SENG202 – Software Engineering Project Workshop

2020

Tutorial 6 – Advanced Version Control with Git

1. Overview

The aim of this tutorial is to increase your awareness and understanding of some of the more advanced features and concepts within Git. Further, we also explore dealing with merge conflicts and provide an overview on different branching strategies.

Objectives:

- Understand advanced Git operations
- Complete the advanced Git quiz on the Quiz Server. The quiz will open for the last half of the tutorial.

2. Recap

In Tutorial 2, we learned about creating Git repositories, commits, pushing and pulling – recall that a commit simply encapsulates changes to the repository.

3. Merging

By now you will have had to deal with merges, and they likely haven't always gone smoothly. When merging changes from one branch (source) to another (destination) there is always a chance of a conflict at the destination. A conflict often happens when the change history is conflicting between the source and destination branch. The merge conflict happens when the destination has been updated with a different change in the same location as the source change; this will become clear with an example later in the lab. First, we need to understand what happens when we perform a merge.

Merging is Git's way of putting a forked history back together again. This lets you take independent lines of development and integrate them into a single branch. Note that merging is not exclusive to different branches – quite likely you have already dealt with merges in the same branch (e.g. when you pull from the repo, but have unstaged changes – these have to be merged back in).

When merging happens automatically

As with the previous example, the merge process is invoked when new changes to your repo are pulled, that have not been integrated with your own code before. If you **have** committed and pushed your local changes, later when you pull (provided you have no unstaged changes), no merge will take place.

Note the wording: "merge process is *invoked*". Often a merge will happen seamlessly (such as when developers are working on distinct features), other times, there will be conflicts that have to be manually merged together – more on this later.

This of course assumes that merges are happening on the same branch, such as when multiple developers commit straight to the master branch.

The four scenarios of a merge conflict

Merging a change into a local branch can be referred to as an 'incoming change'. Alternatively, when the local branch is the source making a change to another branch it can be referred to as an 'outgoing change'. Note the difference refers to the state of the local branch. Are the changes because of the local branch or effecting the local branch?

Now we know how to classify deferent changes the four different merge scenarios are as follows:

1. Merge incoming changes from remote branch to local branch.
2. Merge outgoing changes from local branch to remote branch.
3. Merge changes in one local branch to another local branch.
4. Merge changes in one remote branch to another remote branch.

Helpful resource explaining this in more detail: https://rollout.io/blog/merge-conflict-everything-need-know/

How to merge

Before you merge one branch into another, you should pull any new changes, and switch to the head of the branch you are merging to (often this will be the master branch or some release branch).

Merging can be initiated simply by running the following command (or by using the tools available in your IDE of choice):

```
git merge <branch>
```

Where <branch> is the name of the branch that you are merging *from*.

If there are no merge conflicts, then the merge will complete successfully, otherwise you will need to manually resolve these conflicts – more on this soon.

What happens during a merge?

Git has two distinct methods for merging: Fast-Forward merge, and 3-way merge.

Fast-Forward Merge

This is the simplest of the merge types in Git, and occurs when there is a linear path from the current branch head, to the target branch head. Consider the example in Figure 1 below, it can be seen that the feature branch is simply two commits ahead of the master branch – importantly, there are no commits made to the master branch, that are not already on the feature branch.

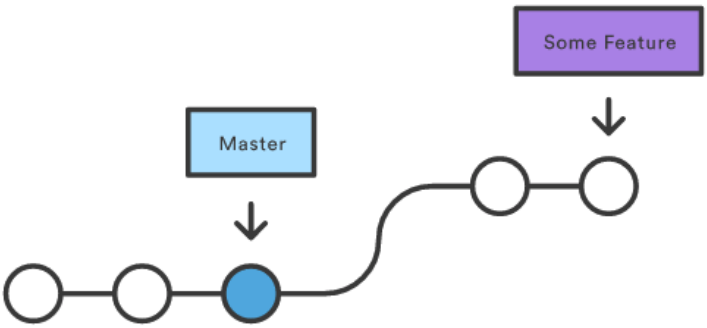

Figure 1. Repository before merging

When the feature branch is merged into the master branch, all that really happens, is the head of master is moved forward to be the current head of the feature branch, as shown in Figure 2.

