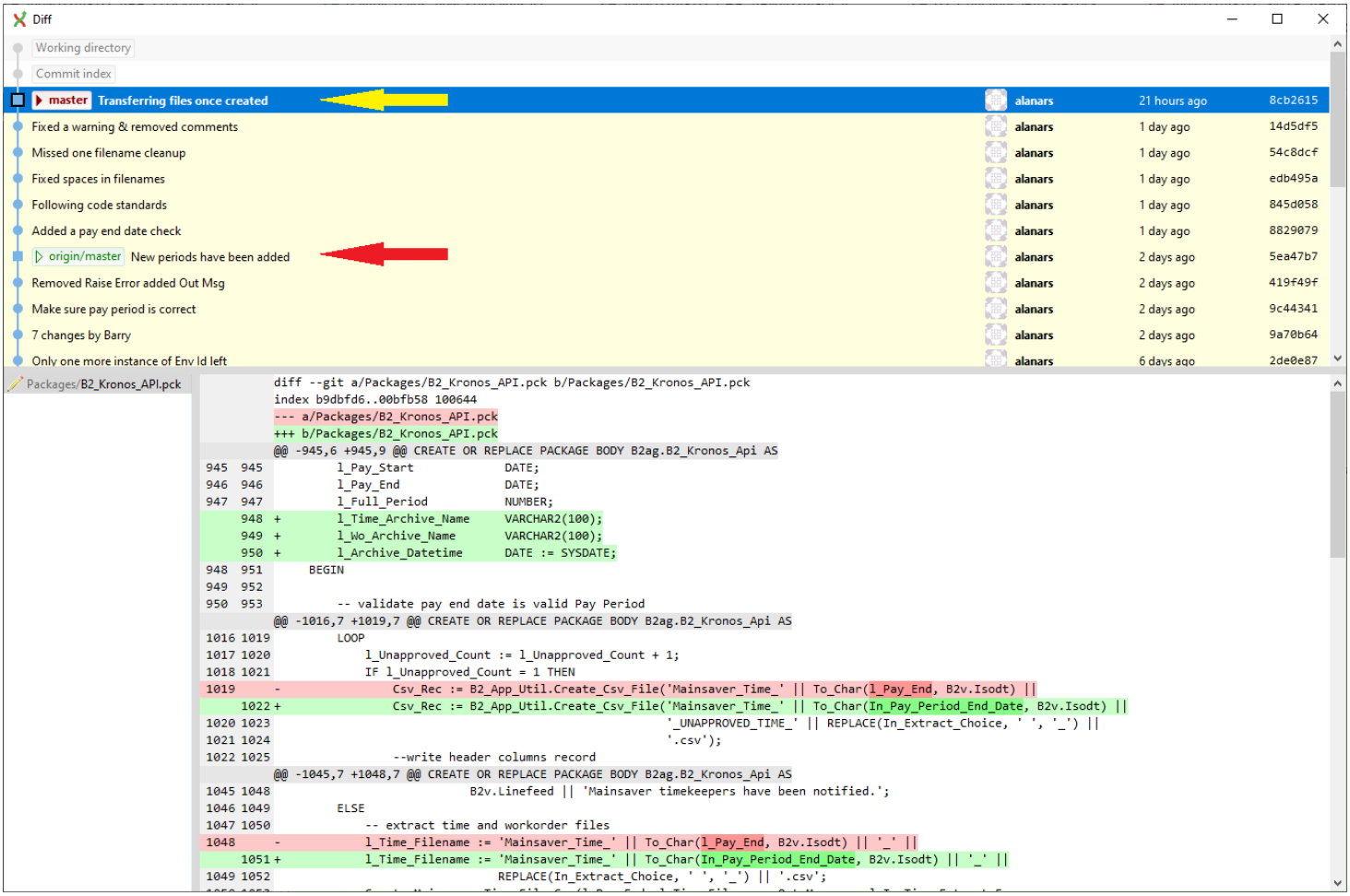
GIT TUTORIAL

PL/SQL Developer 14.0.0.1961

Git Extensions 3.3.1.7897

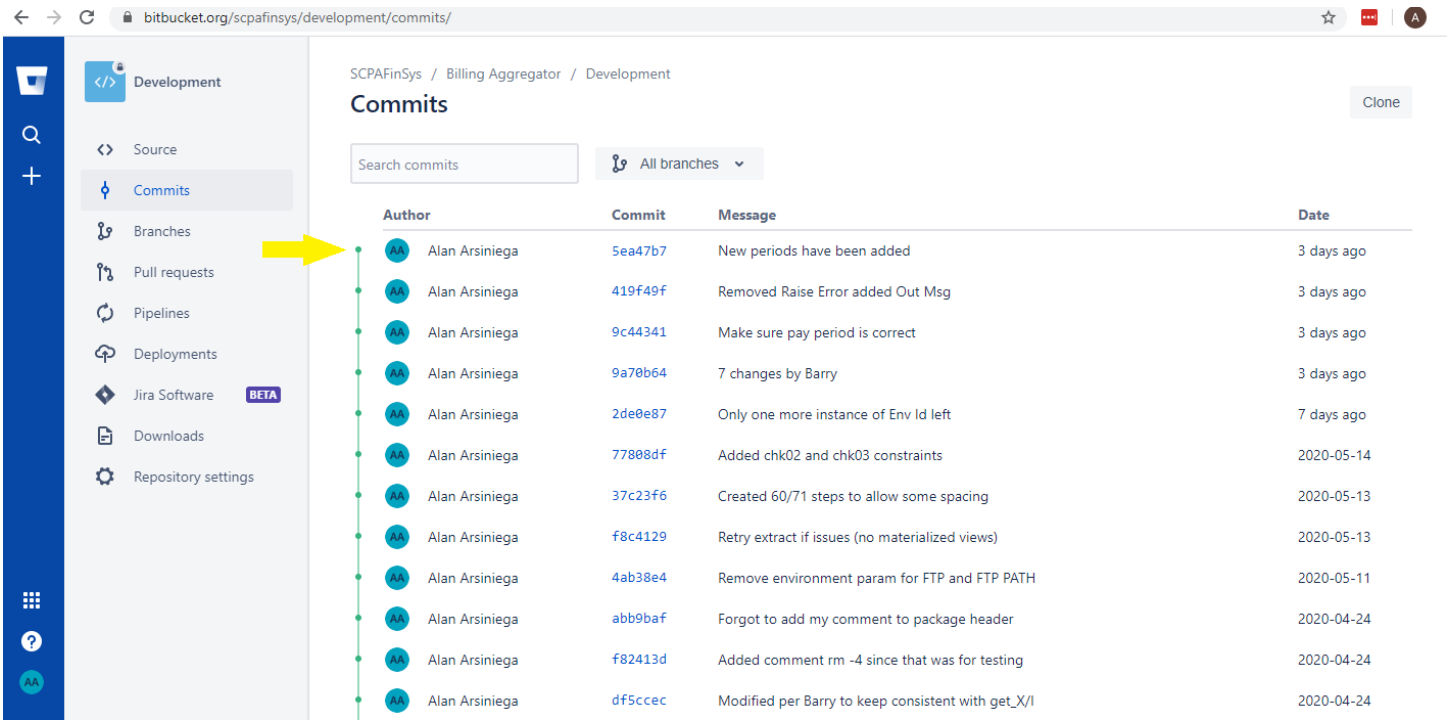
View Changes Using GIT extension in PL/SQL (Right-Click on Files -> Git Extensions -> View Changes):



Notice how *origin/master* (which is the repository in the cloud) is behind 6 commits? Why? Master is the current branch. We can think of master as the trunk or root of the source tree…

This beautifully shows how we've been changing code and by adding meaningful commit comments we get a deep understanding and can always come back to reference if we forget.

Let's look at our repo online (https://bitbucket.org/scpafinsys/development/src/master/):



So, our *origin/master* has all the code from our last *push*. Our *local master* on our desktop is about 6 *commits* ahead. Let's look and see if we have any pending changes we haven't committed to our local master before we *push*.

Stage code, commit, and push (Right-Click on Files -> Git Extensions -> Commit)

