Git part 3: Branching and partial commits

Michaël Van Canneyt

October 17, 2021

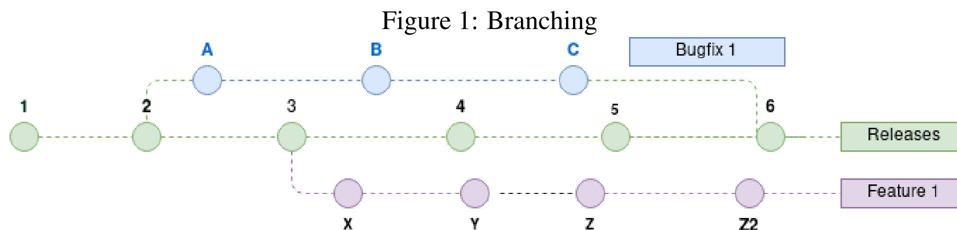
Abstract

Branches are used abundantly in git. In this article we'll explore how to use them and how to deal with a special git feature related to branches: rebase. We'll also see how to split commits in git, in case you did several unrelated things at the same time but wish to commit each thing separately anyway.

1 Introduction

Often, several avenues of development occur at the same time: people working on separate features or fixing issues simultaneously. To prevent these separate development efforts from interfering with each other, they are conducted in different branches of the repository. A branch is simply a series of commits which is kept separate from the 'main' series of commits from which the actual project releases are made. A possible branching situation is depicted in figure 1 on page 1. Each dot in the diagram is a commit, and is named. In reality these names, the so-called commit hashes, will be much longer.

Figure 1: Branching



The middle line (green commits 1..6) are the main branch: from this branch, releases are made. The bottom line (purple dots: commits X..Z2) are the commits from work on a feature development. The top line (blue dots: commits A-C, joining at commit 6) are from work on a bugfix.

When a line of development is finished (as for the bugfix in figure 1 on page 1), the work is reintegrated with the main development branch from which releases are made. This is depicted by commit 6: the work in commits A-C is merged to the main development line, and in the above diagram, this is done through a new commit. This new commit is not a requirement, but more about this later.

Branching is one way to keep various lines of development separate. This is a conscious planning step.

However, imagine you're working on some code: a feature, a bugfix, this does not really matter. You are working, you have changed one or more files, and while working you notice a bug that you quickly fix. When it is time to commit your work, you decide that this fix needs to be committed separately, not together with the work you were originally

intending to do. How to separate this fix from the other changes? This is where a partial commit comes in. This is a simple case, which we'll treat first.

2 Partial commit

Imagine you need to change the following unit, you need to add a second hash function to it:

```
unit myunit;
{$mode objfpc}
interface

Function MyHash(aString : String) : Longint;

implementation

Function MyHash(aString : String) : Longint;

begin
  Result:=Length(aString) div 2;
end;

end.
```

You start working on it and at some point you notice that you should have divided by 3 in `MyHash`, so you change it. After you finish the `MyBetterHash` function, the end result of your work is:

```
unit myunit;
{$mode objfpc}
interface

Function MyHash(aString : String) : Longint;
Function MyBetterHash(aString : String) : Longint;

implementation

Function MyHash(aString : String) : Longint;

begin
  Result:=Length(aString) div 2;
  Result:=Length(aString) div 3;
end;

Function MyBetterHash(aString : String) : Longint;

Var
  C : Char;

begin
  Result:=0;
  For C in aString do
    Result:=Result xor Ord(C);
```

```
end;

end.
```

Now it is time to commit. Obviously, you wish to keep the bugfix separate from the new feature.

As explained in a previous article, the `add` command in Git adds a file to the next commit.

This command has an option `-p` or `-patch`, which tells Git to analyse the changes in the file and ask you for each change whether you want to add it to the commit or not.

```
git add --patch myunit.pp
```

This will result in the following output:

```
diff --git a/myunit.pp b/myunit.pp
index bda87cf..e16c96d 100644
--- a/myunit.pp
+++ b/myunit.pp
@@ -3,13 +3,26 @@ unit myunit;
 interface

 Function MyHash(aString : String) : Longint;
+Function MyBetterHash(aString : String) : Longint;

 implementation

 Function MyHash(aString : String) : Longint;

 begin
-  Result:=Length(aString) div 2;
+  Result:=Length(aString) div 3;
 end;

-end.
\ No newline at end of file
+Function MyBetterHash(aString : String) : Longint;
+
+Var
+  C : Char;
+
+begin
+  Result:=0;
+  For C in aString do
+    Result:=Result xor Ord(C);
+end;
+
+end.
+
(1/1) Stage this hunk [y,n,q,a,d,s,e,?]?
```

Clearly, this will add everything (the whole 'hunk') in one big commit. That's not what you want, obviously, so you can give the `'s'` command, which splits the commit into smaller commits.

Here are all the possible answers to the question Git is asking:

y Add this hunk to the next commit.

n Do not add this hunk to the next commit.

q quit; Do not add this hunk or any of the remaining hunks.

a Add this hunk and all later hunks in the file.

d Drop this hunk and all the later hunks in the file.

g Go to a particular hunk.

/ Search for a hunk: this command must be followed by a regular expression.

j Jump over this hunk (leaves it undecided) and show the next undecided hunk.

J Jump over this hunk (leaves it undecided) and show the next hunk.

k Leave this hunk undecided, show the previous undecided hunk.

K Leave this hunk undecided, and show the previous hunk.

s Split the current hunk into smaller hunks, and repeat the process.

e Edit the current hunk: you can replace +/- by # to skip parts of the hunk.

