# Git/GitHub Workflow Guide

Processes for using Git/GitHub when you code

## When you start work on a new user story or feature… Create a Feature Branch

| | | |
|---|---|---|
| 1 | Make sure that all file changes in your working directory are either discarded or saved, committed, and pushed. | |
| 2 | Project Board → Click the ··· on the user story → "Convert to issue" | |
| 3 | `git checkout main` | Ensure you are in the `main` branch |
| 4 | `git pull` | Fetch all changes from remote `main` branch and merge them into your local copy. |
| 5 | `git checkout -b my_feature` | Create a feature branch off the main branch. Replace "my_feature" with a name related to the user story. |
| 6 | `git push -u origin my_feature` | Share your feature branch to the remote (GitHub). |

## When you sit down to code…

| | | |
|---|---|---|
| 1 | Make sure that all file changes in your working directory are either discarded or saved, committed, and pushed. | |
| 2 | `git checkout my_feature` | Ensure you are in your feature branch. |
| 3 | `git pull` | Ensure that you have the latest changes from the remote. Resolve any ***merge conflicts*** – described on the next page. |
| 4 | Write code and get to a natural stopping point. | Ensure that your project builds without errors. |
| 5 | `git add .` | "Stage" all your working directory changes to be saved. |
| 6 | `git commit -m "A concise description of changes"` | Create a new version of your feature branch. |
| 7 | `git push` | Share your new commits to the remote. You can add and commit multiple versions before pushing. |

## When you are ready to merge your changes into **`main`**…

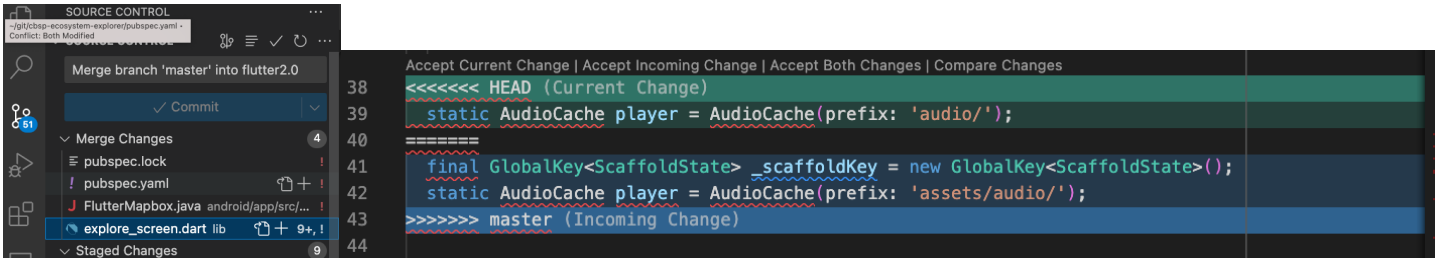
| | | |
|---|---|---|
| 1 | Make sure that all changes to your feature branch are committed. | |
| 2 | `git checkout main` | Switch to the `main` branch. |
| 3 | `git pull` | Ensure that your local `main` is up to date with the remote. Resolve any ***merge conflicts*** – described on the next page. |
| 4 | `git checkout my_feature` | Switch back to your `my_feature` branch |
| 5 | `git merge main` | Merge the changes from the `main` branch into the `my_feature` branch. This ensures that your feature incorporates new versions of `main`. Resolve any ***merge conflicts*** – described on the next page. Test your project to ensure that everything works. |
| 6 | `git add .` | "Stage" all changes you have made to resolve the merge. (This step may not be necessary and will have no effect) |
| 7 | `git commit -m "A concise description of changes"` | Create a new version of your feature branch. |
| 8 | `git push` | Send the merged version of your feature branch to the remote. |
| 9 | On GitHub, Pull requests → "New pull request" -> Change "Compare" drop-down to your feature branch. Click "Create pull request" and add comments that summarize the changes. Click "Create pull request" again to finalize. | |
| 10 | If your reviewer requests changes, go into your feature branch, make code changes, add, commit, and push to update. | |

## When you encounter a merge conflict…

You may get a ***merge conflict*** when you attempt to `pull` or `merge`. You should not have too many merge conflicts if you follow the processes on the other page. If you get a merge conflict because you did something wrong or weren't ready, run the command `git merge --abort` to try to revert to the pre-merged commit.

***Merge conflicts*** look something like this:

***Resolving*** merge conflicts is most easily done in IDEs like VSCode, XCode, PyCharm, or GitHub Desktop. In VSCode, for example, the Version Control tab will show a list of files that contain merge conflicts. Clicking on a file will show you the conflicting lines:




Git detected that these lines changed concurrently in both the `my_feature` and `master` branches. Git does not know how to resolve these changes – this is the conflict. When a merge conflict occurs, Git physically injects some text into your files:

- **YOURS**: The lines of code between `<<<<<<< HEAD (Current Change)` and `=======` are from your working `my_feature` branch.
- **THEIRS**: The lines of code between `=======` and `>>>>>>> master (Incoming Change)` are from the branch you are trying to merge into yours (master in this example).

A single file may have multiple regions of conflicting changes. Your code ***will not build*** until you remove these extra characters.

## Resolving the conflict

To resolve conflicts, you basically have three options:

1) "Accept the current change" – this deletes THEIR lines of code.
2) "Accept the incoming change" – this deletes YOUR lines of code.
3) "Accept both" – This leaves both YOUR and THEIR lines of code. You must then manually edit the file so that the new code is syntactically and semantically correct. That is, that it makes sense and works as intended.

Which option do you choose? You need to use your brain, and it is *always* a good idea to talk to your teammates.

**You must build and test your program once you have resolved all conflicts.** There is no guarantee that your merged code works as intended. Maybe the incoming changes introduce a bug, or your algorithms make assumptions that are incompatible with a change your teammate made. **Always test before you push!**

`add` and `commit` your changes once you are satisfied with your merge conflict resolutions. Then `push` if desired.