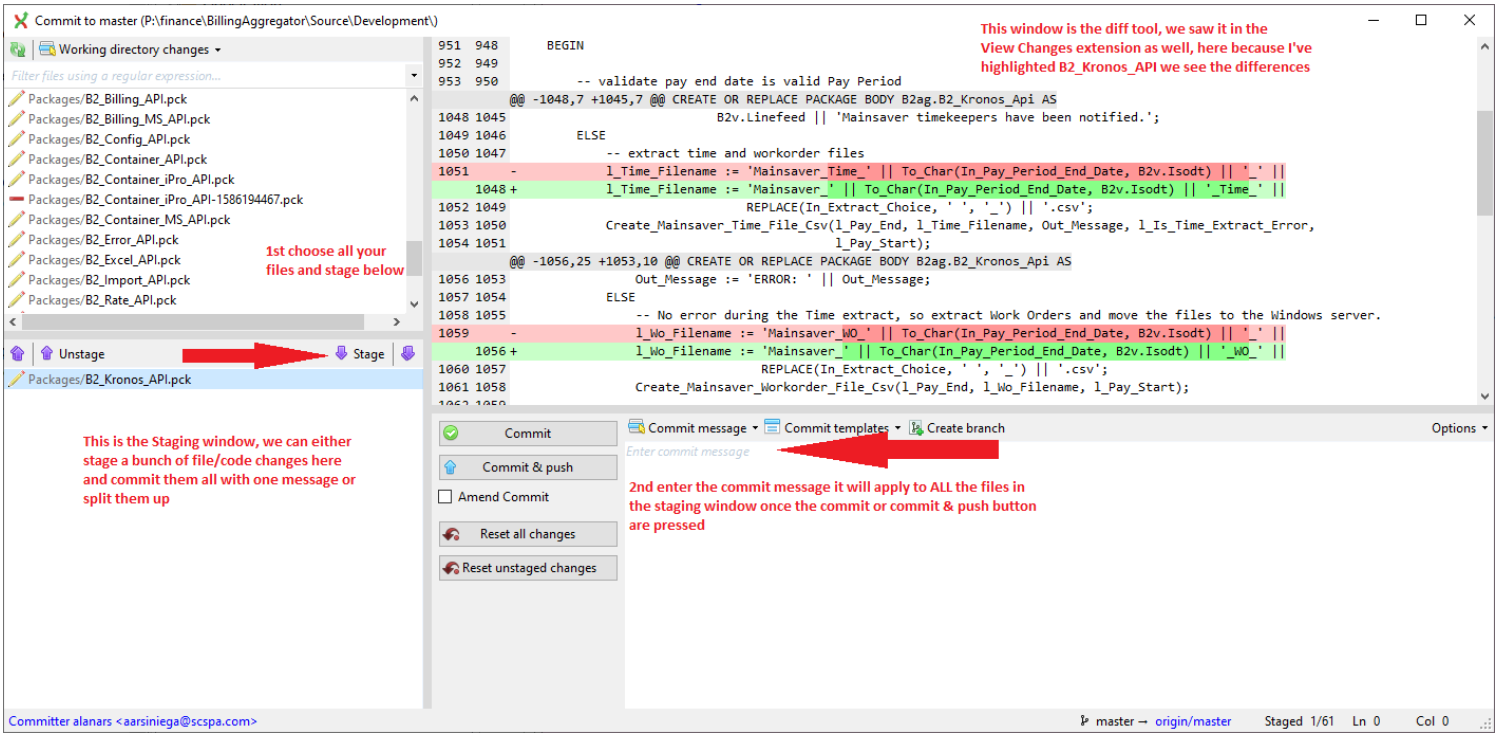
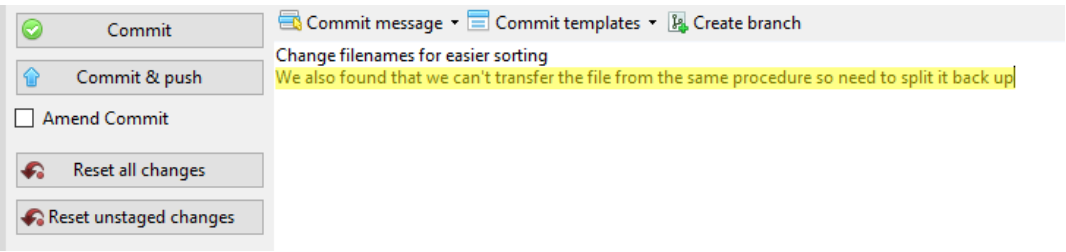


Figure 1 There's actually a WHOLE lot of changes here that are asking to be committed… Everything in the 1st box.

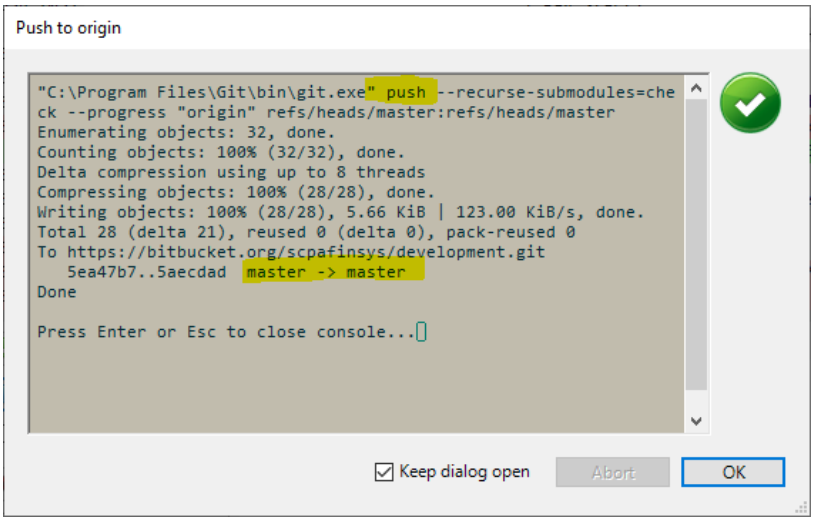
Here is a good *commit* message. After this I was ready to *push* my *local master* to *origin/master***.

Had I wanted to work on the code more I would have just clicked *Commit* instead of *Commit & push*.