? Show a help message.

After choosing `'s'`, Git will respond with:

```
Split into 3 hunks.
@@ -3,9 +3,10 @@
 interface

 Function MyHash(aString : String) : Longint;
+Function MyBetterHash(aString : String) : Longint;

 implementation

 Function MyHash(aString : String) : Longint;

 begin
(1/3) Stage this hunk [y,n,q,a,d,j,J,g,/,e,?]?
```

You can see that this is the first change in the file. Also note that git now offers more possible answers than the first time it asked the question.

Depending on what you want: first commit the feature, then the fix or vice versa, you can now answer `'y'` or `'n'`, respectively. We'll first commit the fix, so we answer `'n'` because we don't need the new function declaration.

Git responds with:

```
@@ -6,9 +7,9 @@

 implementation

 Function MyHash(aString : String) : Longint;

 begin
```

```
-  Result:=Length(aString) div 2;
+  Result:=Length(aString) div 3;
 end;

(2/3) Stage this hunk [y,n,q,a,d,K,j,J,g,/,e,?]?
```

This is the part we want because it contains the fix we wish to commit first, so we answer `'y'`. Git will then ask us what we want to do with the last part:

```
@@ -13,3 +14,15 @@
 end;

-end.
\ No newline at end of file
+Function MyBetterHash(aString : String) : Longint;
+
+Var
+  C : Char;
+
+begin
+  Result:=0;
+  For C in aString do
+    Result:=Result xor Ord(C);
+end;
+
+end.
+
(3/3) Stage this hunk [y,n,q,a,d,K,g,/,e,?]?
```

Here we answer with `'n'`, because this is not part of the fix we want to commit.

After this, git is finished. When we ask for the status, we get the following:

```
> git  status
On branch main
Your branch is up-to-date with 'origin/main'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
modified:   myunit.pp

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
modified:   myunit.pp
```

As you can see, the `myunit.pp` file is listed twice:

- once as modified and ready to be committed.
- once as modified, but not yet staged for commit.

If we ask to see the changes in our file using the `diff` command, Git will respond with the following:

```
git diff myunit.pp
diff --git a/myunit.pp b/myunit.pp
index c6b8536..e16c96d 100644
--- a/myunit.pp
+++ b/myunit.pp
@@ -3,6 +3,7 @@ unit myunit;
interface

Function MyHash(aString : String) : Longint;
+Function MyBetterHash(aString : String) : Longint;

implementation

@@ -12,4 +13,16 @@ begin
  Result:=Length(aString) div 3;
end;

-end.
\ No newline at end of file
+Function MyBetterHash(aString : String) : Longint;
+
+Var
+  C : Char;
+
+begin
+  Result:=0;
+  For C in aString do
+    Result:=Result xor Ord(C);
+end;
+
+end.
```

As you can see, only the new functionality is present in the changes. The bugfix change is already in the 'staging area', the name Git uses for changes that have been scheduled for a commit.

A similar situation can arise when you change a file, and add it to the staging area, but then change the file some more, before committing.

We can now commit our bugfix change with the `commit` command:

```
> git commit -m '* Bugfix for MyHash: divide by 3 instead of 2'
[main 5b10ccb] * Bugfix for MyHash: divide by 3 instead of 2
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Note the commit hash `5b10ccb`. It can be used to check what was committed with the `diff` command of Git:

```
> git diff 5b10ccb^..5b10ccb
diff --git a/myunit.pp b/myunit.pp
index bda87cf..c6b8536 100644
--- a/myunit.pp
+++ b/myunit.pp
@@ -9,7 +9,7 @@ implementation
 Function MyHash(aString : String) : Longint;
```

```
 begin
-  Result:=Length(aString) div 2;
+  Result:=Length(aString) div 3;
 end;

 end.
```

The sequence `5b10ccb^..5b10ccb` in this command-line tells git to show the difference between the commit previous to commit `5b10ccb` (denoted by `5b10ccb^`) and commit `5b10ccb`.

As you can see, only our bugfix was committed. Now, the new feature can be committed directly:

