Git Best Practices for Team Collaboration

# Coding

## Begin coding for a project

1. Open the repo: cd REPONAME (e.g., cd budget-app)
2. Ensure you are on the main branch: git checkout main
3. Ensure you are up-to-date: git pull
4. Create a new branch for your task: git checkout -b BRANCHNAME (e.g., git checkout -b error-messages)
5. Open VS Code: code .
6. Make all your edits
7. Add all your updates: git add .
8. Commit your updates: git commit -m "[FILENAME] UPDATE" (e.g., git commit -m "[Signup] Fix error messages")
9. Set the upstream during the initial push of your updates: git push -u origin BRANCHNAME (e.g., git push -u origin error-messages)
10. Push directly on later updates (only works if you have already set an upstream): git push

## Continue coding after taking a break

1. Go to the branch you were working on git checkout BRANCHYOUWEREWORKINGON (e.g., git checkout test-signup)
2. Pull in any new code from the main branch: git pull origin main
3. Resolve any merge conflicts that may now be revealed
4. Proceed as above (i.e., pick up at step 6 in the “Begin Coding for a Project” section)

## Continue coding after submitting a pull request

1. Create a new branch: git checkout -b NEWBRANCHNAME (e.g., git checkout -b test-login)
2. Pull in your code from the branch you were just working on git merge OLDBRANCHNAME (e.g., git merge test-signup)
3. Pull in any new code from the main branch: git pull origin main
4. Resolve any merge conflicts that may now be revealed
5. Proceed as above (i.e., pick up from step 6 in the “Begin Coding for a Project” section)

# Testing

## Test your code before submitting (website project)

### Frontend

1. Run both the frontend and the backend: npm start (each one)
2. Go to the site in Chrome
3. Navigate to the section of the site related to your updates
4. Investigate page elements if necessary
5. Take screenshots of what you do to help others who will test your code to know they can replicate what you have done

### Backend

1. Run the backend: npm start
2. Open Postman
3. Test routes
4. Take screenshots of what you do to help others who will test your code to know they can replicate what you have done

## Test someone else's code before approving

1. Go to the main branch: git checkout main
2. Ensure your main branch is up-to-date: git pull
3. Checkout the branch with the code you need to test: git checkout SOMEONEELSESBRANCH (e.g., git checkout readme-edits)
4. Test everything indicated in the pull request

# 

# Pull Requests

# Submit a pull request

1. Pull in any new code from the remote repo: git pull origin main
2. Resolve any merge conflicts that may now be revealed
3. Add, commit, and push your latest code
4. Go to the main page of the remote repo on GitHub
5. Go to a list of the repo’s branches by clicking on the “branch” link to the right of the “main” dropdown above the list of all the repo’s top-level folders and files
6. Begin a new pull request by clicking on the “New pull request” button on the far right of the line with your branch’s name
7. Include a list of to-do items for someone else to use for testing in the body of the pull request (these should include all the tests that you have already done from the “Test Your Code Before Submitting” section)
8. Assign someone else to handle your pull request by clicking the sprocket image next to “Assignees” and selecting someone from the list
9. Finalize the pull request by clicking the green “Create pull request” button
10. If you pull request has conflicts, add notes about how to resolve them in the comments section of the pull request

## Approve pull requests

1. Go to the “Pull requests” tab for the repo on GitHub
2. Click on the specific pull request in the list
3. Check off tasks as you perform each test
4. Scroll down and click the green “Merge pull request” button once you have tested everything indicated in the pull request (if this button is grayed out, then you have a merge conflict; see below)
5. After merged, delete the branch

# 

# Conflicts

## Resolve your merge conflicts

1. Ensure you are on the branch with the merge conflict: git checkout CONFLICTEDBRANCH (e.g., git checkout budget-function)
2. Get all the conflicts from the main branch to display in your local repo’s conflicted branch: git pull origin main
3. Open VS Code (if not already open): code .
4. Resolve conflicts (search for the >>>>>, <<<<<, or ===== pipes to find conflicts; your final version must not include any of those)
5. Add all your updates: git add .
6. Commit your updates: git commit -m "[FILENAME] Resolve conflicts with WHATEVER ISSUE" (e.g., git commit -m "[App] Resolve conflicts with updateBudget function")
7. Push your updates: git push origin CONFLICTEDBRANCH (or just git push if branch’s upstream is already established, which it should be)