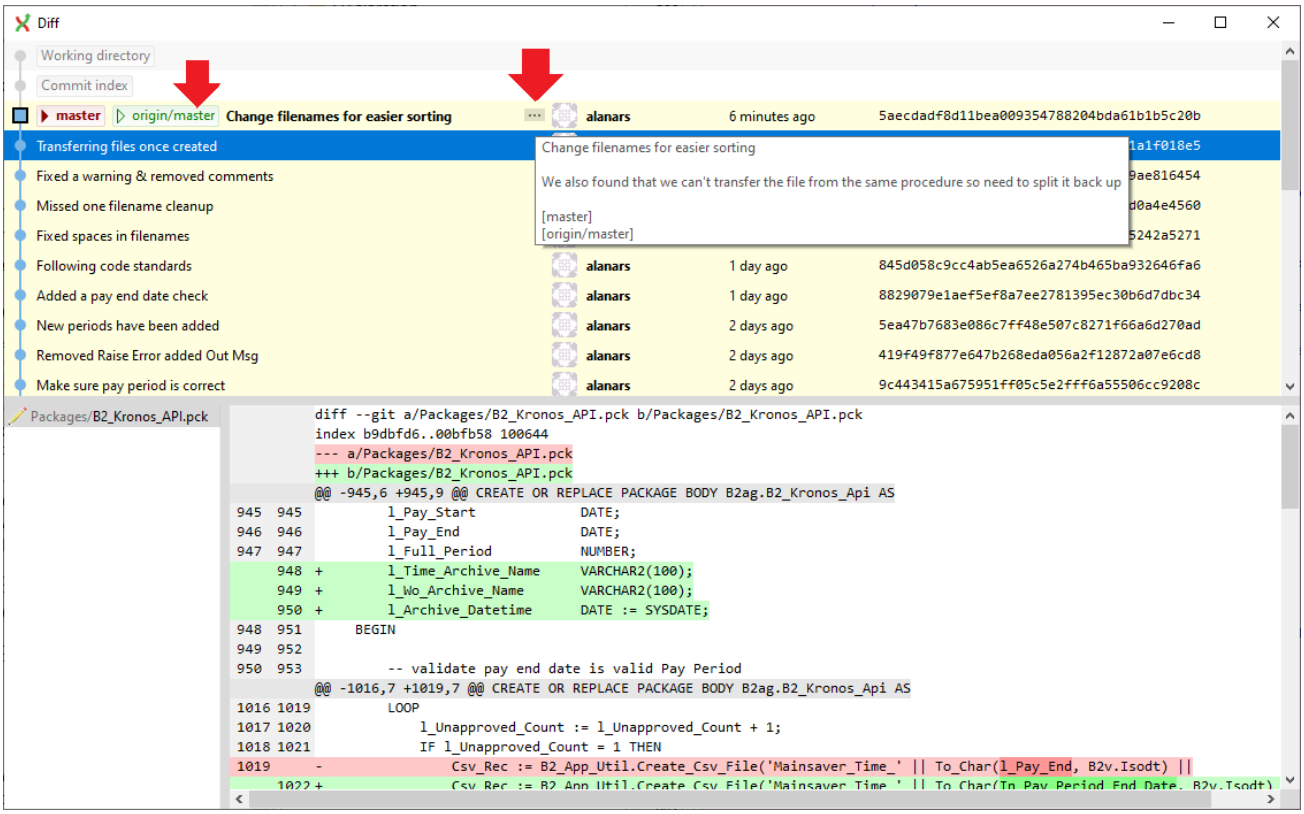
1st window tells us what the commit did (notice the commit cmd being issued):

Once we click OK we get the 2nd window (because we chose Commit & Push) which shows us the result of the push:



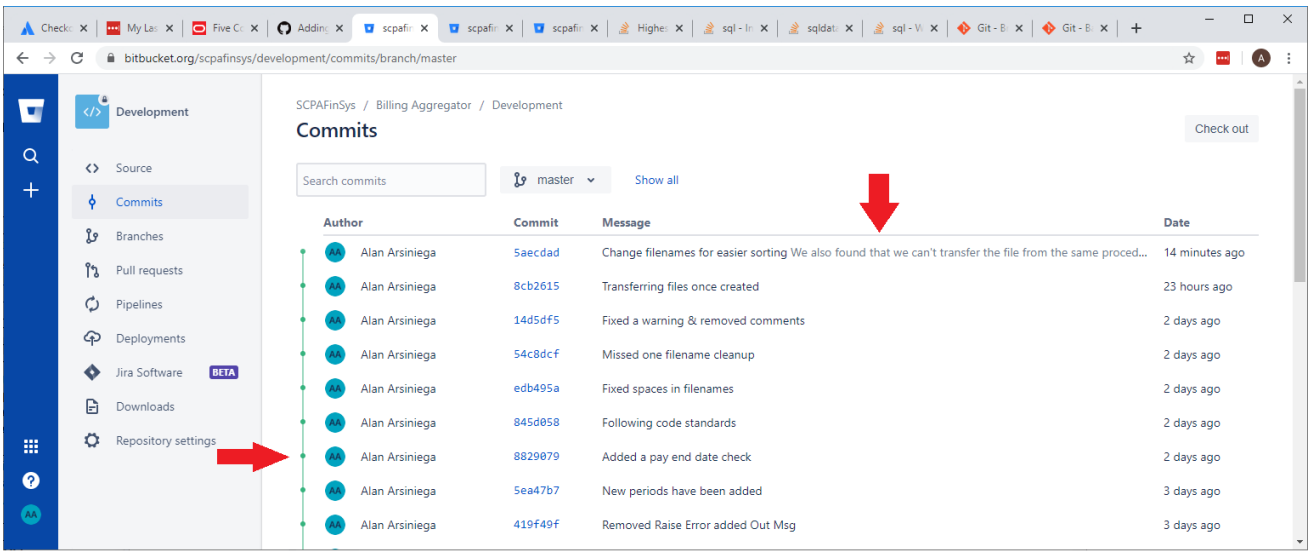
**(Notice that *master* is our branch, *local* is our PC or wherever the local code resides and *origin* is the cloud/bitbucket or wherever our main source code rests)