```
> git commit -m '* Add MyBetterHash function' myunit.pp
[main 324ccf6] * Add MyBetterHash function
 1 file changed, 14 insertions(+), 1 deletion(-)
```

If you want to split the changes in multiple commits, the above interactive `add` command must be repeated for every partial commit you wish to make.

The same operation can be done in TortoiseGit, but it is implemented in a somewhat roundabout manner – presumably this is a heritage of the TortoiseSVN origins, because in Subversion, no native partial commit functionality is available.

- In the commit dialog of ToirtoiseGit, there is a context menu item called `Restore after commit` (see figure 2 on page 8). What this does is to create a backup of the file as it is, which will be restored after the commit has happened.
- Now you can open, from within the same commit dialog, the file with the diff & merge tool of TortoiseGit (see the `Compare with base` menu item in figure 2 on page 8).
- In this tool you can then indicate what changes you want to see included in the commit using the `Mark this block` menu item in TortoiseGit (see figure 3 on page 8).
- When you have marked all changed, you can discard all other changes with the `Leave only marked blocks` menu item (see again figure 3 on page 8).
- Now, after you save the file in the diff tool, you can commit the resulting file: see figure 4 on page 9, and note the counts of added and removed lines.
- After the commit, TortoiseGit will then restore the backup it made and as a result, all the remaining changes will again be present in your working copy.
- The file will still be marked changed in the explorer, and the remaining changes can also be committed. See figure 5 on page 10, and again note the counts for added and removed lines.

If you want to split the changes in multiple commits, the above procedure must be repeated for every partial commit you wish to make.

Figure 2: The ‘restore after commit’ menu item in the commit dialog

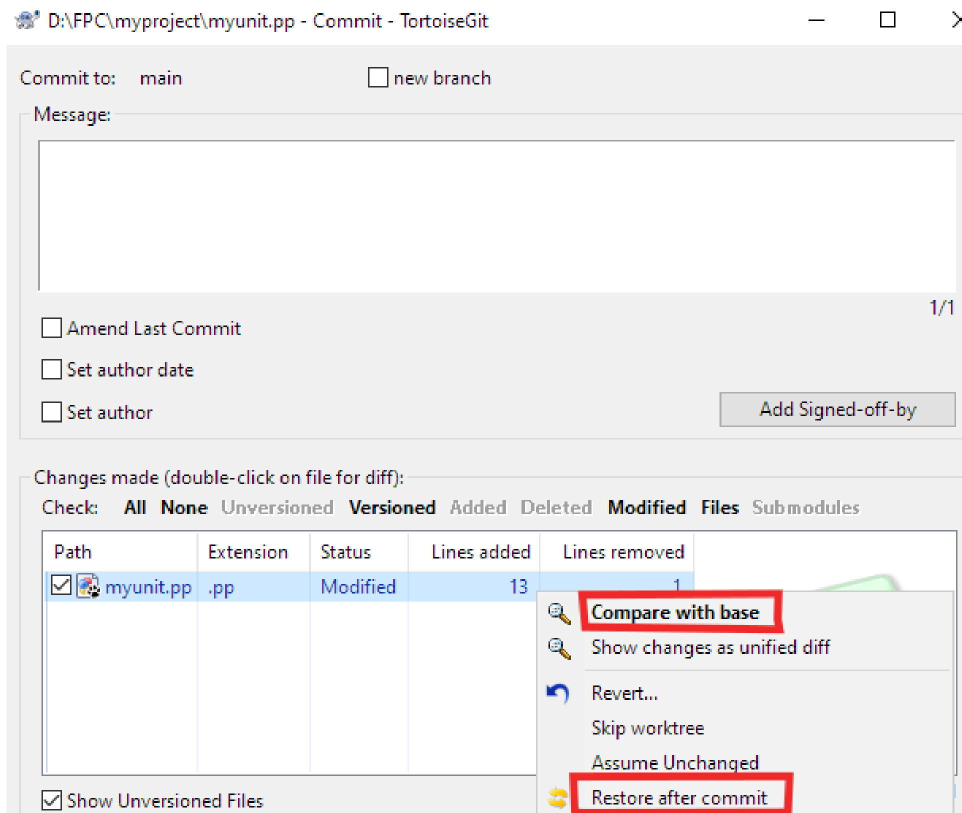
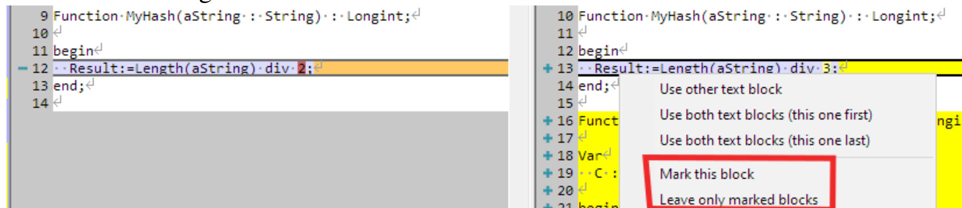


Figure 3: The ‘mark this block’ menu item in TortoiseDiff



3 Branches

A better way to split fixes and features is to use branches. As explained in the introduction, branches are lines of development that do not interfere: a series of commits that you will only see when you are working in the correct branch.

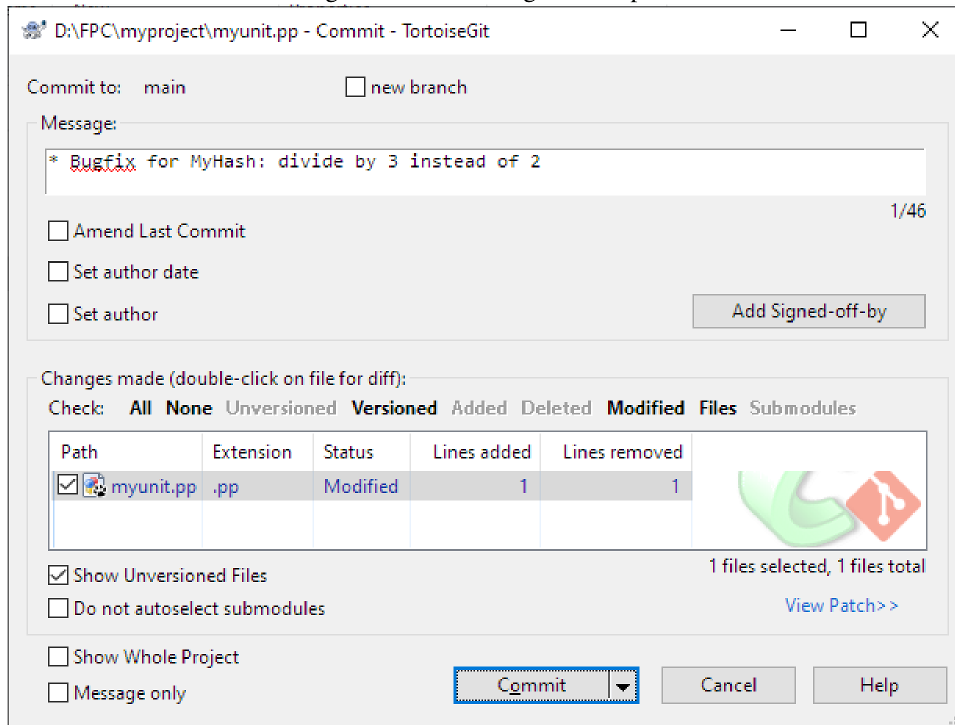
Branches need to be planned: in difference with the above case where you decide to split a commit to clearly separate a fix from a feature at commit time, the decision to use a branch should be made before changing files (although you can work around this).

So, when to branch? Clearly, when parallel development of 2 features are planned that you expect will take a long time, developing them in separate branches makes sense.

But for a small bugfix (a typo or so), it’s a matter of taste whether or not to create a branch for this. One could argue that this is a matter of being consequent: do every new development in a branch, and merge it to the main development line when it is ready.

Creating a branch is easy and can be done in 2 ways:

Figure 4: Committing the first part




