Overall Project Workflow

We describe here the steps to take for each session when you work on your project.

References

Pro Git chapter on contributing[1]

Notes

This assumes a small team, in our case two people.

Initial Setup

- Decide on one person, we will call him person A, to be the "owner" of the main repository. The other person will be person B.
- Create such a repository on A's GitHub account. This is the "remote" repository that both of you will be using.
- In the GitHub project's page, go to the Settings tab on the right, then to Collaborators. Type in person's B name and add them as collaborator.
- Both team members, log into your accounts, and open up:
  - A web browser. Navigate the browser to person A's repository. Copy the url that appears in the right side under "HTTPS clone URL".
  - A terminal window. Choose a location where you want to store the "local" repositories, and navigate there. Use `mkdir folderName` when needed to create subfolders. Do NOT create a folder specifically to hold the project, cloning will do that. Just go into the subfolder in which you want that project folder to be created.

    Now do `git clone thatURLYouCopied` to create a local clone of the repository.
- You are set!

Daily workflow

- Pair programming assumes you are both working on the same computer. Read more about pair programming here[2]. You will be carrying out all the following steps together.
- Choose which one of you will be the one typing for this session. Switch roles every session or every few hours.
- The person typing should log into their account.

[1] http://git-scm.com/book/en/v2/Distributed-Git-Contributing-to-a-Project
[2] http://en.wikipedia.org/wiki/Pair_programming

- Navigate in your terminal to the project folder.
- Do `git pull` to pull in any changes that were done in the previous session. Since it was the other person's computer last time, your repository does not have these changes yet; this command will update your system and bring it in line with the most recent changes.
- There should not be any merge conflicts, if there are then resolving them can get a bit tricky and require some more reading. But if you follow our workflow you should not have this problem.
- Back in GitHub, review the project's Issues page.
  - Are there issues that need to be addressed?
  - Should new issues be added? Add them, with appropriate labels and clear comments.
  - You can create your own labels as well if you don't like the ones that are there.
  - Do you have some new thoughts on an issue? Then leave a comment!
  - Reading through the totality of issues should give someone a good idea on where your project is.
  - Ideally every thing you do related to your project should have an issue associated with it. And when you make a commit you link that commit to a particular issue by mentioning the issue number (e.g. #1) in the message.
  - In both issues and commits, try to use imperative sentences or questions (well, questions only on the issues). e.g. "Add a test for empty arrays", "Should myAwesomeMethod treat 'null' specially?" and so on.
  - Keep your issues "small". Some times it makes sense to have a bigger issue, but to break it into smaller parts. See this article[3] for how to create checklists in your issue's comment.
  - Ideally an "issue" would be something you can work on in a single session, within a couple of hours.
  - Essentially the Issues page should contain everything in relation to your project. You shouldn't be keeping in your head things that need to be done or questions that need to be asked, turn them into issues. And this is something you can do at any time any place, you don't have to be with your partner to create new issues.
- Decide which issue you will work on addressing.
- Open up your favorite editor (for Sublime Text you can do "`subl .`" from the terminal to open the whole folder).
- Write tests for the issue you are about to address. (This will be a topic we will discuss later on)
  - Run them to make sure they fail.
  - Think about them to make sure they are testing all you will need this code to do!
  - If there are some maybe more extensive tests that you need but don't want to write right now, make an issue for them.

[3] https://github.com/blog/1375%0A-task-lists-in-gfm-issues-pulls-comments

- Add the code that would address the issue. (This line probably will take longer than its length here suggests)
  - As you work on your code, new issues might come up. Go back to the GitHub Issues page and add them!
- Run your tests to make sure that they have now been addressed.
- Run eslint to make sure your code follows the style standards. We will discuss this later.
  - Everything you commit for your project should be passing the tests and the lint process.
- Now you need to prepare commits.
  - The commits are essentially the history of your project, a journal of what happened when and by whom. Treat it with respect! It is what your coworkers will have to rely on to get up to speed with your project.
  - Do `git status` followed by `git diff` on individual files to see all the modifications you have done.
  - Decide if some modifications deserve to be a separate commit. Add only those modifications (using `git add`).
  - Some times you need to add only a part of a file. Use `git add --patch theFileName` to choose which sections of that file to commit.
  - Do a `git commit`.
    - Add your "commit message" at the top of the window that popped on your editor, above all the hashtagged lines.
    - Do NOT mess with any of the hashtagged lines.
    - The first line of your commit message should be short, no more than 50 characters, and should include any issue numbers addressed (possibly as "Close #5". It will often be very similar to the issue's title.
    - If you want to leave a longer explanation, add an empty line and your message after that. Use Markdown formatting rules. This is a good place to say in plain words what changes your code introduces.
    - Commits should be "**atomic**": A commit ideally should be doing one thing and one thing only.
  - Repeat as long as you have uncommitted modifications.
  - Do a `git log`, possibly `git log -p -2`, to see all the commits you just created and make sure they seem in order.
  - Do a `git status` to make sure you haven't left something out.
- Now you need to push your changes to the GitHub repository. Do `git push`.
- NEVER leave the project:
  - With local modifications that have not been committed.
  - Without doing a `git push` of your commits to the main repository.
- Back on your web browser, you should see the new commits there, as well as any issues you addressed having been closed.
- Congratulations! Take a short break and swap roles, then have at it on the next issue!