Let's view our changes:



First, we see that our master branch is caught up with origin/master. Also, by adding extra comments to the commit message we hover our cursor over the ellipsis (…) and see the full message.

Let's check origin/master (bitbucket):



That push caught up origin with ALL the previous commits that had been done on that branch (which happens to be master). Also, the extended commit message is visible just need to hover cursor over to see pop-up window.

Possible Hardships working with Master

The problems with working on solely master are numerous. A cursory google search rendered 3 articles:

Don't Mess with the Master: Working with Branches in Git and GitHub

(https://thenewstack.io/dont-mess-with-the-master-working-with-branches-in-git-and-github/)

Now it is time to start actually working with GitHub (and git) the way they are meant to be used: making changes in the project safely off to one side, and merging them back into the original project once they have proved to be correct. Or at least not disastrous.

Git-what-issues-arise-from-working-directly-on-master

(https://softwareengineering.stackexchange.com/questions/335654/git-what-issues-arise-from-working-directly-on-master)

If another developer starts work for a new feature from master, she starts with a potentially broken state. This slows down development

Different features/bugfixes are not isolated, so that the complexity of all ongoing development tasks is combined in one branch. This increases the amount of communication necessary between all developers

new features need their own development branch that can be deployed to a test environment before it is pushed to production.

Otherwise, you're in a perpetual state of half-completed features. You can't deploy half-completed features to production, so if you're working directly on the master branch, everyone else must wait for you to finish your feature before anyone else's changes can go to production, including bug fixes.

Reasons-for-not-working-on-the-master-branch-in-git

(https://stackoverflow.com/questions/5713563/reasons-for-not-working-on-the-master-branch-in-git)

The master branch should represent the 'stable' history of your code. use branches to experiment with new features, implement them, and when they have matured enough you can merge them back to master.

Using independent branches for features means that each new feature can be tested and deployed independently of the others.

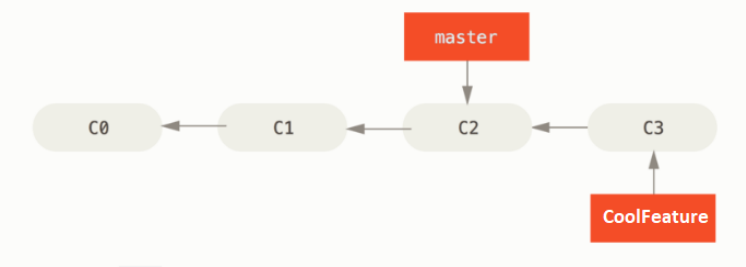
Luckily, branching amounts to learning two extra commands or just using the pl/sql extension further.

Branches in our Repository (Right-Click on Files -> Git Extensions -> Open Repository)

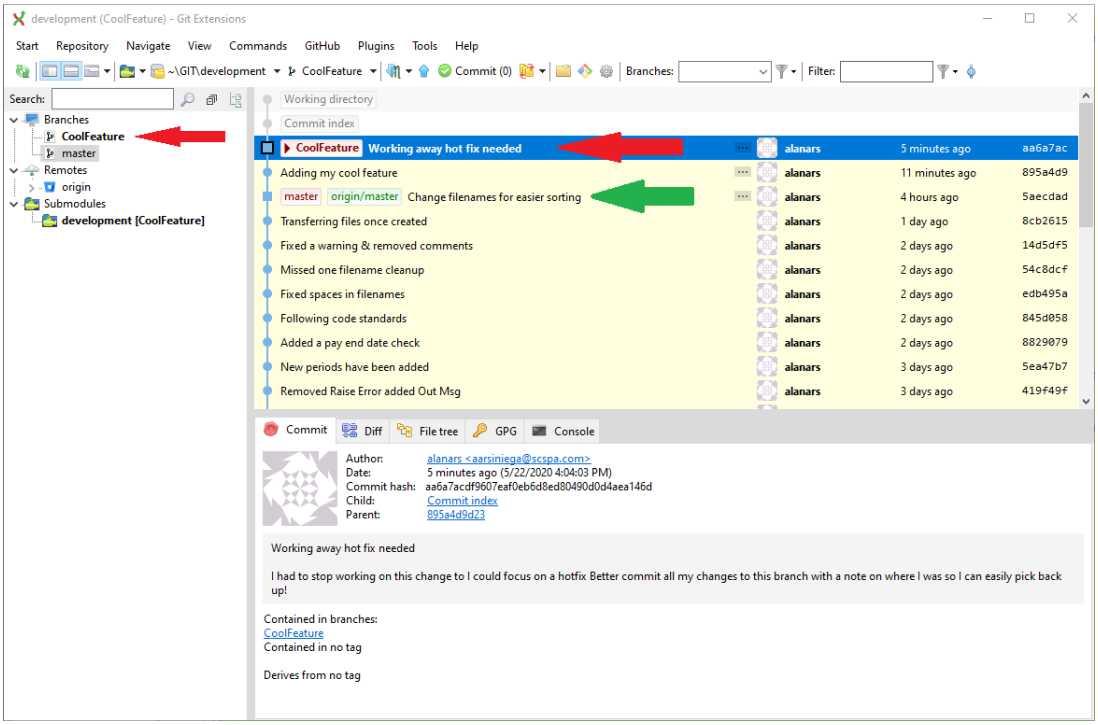
GIT has hast mastered working with branches to organize our work, whether it's a hotfix, a feature, or some lesser important bug fix – we can easily switch in and out of branches to logically break up the work. This eliminates the need of manually keeping track of which code changes are developed and tested to get pushed and which ones haven't and need to be commented out, for example.