```
git branch myfeature
```

This will create a new branch at the currently checked out commit, which is probably the standard case.

But if you want to create a branch at an earlier (or later) commit, you can instruct Git to do so:

```
git branch myfeature 5b10ccbfdcf73f88a1dd790346e73563788314dd
```

Here `5b10ccbfdcf73f88a1dd790346e73563788314dd` is the hash of a particular commit in the commit history, which you can find in the commit logs:

```
commit 5b10ccbfdcf73f88a1dd790346e73563788314dd (myfeature2)
Author: Michaël Van Canneyt <michael@freepascal.org>
Date:   Wed Oct 13 21:51:54 2021 +0200

    * Bugfix for MyHash: divide by 3 instead of 2
```

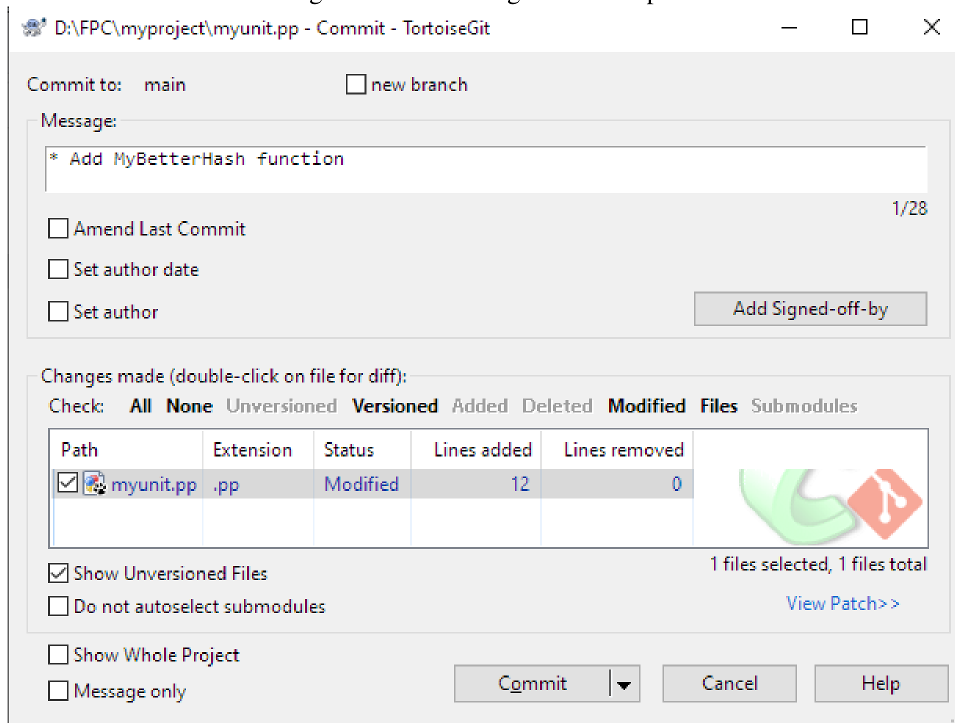
You can also use the short form of the hash:

```
git branch myfeature 5b10ccb
```

Note that at this point, the current branch has not yet changed: all the above commands do is create a new branch. To make this branch the active branch, the `switch` command can be used:

```
> git switch myfeature
Switched to branch 'myfeature'
```

Figure 5: Committing the second part



Now, every change you make and commit will be part of the branch 'myfeature'.

The second way to create a branch uses a variation of the `switch` command:

```
git switch -c myfeature
```

The `-c` switch instructs Git to create the branch and immediatly makes it the active branch.

If we now commit something in the branch, then git will show us that the branch was updated:

```
> git commit -m '* Added text 1' README.md
[myfeature 5836f28] * Added text 1
 1 file changed, 1 insertion(+)
```

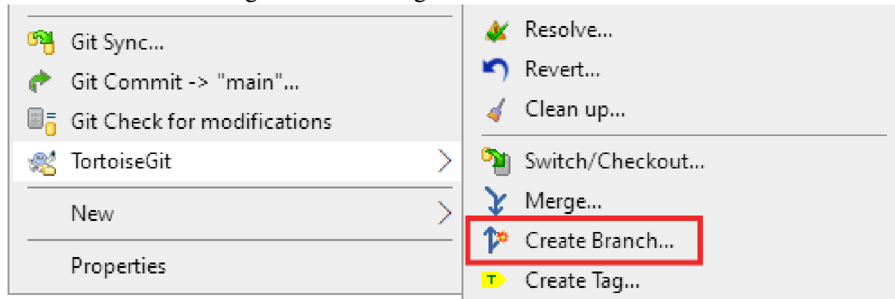
We can add a second commit:

```
> git commit -m '* Added text 2' README.md
[myfeature 57a7dbb] * Added text 2
 1 file changed, 2 insertions(+)
```

To create a branch in TortoiseGit happens using the context menu, shown in figure 6 on page 11. In the dialog that appears then (see figure 7 on page 11), you can enter the details for the new branch:

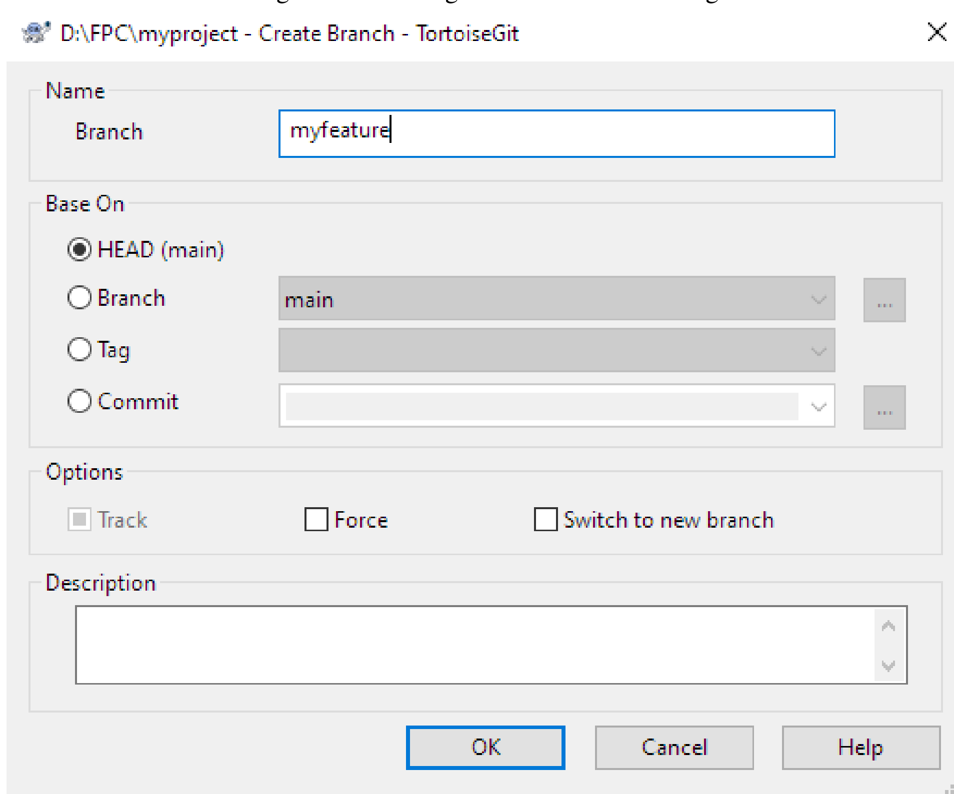
- The name of the new branch.
- 'Base on' here you indicate what commit the branch should start: the default is HEAD on the current branch, which the name Git uses for the current commit. But you can also specify the tip of an existing branch, a tag (which is just a symbolic name for a specific commit), or a specific commit.

Figure 6: Tortoisegit ‘Create branch’ menu



- Whether Git should immediatly switch to the new branch.

Figure 7: Tortoisegit ‘Create branch’ dialog



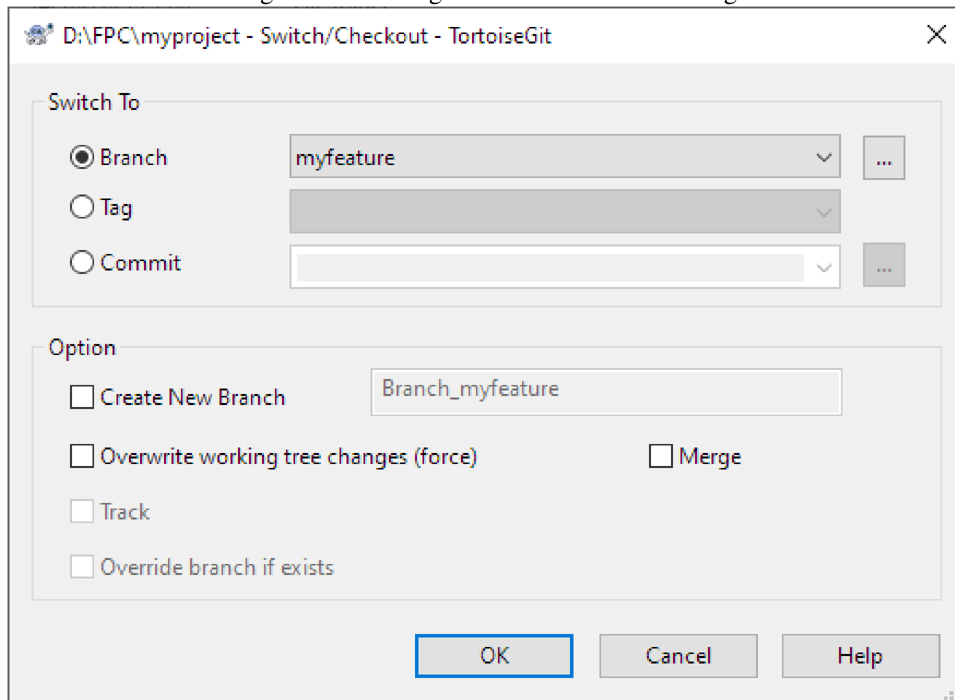
Assuming you did not switch to the newly created branch yet, you can still do so after the branch was created. The `Switch/Checkout` context menu item can be used for this, and it will show the switch dialog as in figure 8 on page 12

4 Merging branches

Once you are done with the new feature or bugfix on your branch, it is time to merge your changes back into the main development line.

This is done with the Git `merge` command. This command will take all changes in a

Figure 8: Tortoisegit ‘Switch/Checkout’ dialog



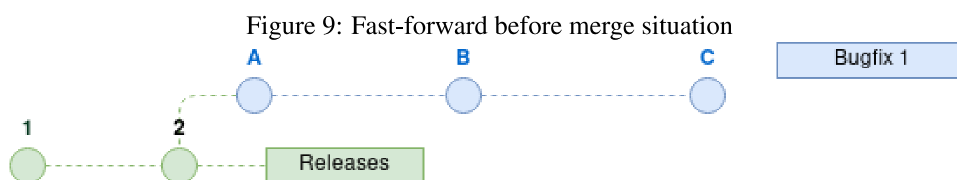
branch and merge them into the current branch. The following 2 commands demonstrate this:

```
> git switch main
Switched to branch 'main'
Your branch is up-to-date with 'origin/main'.
> git merge myfeature
Updating 324ccf6..57a7dbb
Fast-forward
 README.md | 3 +++
 1 file changed, 3 insertions(+)
home: ~/source/articles/git/demo (main)
git merge myfeature
```

The output `Fast-forward` means that Git was able to apply your changes as-is: Git stores all commits as diffs between one version and the next of a file. When merging, Git attempts to replay all the diffs of the branch you want to merge to the current branch: when the current branch does not have any commits that the branch you're trying to merge has not, then this is called a fast-forward.

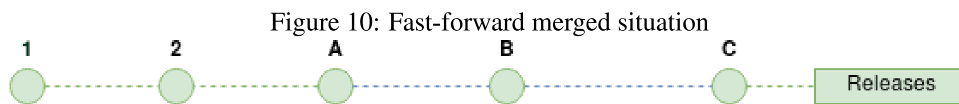
This situation is depicted in figure 9 on page 12: when branch `Bugfix 1` is merged into branch `Releases`, git will perform a fast-forward. When the fast-forward is done, the

Figure 9: Fast-forward before merge situation



history will look like figure 10 on page 13:

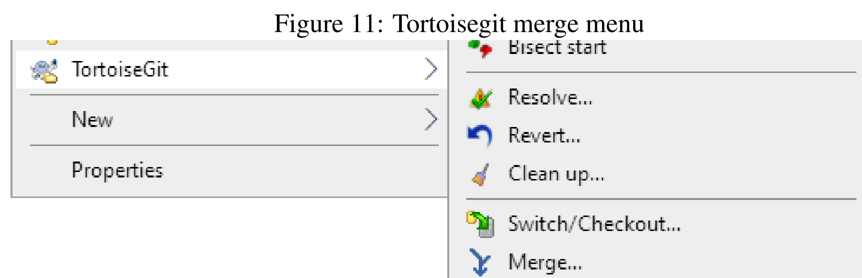
Figure 10: Fast-forward merged situation



The same is true if you do a `git pull`: If your local repository does not have any commits that a remote repository has, then git will perform a fast-forward: a `pull` operation is simply fetching all commits from a remote branch, and then merging them into the locally checked-out branch.

To perform a merge in TortoiseGit, the 'Merge...' menu item must be chosen from the TortoiseGit popup menu (figure 11 on page 13). When chosen, the `Merge` dialog appears

Figure 11: Tortoisegit merge menu



(figure 12 on page 14), which offers you a choice which branch to merge, plus some options to observe when merging.

Squash When you check this, all commits will be squashed into 1 big commit: if you're not interested in the commit history of the branch, but just the result, you can select this. On the command-line you can use `-squash`.

Messages the number of messages to include in a merge commit. On the command-line you can use `-log=N`.

No Fast Forward will force a merge commit, even if the merge can be performed with a fast-forward. On the command-line you can use `-no-ff`

Fast Forward Only only do the merge if the merge can be performed as a Fast-Forward `-ff-only`