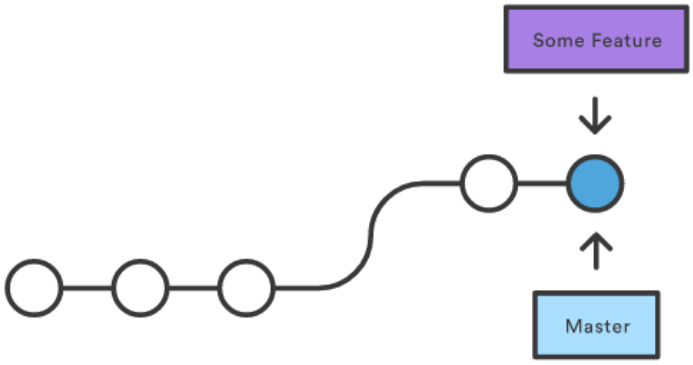

Figure 2. Repository after performing a fast-forward merge

3-way Merge

Often a fast-forward merge is not possible, such as when the branches have diverged, and one branch is not simply X commits ahead of the other. An example of this is shown in Figure 3, where a commit has been made to master, *after,* the feature branch has branched off master. Clearly, we can not apply the fast-forward merge tactic and set the head of master to be the head of the feature branch.

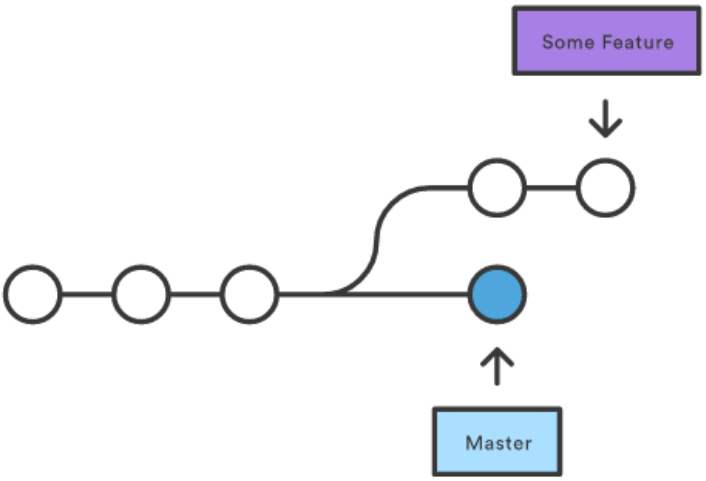

Figure 3. State of repository where a 3-way merge is necessary

When there is not a linear path to the target branch, Git has no choice but to combine them via a 3-way merge. An important distinction with a 3-way merge is that it generates a dedicated commit to tie together the two histories. When the merge is completed (after resolving any merge conflicts, if applicable) the state of the repository will look similar to that of Figure 4.

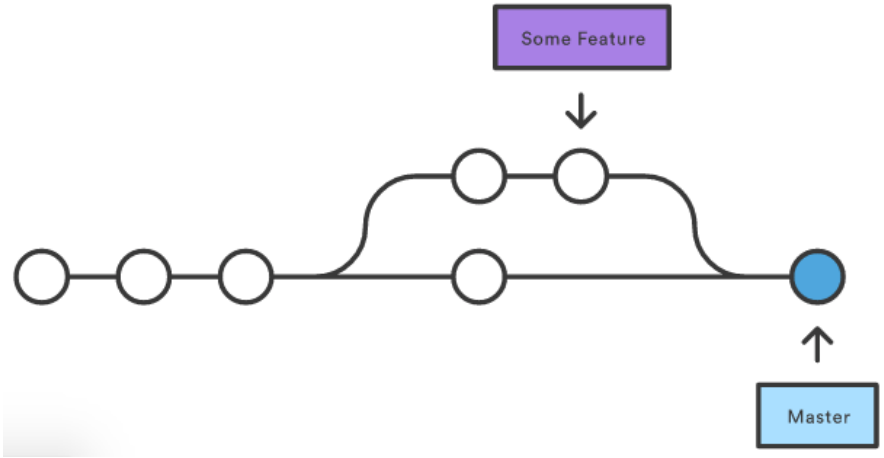

Figure 4. Repository after performing a 3-way merge

Merge conflicts

Merge conflicts occur if you are trying to merge the same file, and changes occurred in the same location in that file. Merge conflicts occur because Git cannot figure out which version to use – most often you will want both! In such a situation, Git stops automatically merging, so that the conflicts can be resolved manually.