We will create a branch, create and switch into another branch, and then merge to master from the completed branch (https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging)

Some of this gets very repetitive so let's assume we are at this point (from the link above):



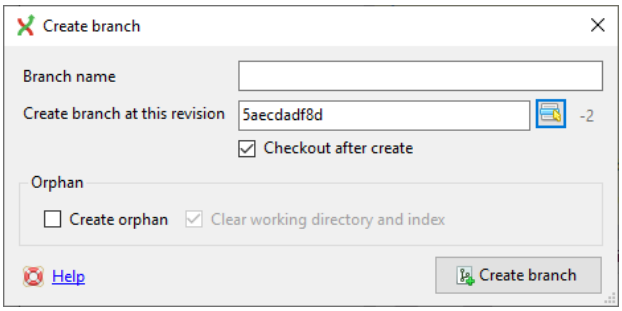
We open the repository (**Right-Click on Files -> Git Extensions -> Open Repository**) and we have:



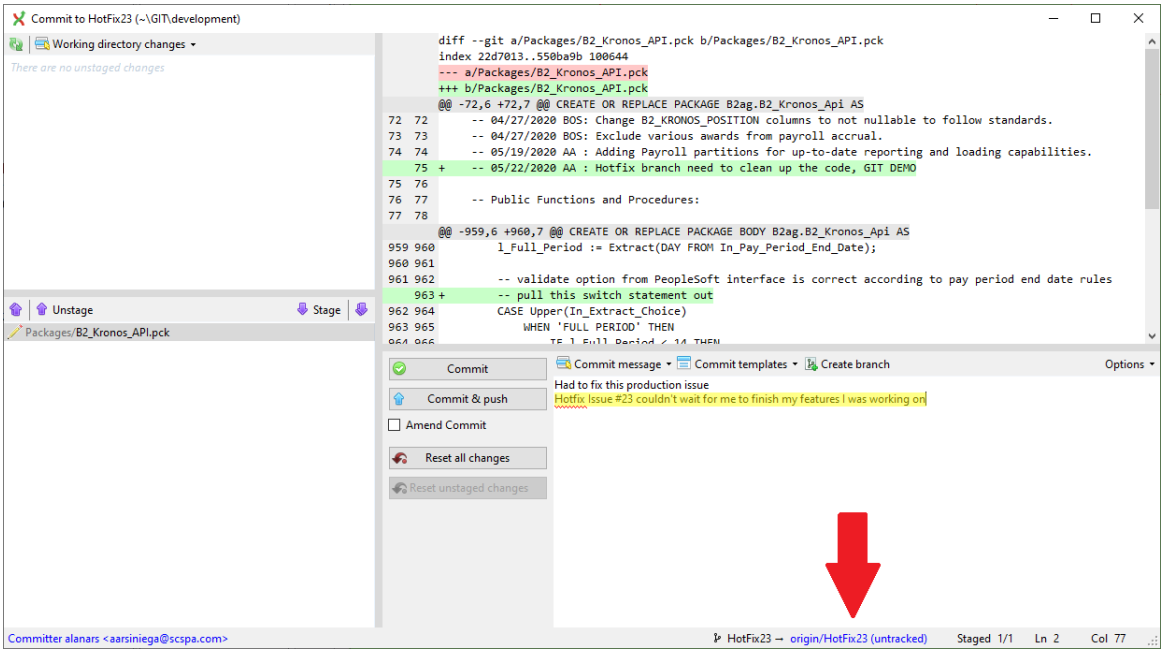
Notice how master is just another branch (left arrow). We've added our CoolFeature branch that we were diligently working on (right red arrow) when the request for a Hotfix comes rushing in. We will create that hotfix branch despite already being two commits ahead of master on our CoolFeature branch, as you can see above. Because our ticket system (like Jira) has an ID of 23, we can use that as part of the branch name.

