Some (hopefully) helpful tips for using git

Mateusz Krajewski

March 12, 2022

Contents

1 Introduction

It is of paramount importance that we use git in an organised and consistent manner. It will reduce confusion and make it easier to mitigate the consequences of our mistakes. And there will be plenty, inevitably. The following is my attempt to standardise our git workflow. Feel free to share this document and adjust it to Your needs – just remember to keep my name in. Any feedback will be appreciated.

In order to make the branch history as clean as possible, we have decided to create seven separate branches - one for every team member. Every developer is going to work on their issues on the branch assigned to them - and merge it with main when necessary.

I am going to present a couple of common scenarios that can arise when working with git. I tried to make it as relatable to the current project as possible. For that reason, all the examples below are based solely in Eclipse IDE. I may include some command-line examples later on.

2 The development process

The following is the general overview of our development process. You should try to adhere to it as much as possible. You will find more details later on in this document.

1. Start working on the issue on Your own branch.
2. Remember to commit and push regularly, so that other team members can review Your changes.
3. Your work is only done when Your code reviewer says so.
4. In order to merge, first make sure that Your local main branch is up-to-date with the remote one. In order to do that, *switch* to main branch and *pull* from the remote.
5. Then, *merge* Your branch into main. Solve any conflicts that may arise.
6. Close the issue in the issue tracker.

3 The basics

3.1 Git staging

There are two helpful views that You can enable in order to better understand what's going on. One of them is *Git staging*. You'll find it in:

`Window` → `Show view` → `Other` → `Git` → `Git Staging`.

It's an easy tool for reviewing the changes You have made, and ultimately committing them.

3.2 History

You can find *History* view under:

`Window` → `Show view` → `Other` → `Team` → `History`.

It shows the history of changes made to the chosen resource in the *Package Explorer*. Besides, it makes it extremely simple to see the state of current branches, both local and remote.

3.3 Current branch

The branch You're currently in is displayed right next to the project name in *Package Explorer* (Figure **??**). You almost certainly know this. But if I want to make this document as comprehensive as possible, it needs to be here.

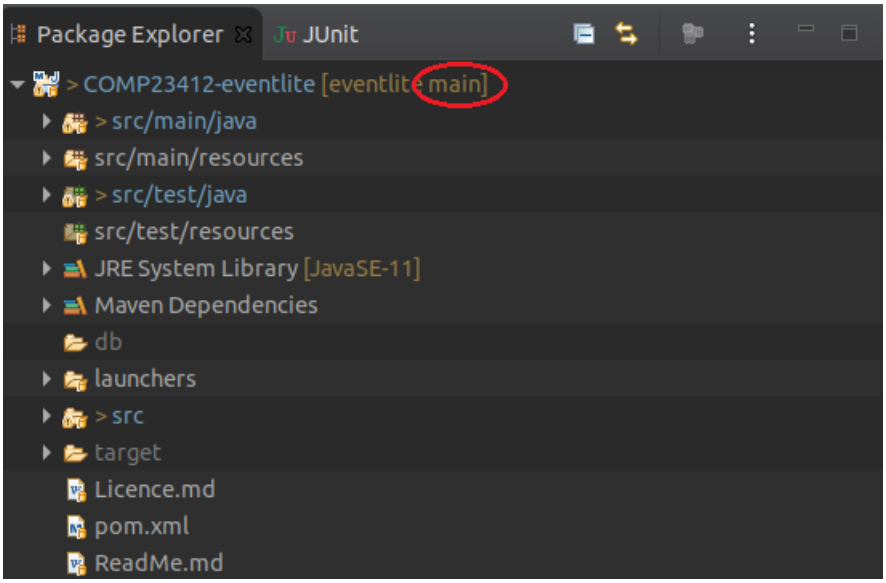

Figure 1: The location of the current branch name.

4 Switching

This is probably basic knowledge as well. In Eclipse, You can switch between branches/commits by right-clicking the project name in the *Package Explorer*

and choosing:

`Team` → `Switch to`.

With the help of this tool, it is possible to:

- create a new local branch;
- create a local copy of a remote branch;
- switch between local branches;
- switch between commits.

NOTE: Before You do any of that, make sure to look into *Git staging* and commit any changes You have made so far. Otherwise, git may behave unpredictably and You will spend the next two hours trying to figure out what to do (it certainly never happened to me, duh).