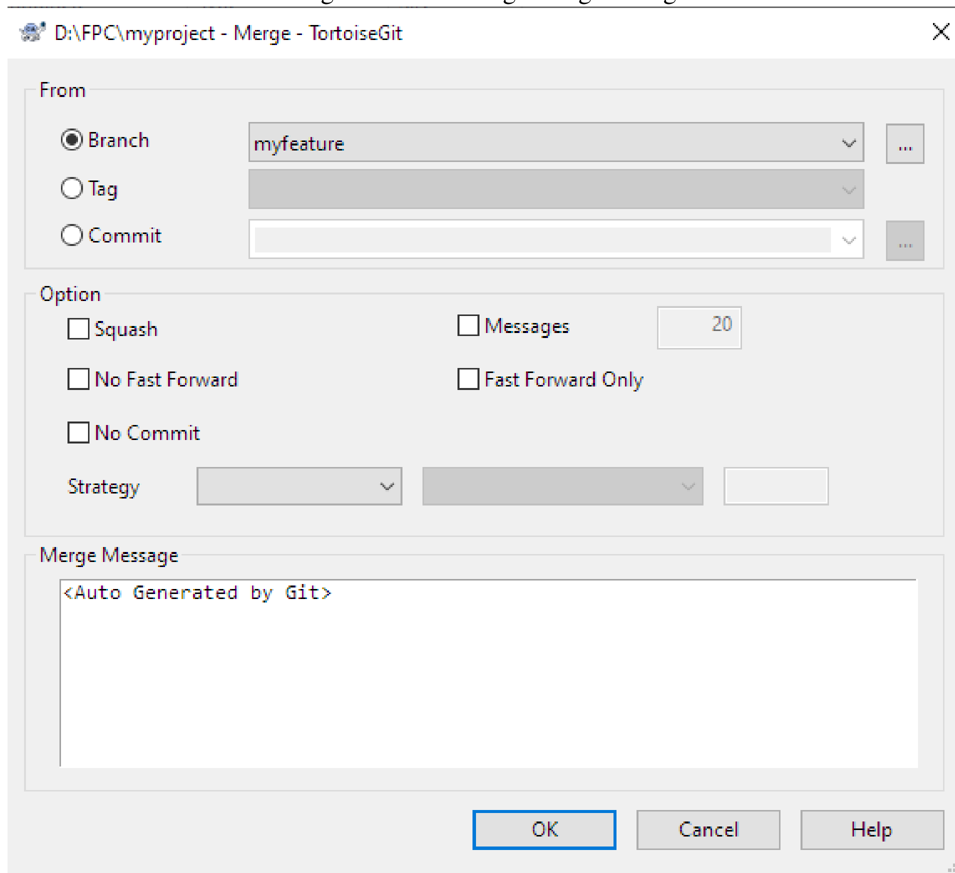
The command-line has more options than shown here, but it would lead too far to discuss them all.

If you do not specify any options, Git will attempt a fast-forward, but if it fails, it will create a merge commit to perform the merge. TortoiseGit will show the result of the operation in the merge result dialog, depicted in figure 13 on page 15. Note that this dialog offers to delete the merged branch: if the branch is no longer needed, it's a good idea to remove it.

5 Rebase

There are developers that do not like the large trees of branches. The situation in figure 1 on page 1 is actually quite simple, but over time, having many people working on a project simultaneously can create a tangle of branches and commits, for an example see figure 14 on page 16.

Figure 12: Tortoisegit merge dialog



(image taken from `https://blog.carbonfive.com/always-squash-and-rebase-your-git-commits`

By contrast, a situation like figure 10 on page 13 where a merge can be done without merge commits is more simple, it looks like a linear history, although in reality the commit history is not linear.

In fact, such a situation can be achieved with a git command: Git stores commits as diffs. When merging the diffs from a branch are replayed on the branch you're merging into.

The use of diffs suggests a mechanism to work towards a 'fast-forward' scenario in the commit history: This mechanism is implemented using the `rebase` command. What does it do?

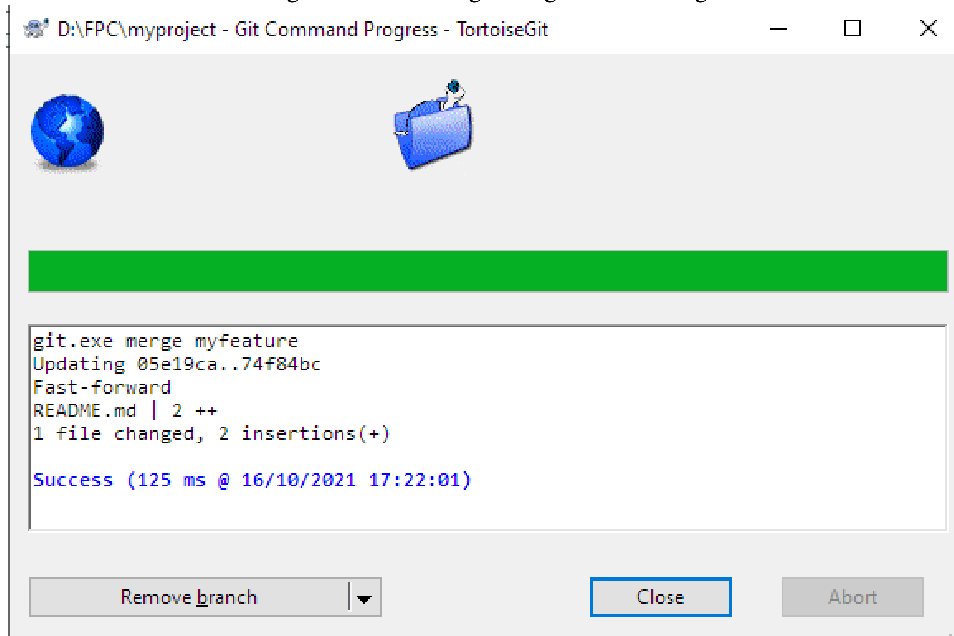
The rebase command must be executed in the branch you want to merge, for example in the `Bugfix 1` branch as shown in figure 15 on page 17: this depicts the same repository as figure 1 on page 1, but without the merge commit `6`. You must tell the rebase command what branch you want to rebase on (Releases in the case of the figure).

When the `rebase` command is executed, it will restore the situation of commit `2`, apply all the commits in the `Releases` branch, and rewrite the commits in the `BugFix 1` branch `A-C` in such a way that they appear to have been applied after commit `5` in the `Releases` branch.

What the Git `rebase` command does, looks like this (the releases branch is called 'main' in the example):