Right-Click anywhere on the master, origin/master row (green arrow) to create a branch and choose '**Create new branch here… Ctrl+B**' from that point. It makes sense to branch from there and not keep adding on to our current

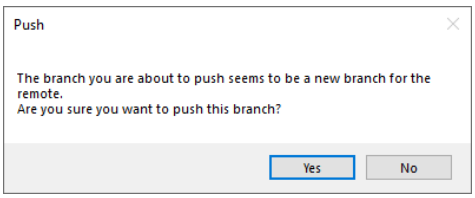
CoolFeature branch because we are still in the middle of development on that branch so we can just split from a point in time when master was in the ideal state.



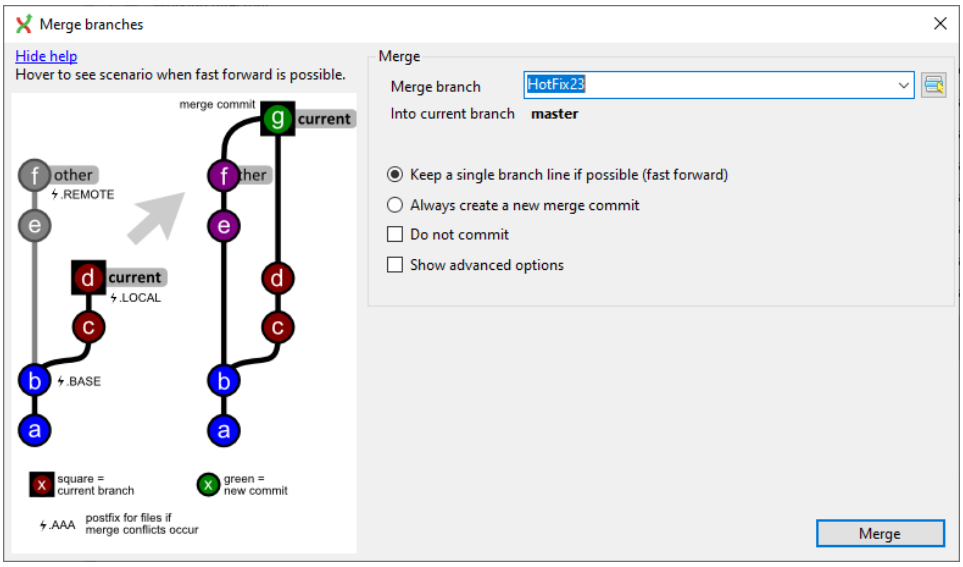
Let's call this branch HotFix23 and leave "Checkout after create" box checked this automatically check out this branch for you so that your local repo is now on this branch (it will show as bold in the Repository, left arrow above.) We will make a change to our code in the HotFix23 branch and commit the changes (**Right-Click on Files -> Git Extensions -> Commit**):



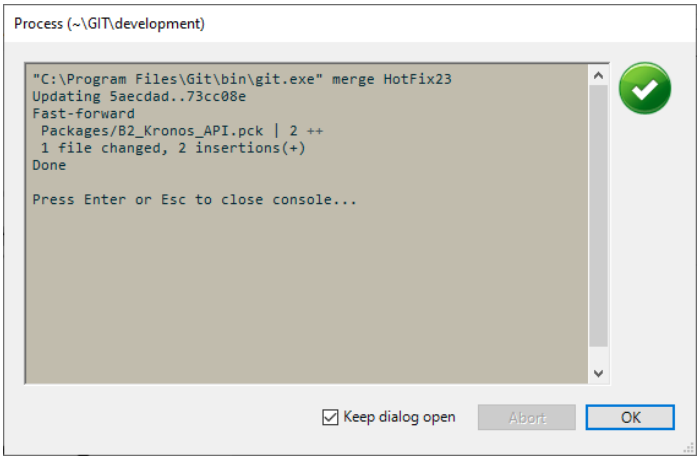
Way on the bottom we see origin/HotFix23 (untracked). This means the branch exists locally but not in *origin*, we can change that by choosing to Commit & Push and you would get this message to confirm:



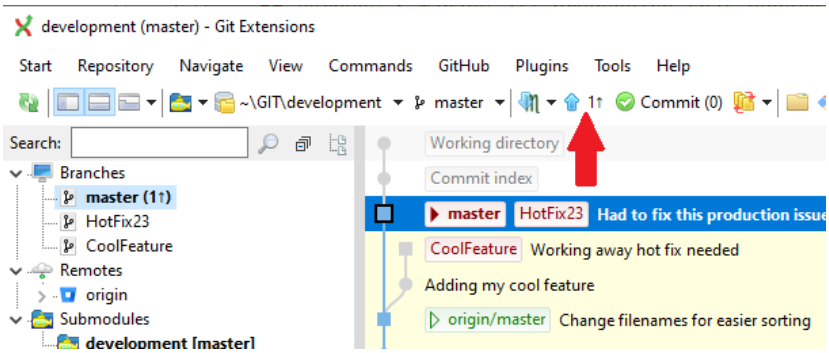
Let's choose Commit and am we now ready to merge this HotFix23 branch. Open the Repository, right-click on "master" underneath "Branches" and choose Checkout. A window pop us letting us know we've switched to branch "master." Hit OK and the font on "master" turns to **bold**. Now, choose the correct branch we want to *merge* into master, in this case HotFix23, right-click on it and choose *Merge.* Now we should see this window, with default settings:



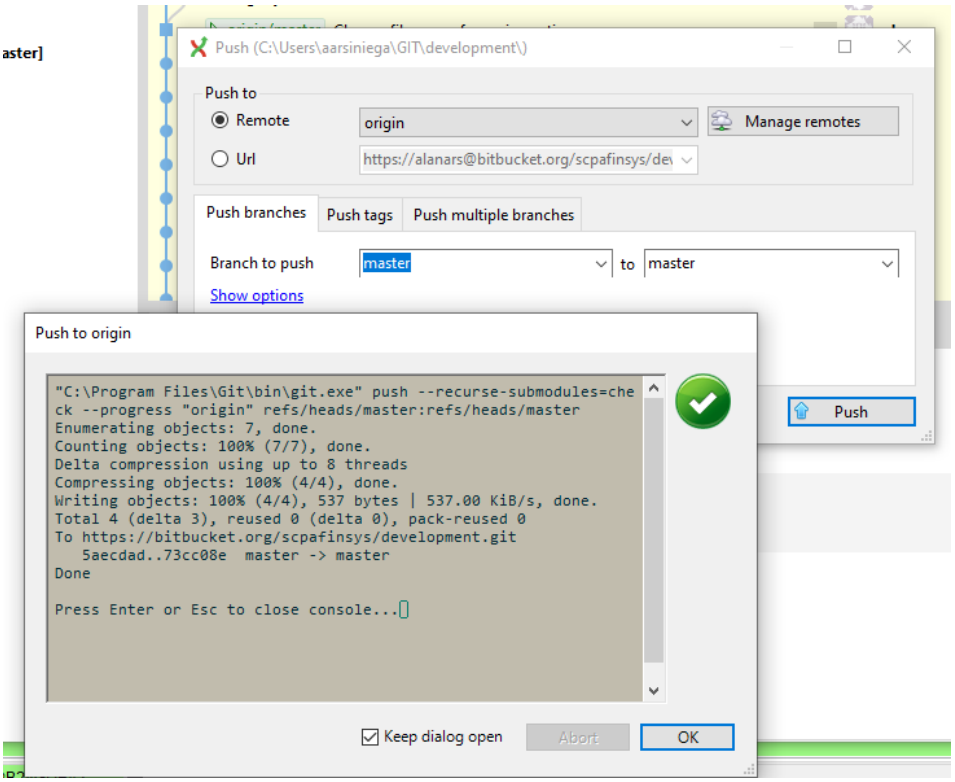
Pay close attention we have the right Merge branch and it's going into the right current branch, in this case **master**. Click merge, now we see:



Success! We have one small thing to do before we run this code in Production. So far, we are working on our local branch. We need to get these changes to the origin/master in the Cloud so they're visible by all. Still in the repository we notice a blue arrow with a (1) beside it:



I think this is saying we have a major merge we need to push up to origin/master since our local/master branch is now ahead in commits**. So, let's go ahead and click on the blue arrow pointing towards the cloud.



To Quickly Recap full lifecycle of GIT and code changes:

- Create the CoolFeature branch – (Open Repo, right click on master row to Create branch)
- Checkout the branch – (branch turns bold in the Repo if you leave "*Checkout after create*" box checked)
- Code – commit (ensure commits go to right BRANCH, publish the branch to the *origin* if untracked)
- Create hotfix branch from master (Open Repo)
  PL/SQL Developer will warn you the timestamps changed. Always RELOAD or we might commit our coolfeature code into our hotfix branch.
- Code and commit (note code differences when we commit) until hotfix is resolved
  Note new branch name – push to publish branch on BitBucket is optional)
- To merge our hotfix into the master branch
  1) Checkout branch we want to merge into (will show as bold)
  2) Right click & merge on branch you want to merge (hotfix23)
  3) Make sure you're FROM branch is correct and TO is master and click MERGE.
- At this point we're in master branch and should Push this new feature to our origin/master and resolve any conflicts before we run the code in production! Click on blue arrow pointing up to do so.

** If there were any conflicts we would need to resolve them since we are working locally but our origin/master is in the cloud and unprotected so someone could have pushed some commits to it without branching and now we need to resolve those commits if they're conflicting. Most of the times resolving conflicts isn't that difficult, please refer to the git-scm link above for an example.

For this reason, many shops will choose to protect their Master branch and require Pull Requests with a review required or several reviews and maybe even automation to fire off several tests before being able to merge into Master. In some cases, there can be a couple main branches that get pull requests instead adding another layer of protection. Every organization should implement the best strategy that adds the most value to their internal business processes and all their customers.