5 Updating the feature branch

Suppose You started working on Your branch and already made some commits. In the meantime, however, one of Your teammates merged their work into main and now Your branch is outdated. In such a case, the commit graph may look (roughly) like in Figure **??**. You may want to update Your branch and keep working on it, without having to merge. There is an excellent tool that can help You with that - it's called *Git Rebase*. After rebasing, Your commits will appear as if they were made after the teammate's merge. The results would be similar to those in Figure **??**.

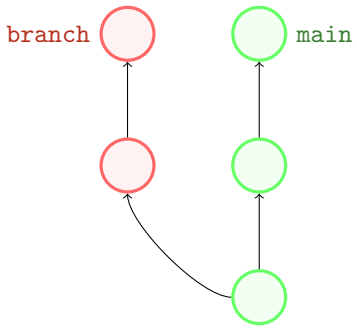

Figure 2: Before rebase.

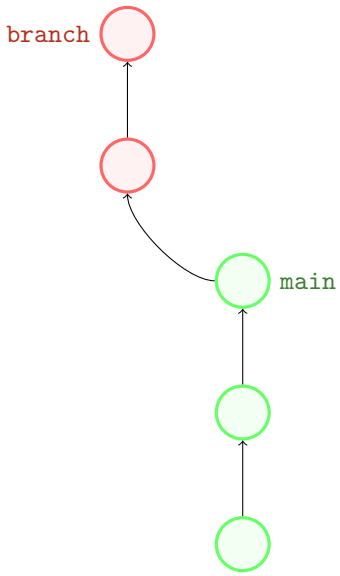

Figure 3: After rebase.

Now, how to do that? The process is quite simple. First, make sure that the main branch is synchronised with the remote. Then, **check out the feature branch**. Right-click the project name and choose the following:

`Team` → `Rebase`.

After that, select the local main branch. Boom. You're done.

5.1 Conflicts with the local main branch

If there are conflicts, however, things get a little more complicated. Most likely, a window like that in Figure **??** shows up. It displays the first of Your commits with conflicts detected, as well as the affected files. If You prefer to give Yourself some time to think, choose the *abort rebase* option. Otherwise, if You're feeling daring today and like living life on the edge, *start merge tool to resolve conflicts*.

Next, You will be taken to the *Merge Tool*, as shown in Figure **??**. There are three sections here:

- The upper section shows the files You need to fix.
- The bottom left section shows what the files look like in the last commit in main. This is where You apply Your changes.
- The bottom right section shows what the files look like in the first conflicting commit of Your branch. You can't change anything here.