## Resolve others' merge conflicts

1. Go to the “Pull requests” tab for the repo on GitHub
2. Click on the specific pull request with the conflict in the list
3. Read the provided suggestions about the merge conflict in the comments
4. Consider setting up a call with whoever submitted the pull request and possibly other members of the team
5. Consider doing a driver-navigator setup to fix the problems with others’ inputs
6. Click the gray “Resolve conflicts” button
7. Resolve conflicts on each page (search for the >>>>>, <<<<<, or ===== pipes to find conflicts; your final version must not include any of those)
8. Click the gray “Mark as resolved” button after resolving conflicts on each page (you will need to do this separately for each page)
9. Finalize merge after resolving all the conflicts by clicking the green “Commit merge” button

# 

# View Old Work

## Check out old commits

1. Go to the main branch: git checkout main
2. Ensure your main branch is up-to-date: git pull
3. See a list of all the commits: git log
4. Go to a particular commit: git checkout STRINGOFRANDOMLETTERSANDNUMBERSASSOCIATEDWITHASPECIFICCOMMIT (e.g., git checkout 6ade91e6f0d15a83eg775f0b9907e6fc021fa0fa)
5. Create a new branch starting at this commit: git switch -c NEWBRANCHNAME (e.g., git switch -c updates)
6. Make any edits in this new branch, which will not include any code from later commits

## Undelete an old branch

1. Go to the “Pull requests” tab for the repo on GitHub
2. Click on the “Closed” link (next to the “Open” link, which is the default tab)
3. Click on a pull request from the branch you want to resurrect
4. Scroll down to the bottom of the pull request, which includes a line about deleting the branch, and click the “Restore branch” button on that line
5. Work with any code specific to this deleted branch (this is only useful if the deleted branch contains code that is not accessible in any other branch; this approach is only possible if the branch was deleted after completing a pull request)

# Suggestions

## Create useful commit messages

* Include the name of the file you edited in brackets: git commit -m "[FILE] MESSAGE" (e.g., git commit -m "[App] Change function to class")
* Include a short, imperative message after the brackets to describe what you did in the edits associated with this commit (so instead of Fixed the stuff with the thing or I resolved the conflict we were having with the error messages, do Fix error messages)
* If your updates affect multiple files or an entire folder, either put all files in the brackets (e.g., [Signup, Login]) or the folder name in the brackets (e.g., [src] or [components])
* Aim for precision and accuracy, both of which should make the commit log a useful resource in case we run into hiccups and need to backtrack
* In contrast, include detailed information when handling bugs: If you’re troubleshooting a bug, then the longer and more detailed the commit message, the better; do a commit before you attempt to resolve the bug (e.g., [Signup] Pull json error messages from backend), along with various commits while attempting to resolve the bug (e.g., [Signup] Try to drill into the error object handled by the catch block in the handleSubmit function), and then a final commit once the bug is resolved (e.g., [Signup] Display the backend error messages by using error.response.data.msg in the catch block of the handleSubmit function); this saves the entire debugging process in the commit log for posterity (presumably, if you run into a bug with a functionality on one page, the rest of the team will probably run into a similar bug with a similar functionality on another page, so having the entire history of fixing your bug in the commit log will make the rest of the team’s experience fixing their similar bugs much easier)

## Create useful pull requests

* Include a list of to-do items with file paths or URL paths for the assignee to follow to perform specific tests: - [ ] ONEFOLDER/ANOTHERFOLDER/FILE (e.g., - [ ] src/components/pages/Signup.dart)
* Include screenshots of what the assignee should see at those paths
* Include any additional notes to help the assignee replicate the tests that you have already done

## Perform these tasks frequently

* Pull updates from the main branch into your current branch: git pull origin main
* Resolve any merge conflicts that may result from the above

## Adjust frequency of pull requests

* During a workday, submitting multiple pull requests per day makes sense
* Over the weekend or on holiday, submitting multiple pull requests makes less sense
* Coordinate with whoever will be assigned your pull requests to ensure you are doing what will best help them to quickly approve your pull requests

## Get GitHub notifications

* Ensure your notifications are turned on for web and mobile devices
* Be on the lookout for notifications on GitHub (they will appear as blue dots on the notification bell image in the upper right corner of the page)
* Whenever anyone assigns you anything, you should receive a notification