Even if you both added a separate unrelated method to a class, if it was inserted at the same location, a merge conflict will occur – these conflicts are generally easy to resolve.

Merge conflicts occur exclusively with 3-way merges – it is not possible to have conflicting changes in a fast-forward merge!

Dealing with merge conflicts manually

These next two sections will demonstrate resolving a merge conflict manually.

You will need to initialise a new local git repository (refer to Tutorial 2 if you are unsure how), and issue the following commands, or follow the instructions:

```
# Create a new file "file1", and add "The number of planets are:"
git commit . -m "Added file template"

git checkout -b feature
# Edit the file, and add a new line "9"
git commit . -m "Added value"

git checkout master
# Edit the file, and add a new line "8"
git commit . -m "Added up-to-date value"
```

```
git merge feature
```

When you attempt to merge a branch, and there is a conflict, you will see a message similar to the following:

```
Auto-merging file1
CONFLICT (content): Merge conflict in file1
Automatic merge failed; fix conflicts and then commit the result.
```

Verifying with "git status", it shows that we need to fix the merge conflicts, and run "git commit".

```
$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")
```

Open up the file "file1" in your text editor of choice and inspect the contents of this file, it should look similar to that shown below. (If the file is already open you will need to refresh or re-open the file)

```
The number of planets are:
<<<<<<< HEAD
8
=======
9
>>>>>>> feature
```

This is Git's syntax for showing merge conflicts. A conflict-marked area begins with "<<<<<<<" and ends with ">>>>>>>" – these are known as conflict markers. The two conflicting blocks are divided by "=======".

In plain English, what this file is saying is that the current checked-out head (master branch) wants to add the line "8", but the feature branch wants to add the line "9" – both in the same place. Git can not simply add both of these lines automatically (order matters), so the developer needs to resolve the merge conflict.

Resolving the merge conflict is conceptually trivial – this file is fully editable by the developer and should be edited until they are happy with the outcome. The conflict marker notation is simply there for the benefit of tools that allow for merging (e.g. the graphical merge tool in your IDE).

Edit the file to your liking – note that you can add anything you want here, you're not limited to the options of "9" or "8". Just make sure to remove all conflict markers and dividers.

When finished, you can add file and commit this file as per usual. (git add file1)

Dealing with merge conflicts graphically

We have provided a repository that is currently in a state that requires merge conflict resolution. This example is fairly straight forward and you are not required to attempt it yourselves, however, it is important that you understand the steps that are being performed. For those of you that wish to follow along, the resources are located on Learn (google-maps-api.zip). Download and import the project into IntelliJ.

IntelliJ provides a merge conflict resolution tool to locally resolve conflicts in a user friendly way. This tool consists of three panes. The left pane shows the read-only local copy; the right pane shows the read-only version checked in to the repository. The central pane shows a fully-functional editor with the results of merging and conflict resolving.

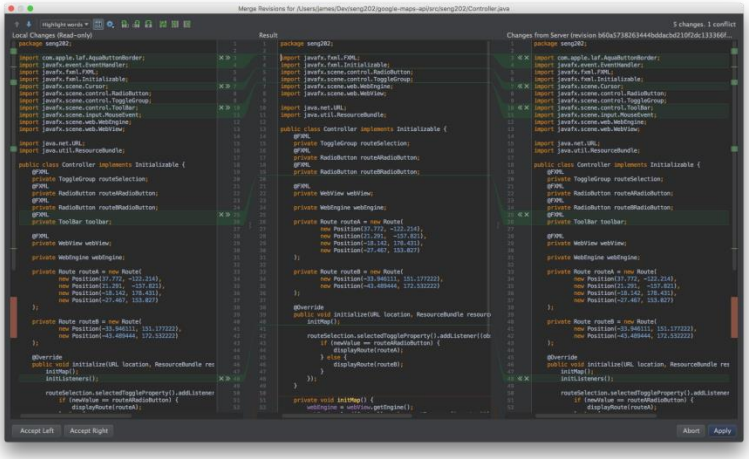

Figure 5. IntelliJ merge conflict resolution tool

In order to begin the merge conflict resolution, you will need to merge the feature branch into master. At the bottom right corner of IntelliJ, click “Git: Master”, place the cursor over “feature” and select “Merge into Current” (Figure 6). This will show a dialog listing which files have conflicts needing resolving. Click the “Merge” button and begin the conflict resolution process.