```
> git switch Bugfix_1
```

Figure 13: Tortoisegit merge result dialog




```
Switched to branch 'Bugfix_1'
> git rebase main
First, rewinding head to replay your work on top of it...
Applying: * MyBetterHash2 start
Using index info to reconstruct a base tree...
M myotherunit.pp
Falling back to patching base and 3-way merge...
Auto-merging myotherunit.pp
Applying: * Implement MyBetterHash2
```

This shows exactly what is being done: it rewinds to the branch point, applies the patches from the main branch and then applies the patches from the `Bugfix_1` branch.

Now, when the `Bugfix_1` branch is merged to the `Releases` branch, Git can perform a Fast-Forward, and the end result will be a more linear history, as shown in figure 16 on page 17.

It is important to note that the `rebase` command can fail: when incompatible changes are encountered in the two branches, the process will stop when applying a patch that fails: In that case, you must manually fix the incompatibilities, add the changed file to the staging area and then execute the `rebase` command again with the `-continue` option: git will then resume the rebase operation and apply the remaining patches.

To do this in TortoiseGit, the 'Rebase' menu item must be chosen from the context menu, as in figure 17 on page 18.

When this menu item is chosen, the dialog in figure 18 on page 18 is shown. In this dialog, you must select the branch to rebase (top left), and the current branch is selected.

At the top-right you must select the branch on which to rebase: the 'Onto' button will open a dialog showing a tree of branches. You can also select one from the dropdown list. The list of commits that will be rebased is then shown, and they are all selected (selected commits have the word PICK before them). You can, if you so desire, using the buttons below the list rearrange or skip some commits. For instance commits that are known to

Figure 14: Git tangled branches

result in conflicts can be removed in this way.

The ‘Start rebase’ button can be used to start the rebasing process (remember, it can fail): The log of what git is doing will then be shown in the bottom half of the dialog. When all has completed successfully, the result will look for example like in figure 19 on page 18.

As noted earlier, branches in a remote repository are branches like any other, and doing a ‘git pull’ is like doing a merge from a remote branch: the story of entangled branches and a lineair history as explained here with purely local branches, also applies across repositories.

In order to get a lineair commit history, we must enable Git to always do a merge from the remote repository using a fast-forward. To ensure this, you can tell Git to rebase your local changes on top of the upstream branch when doing a `git pull`. The option to do this is called `-rebase`, so the command becomes:

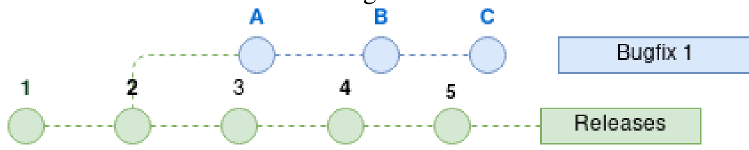
```
git pull --rebase
```

In TortoiseGit, there is an option in the ‘git push/pull’ dialog to achieve the same effect.

6 Conclusion

In this and the two previous articles, we’ve explained the basic operation of git: basic operation of git is not very different from Subversion, but Git provides more possibilities,

Figure 15: Git rebase effect



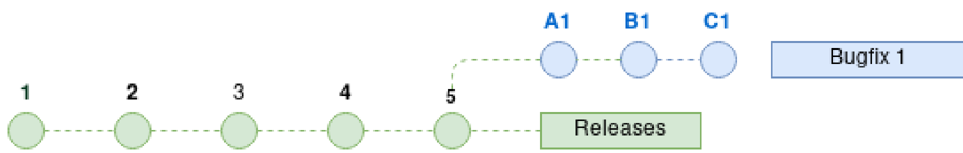


Figure 16: Git merge after rebase

more than we've treated in this article. The Internet is however full of resources that treat git in any desired level of depth.

Figure 17: TortoiseGit rebase menu

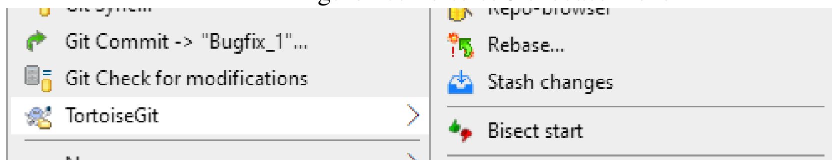


Figure 18: TortoiseGit rebase dialog

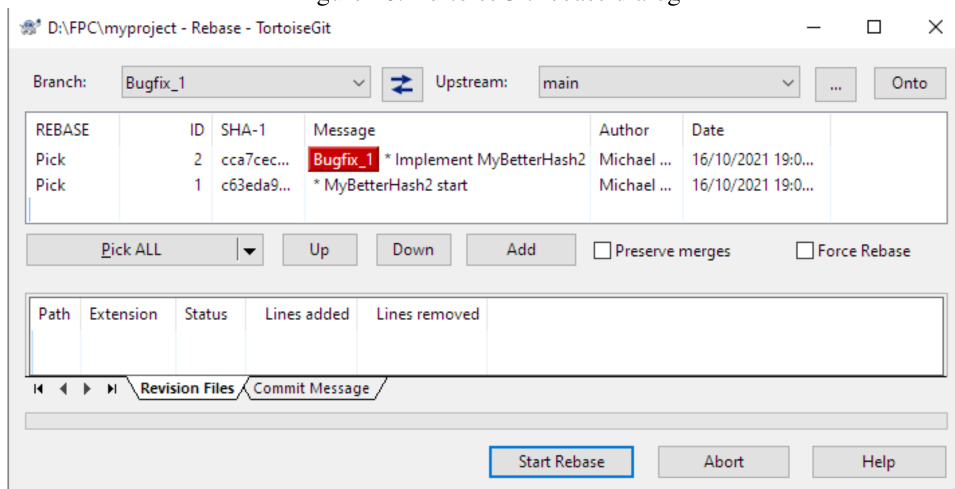


Figure 19: TortoiseGit rebase dialog showing the result of the operation