## Create a backup branch

* When you plan to rewind, delete changes, or otherwise do something drastic, then create a new branch: git checkout -b BACKUPBRANCH (e.g., git checkout -b backup)
* Save the current version of the project in this branch, and push it to GitHub
* Access this later if your radical changes need to be discarded

## Never use hard or force options

* Discard all history and go back to the specified commit (thus moving the repo’s HEAD backwards): git reset --hard STRINGOFRANDOMLETTERSANDNUMBERSASSOCIATEDWITHASPECIFICCOMMIT (e.g., git reset --hard 6ade91e6f0d15a83eg775f0b9907e6fc021fa0fa)
* Force your local repo’s main branch to become the remote repo’s main branch (thus ignoring any conflicts or anyone else’s work and commits): git push --force

# Common Git Commands

## Clone

* Copy a remote repo to your local computer by using its URL: git clone https://github.com/USER/REMOTEREPONAME (e.g., git clone https://github.com/IMG/template-application)
* Name your local repo something different than the remote repo: git clone https://github.com/USER/REMOTEREPONAME LOCALREPONAME (e.g., git clone https://github.com/IMG/template-application)

## Checkout

* Switch from one branch to another: git checkout OTHERBRANCH (e.g., git checkout readme-edits)
* Create and switch to a new branch: git checkout -b NEWBRANCH (e.g., git checkout -b signup-edits)

## Add

* Add a file to staging area: git add FILENAME (e.g., git add index.html)
* Add all recently edited files to staging area: git add .

## Commit

* Record all currently staged changes as a new instance of the version history: git commit -m "[FILE] IMPERATIVE MESSAGE" (e.g., git commit -m "[Signup] Edit appearance")
* Record all currently staged changed, and make any recently unstaged changes suddenly become staged to lock them into version control, all without using a message: git commit -a

## Push

* Send locally committed changes to the remote repo (to main branch if on main branch or to specified branch if on that branch and an upstream for it is already established): git push
* Specify that you are sending from your local main branch to the remote’s main branch (achieves the same as above, but with more words to clarify its functionality): git push origin main
* Specifically send changes from your local branch to its corresponding remote branch: git push origin BRANCHNAME (e.g., git push origin error-messages)
* Set an upstream for a branch (after which, you can just push with git push): git push -u origin BRANCHNAME (e.g., git push -u origin error-messages)
* Send all recently changed local branches to the remote repo: git push --all origin

## Pull

* Fetch and merge changes from the remote directory to your local directory (if on main branch): git pull
* Specify that you are pulling from your remote main branch to whatever local branch you are on (achieves the same thing as above if in the main branch, but helps reveal merge conflicts if not in main branch): git pull origin main

## Merge

* Merge a specific branch’s history into the current branch: git merge OTHERBRANCH (e.g., git merge signup-edits)
* Back out of a merge: git merge --abort

## Status

* List all files yet to be staged: git status
* Show branch and tracking info along with the files to be staged: git status -b

## Branch

* List all local branches: git branch
* Create a new branch: git branch BRANCHNAME (e.g., git branch signup-edits)
* Delete a branch: git branch -d BRANCHNAME (e.g., git branch -d signup-edits)

## Log

* List the version history of the current branch (i.e., its commit history): git log
* List the version history of a particular file: git log --follow FILE (e.g., git log --follow index.html)
* List only a one-line version of the current branch’s version history: git log --oneline

## Stash

* Temporarily store all the modified tracked files: git stash save
* Restore the most recently stashed files: git stash pop
* List all stashed changed files: git stash list
* Discard most recently stashed changed files: git stash drop
* Stash all files, even the ones that aren’t staged, possibly to never touch again: git stash --include-untracked

## Delete

* Delete the file from the working directory, and stage the deletion: git rm FILE (e.g., git rm utils.dart)

## Clean

* Remove untracked files from the working tree (this will delete files): git clean -f

## Reset

* Unstage your files, but preserve their contents (this will undo an unpushed commit): git reset

## Switch

* Change from one branch to another, and move to the head of that branch (very similar to checking out a branch): git switch BRANCHNAME (e.g., git switch signup-edits)
* Create a new branch and switch to it (very similar to checking out a branch when you create it; necessary for working with old commits): git switch -c NEWBRANCHNAME (e.g., git switch -c updates)