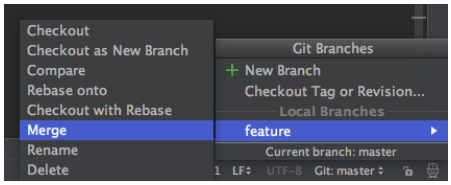

Figure 5. Merging

Begin by merging non-conflicting changes first, this allows you to focus on the conflicts and not get side-tracked by non-conflicting changes. This is easily accomplished by pressing the “Apply All Non-Conflicting Changes” button located in the top left of the tool (Figure 7).

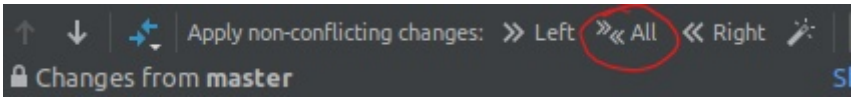

Figure 6. Apply All Non-Conflicting Changes button

In the left or right pane, select the changes to be merged. Accept or reject changes by clicking the chevron or X buttons respectively. In our case we want to keep the changes on the master branch and discard the feature. Therefore click the chevron from the left pane and the X for the right (Figure 8).

Figure 7. Chevron and X buttons

Once this is finished you can click the “Apply” button and finish.

It is important to note that the middle pane is editable and you can enter/delete text it contains and those changes will be propagated to the file once apply is pressed.

4. Code Reviews & Merge Requests

Merge requests (or pull requests) are a feature that makes it easier for developers to collaborate. They provide a user-friendly web interface for discussing proposed changes before integrating them into the official/main repository.

In their simplest form, merge requests are a mechanism for a developer to notify team members that they have completed a feature. Once their feature branch is ready, the developer creates a merge request (e.g. through the GitLab web interface) – the developer can then assign this merge request to another developer(s) for approval.

When you create a merge request, all you are effectively doing is requesting that another developer (e.g. the project maintainer) merges your branch into their repository. You will always need to specify the source branch (your branch) and the target branch (e.g. master), but sometimes you will also need to specify the repositories involved – as often in public contributions, your work will be on a separate/forked repository.

You may want to experiment with merge requests the next time you push a new feature to your project.

Note: Assigning merge requests to another team member is a valuable process to ensure the whole team is responsible for code ownership. If you are responsible for merging and releasing your own code you may lack the critical observation required to notice potential bugs being introduced within the new changes.

5. Cherry-Picking

Another interesting tool within Git is git-cherry-pick. Cherry picking allows a developer to apply the changes introduced in any arbitrary commit, to the current working directory.

Consider this example: You are working on your project and need a helper function to calculate GST, luckily one of your colleagues has already implemented this functionality - but it is on another branch that isn't ready to be merged in yet.

By finding the exact commit that introduced the GST function, we can apply the changes within that commit, to the working directory – regardless of which branch we are on.

There are many variations on the cherry-pick command, but the simplest form is to take a single commit based on the commit hash:

```
git cherry-pick <commit>
```

Depending on what changes were in the commit specified, everything may go smoothly, however there is also the possibility of a merge conflict – in that case, you will need to resolve this conflict as explained previously.

6. Branching Strategies

There are many different branching strategies and it is impossible to say which one is the best as it depends entirely on the development team or company decide what best matches their particular needs. Therefore, you should spend some time researching which strategy best suits your needs prior to beginning implementation. A good place to start is feature branching:

https://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow

https://rollout.io/blog/pitfalls-feature-branching/

7. References

The figures used in this example are taken from Atlassian's guide to merging (https://www.atlassian.com/git/tutorials/using-branches/git-merge), and are licensed under the Creative Commons.