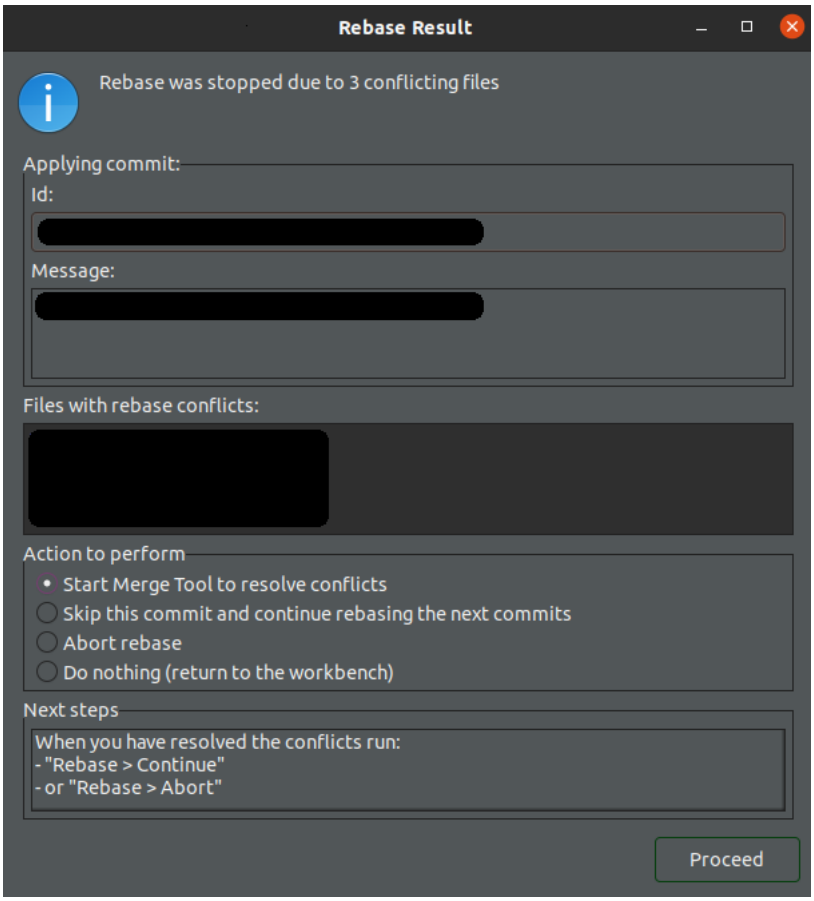

Figure 4: If there are conflicts during a rebase, this window will show up.

After You've made Your changes, stage them in the *Git Staging* view. Next, choose the *Continue* button, just like in Figure **??**. If any more of Your commits are in conflict with *main*, You will have to repeat the procedure. Otherwise, the rebasing is done and You can keep working on Your branch (or merge it into main without any conflicts).

5.2 Conflicts with the remote branch

Imagine the following scenario: You pushed Your commits to the remote feature branch, tried to rebase locally, encountered conflicts, solved conflicts, and tried to push the branch again. **Shucks**. You probably got a message like 'fast-forward merge rejected'. That's because when resolving the rebase conflicts, You modified Your commits, and now they're different from what's in the remote. If You're really certain of what You're doing, You can force git to

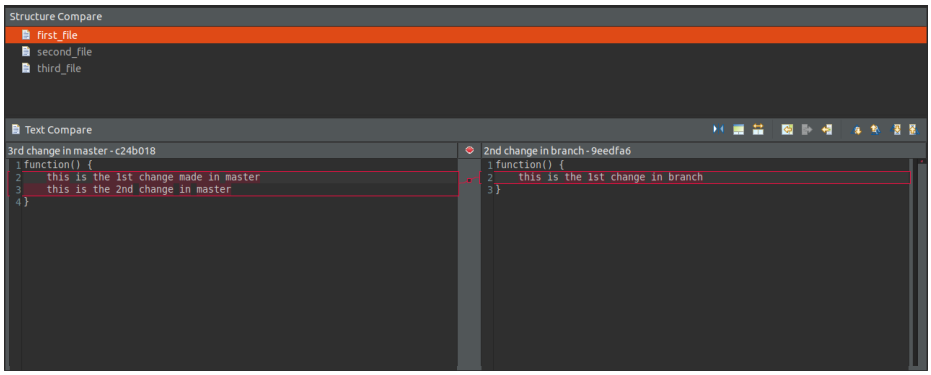

Figure 5: The Merge Tool shown if there are conflicts during a rebase.

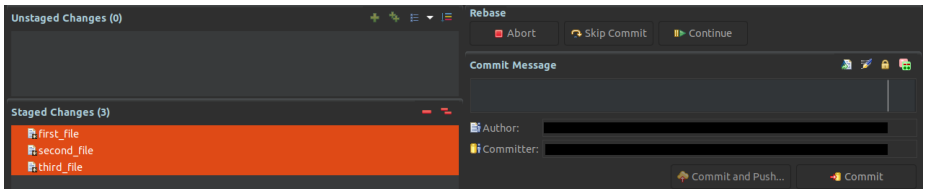

Figure 6: The *Git Staging* view shown if there are conflicts during a rebase.

overwrite the remote branch with Your current local one. In order to do that, right-click the project name and choose:

`Team` → `Push Branch` '`branch_name`'.

After that, the window similar to that in Figure **??** will show up. Tick the option to *force overwrite branch in remote if it exists and has diverged* and *push*. That's it. Now Your local branch is synchronised with the remote one.

6 Squashing

This is not necessarily useful, but git offers the option to 'squeeze' multiple consecutive commits into one. You may want to do it to simplify Your commit history. The process is simple: go to *History* view, ctrl-select the commits You want to squash and right-click them. After that, choose:

`Modify` → `Squash`.

If You pushed the commits before squashing them, You may be prohibited from pushing again. In such a case, refer to Subsection **??**.

