunk "a + b" (but no definitions of "a" or "b").
- If you copy-paste "a + b" to the Console, it will run (and return 5).
- If you execute the code chunk with "a + b", it will run (and return 5).
- If you knit the R Notebook, it will return an error, it won't knit.

This may seem annoying but it is actually a helpful feature! Knitting your R Notebook before merging ensures that the code is portable, so that team members can run it as well in their R sessions.

6. Avoiding Redundancy

An additional strategy to increase portability is to write R objects to files (e.g., as ".rds" files). This is especially useful for R objects that contain the results of an important analysis step (e.g., a genind object with imported genetic data)

7. Learn about some advanced git concepts

Recommended videos:

- What is a merge conflict?
- Merge vs rebase
- What is stashing?
- What is git cherry picking?

Next steps:

- Check out the next tutorial “Surviving your first merge conflicts”.