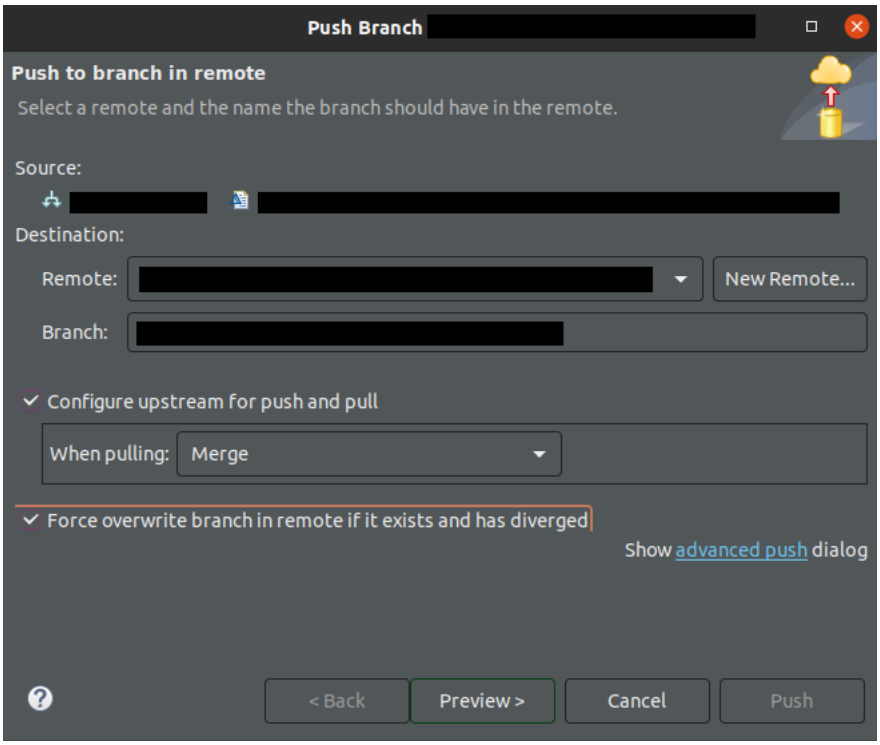

Figure 7: The window that shows up when You try to push a local branch.

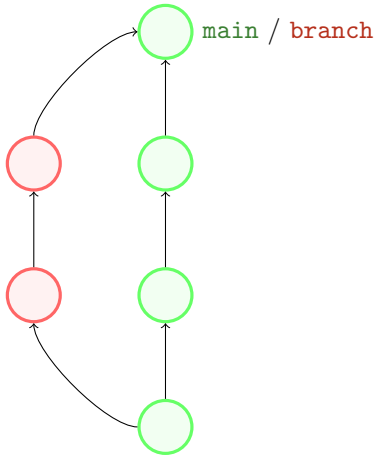

Figure 8: After merge.

7 Merging

Merging is really quite simple. **Check out the main branch**, right-click the project name and choose

`Team` → `Merge`.

After that, select the branch You want to merge into main. No need to meddle with the additional options. Now, if we imagine a commit graph like that in Figure **??**, its structure after the merge would be similar to that in Figure **??**. Note how git creates an additional merge commit, which incorporates the changes made both in the main branch and in the feature branch.

7.1 Conflicts

If the attempt to merge results in conflicts, it might be worth looking into the *Git Staging* view. The *Staged Changes* window will contain all successfully merged files. The *Unstaged Changes*, in turn, will contain all the files that'll have to be fixed. You can do it either with the *Merge Tool* (refer to section **??**) or edit every file by Yourself. In the latter case, every conflict is of the form:

```
<<<<<<< HEAD
what you see in the main branch
=======
what you see in the feature branch
>>>>>>>
```

You can choose one of the two options, merge them, or do anything else, really. Just remember to remove the redundant symbols - git won't do it for You! Once You're done, stage and commit Your changes.

8 Git commit messages

I know it's hard, but it is very important to try your best to commit in a standardised and consistent way. Some reasons why:

- You'll be able to track what you are doing better; when you come to work on the project two weeks later a commit message titled *Add search button* will be far more helpful than a commit message titled *Commit 5, I solved the issue.*
- When your team members or other people go to work on the repository, they will be able to scan through what you have done and work on your code.

How to do it:

1. Start with an imperative form of the verb i.e. *add, fix, delete*;
2. Start with a capital letter;
3. Keep it short;
4. Provide the concise outline of what You did;
5. Don't end the commit message with a period.

9 Acknowledgements