Git Lab 5
Working with a BitBucket repo

1   Commands covered

Cloning a remote repo	Getting info on a remote repo

`git clone remoteURL`	`git remote`
	`git remote --verbose`
	`git remote show origin`
	`git remote prune origin`

	`git fetch` `git fetch --prune`
	`git branch --remotes` `git branch --all`

Uploading content to a remote repo	Downloading content from a remote repo

`git push`
 	
`git push --set-upstream origin branch-name`
 	
`git push origin --delete branch-name`
	`git pull`

Various operations on remote / local branches

`git checkout repo-handle/branch-name`
 	
`git diff branch-name repo-handle/branch-name`
 	
`git merge repo-handle/branch-name`
 	
`git log --oneline --graph`

2   Lab setup

Make sure you have a suitable text editor installed for your OS.

Make sure you have a valid BitBucket Cloud account.

This lab provides a demonstration based on BitBucket Cloud functionality. This is one of the 3 major Bitbucket product versions that is offered by Atlassian.

We will be cloning a remote repo into a subfolder within the main folder where we previously created our Git repositories (`labs`) . If are using a directory with a different name, substitute that name into all the relevant commands in this lab session.

If you are working with Linux / MacOS, open a terminal shell to type in your Git commands.
If you are working with Windows, open Git Bash we will be typing in the commands via Git Bash. Right click on the directory and select Git Bash Here from the context menu to open the Bash shell in `G:\labs`

The Git commands to type are listed after the $ prompt and their output (or relevant parts of their output) will be shown as well.

3   Creating a private remote repo in a BitBucket account

Login to your BitBucket Cloud account. If this is a completely new account, you should see a main page that looks similar to the screen shot below.
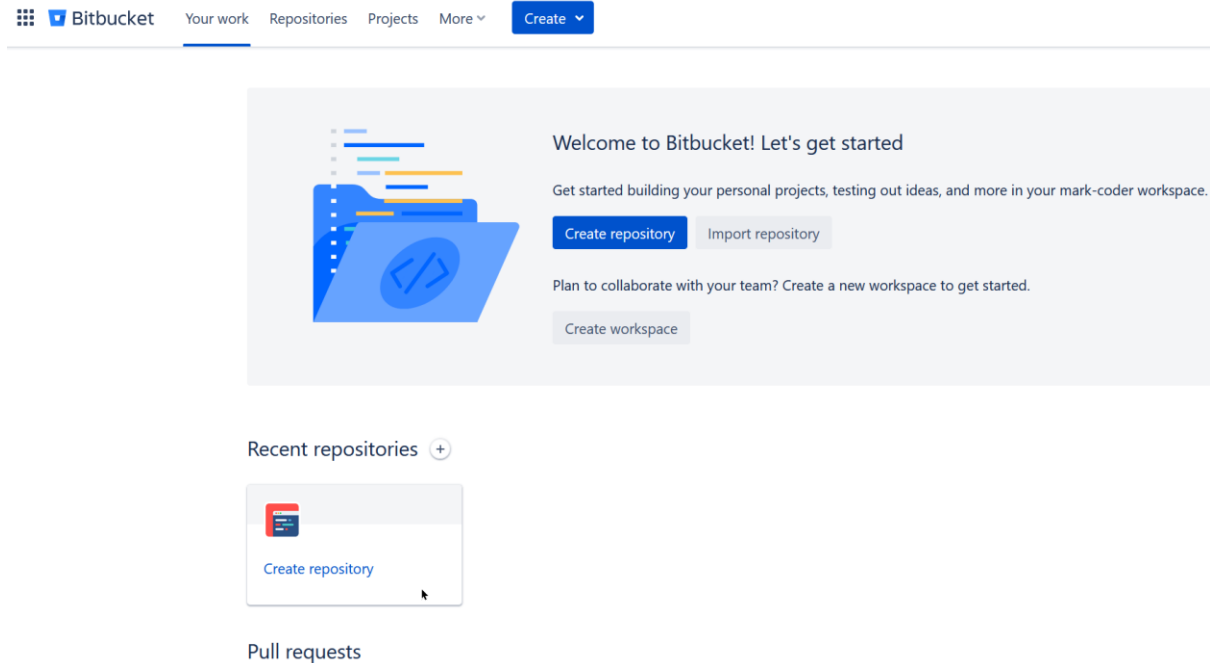

We can proceed to create a new repository in this account.

Enter the values for the following fields.
Project Name: `CoolRemoteGitProject`
Repository Name: `FirstRemoteRepo`

Accept the defaults for the other fields, and click Create Repository. You will be transitioned to the Source view for the newly created repo, where you can view all the content in the project folder.
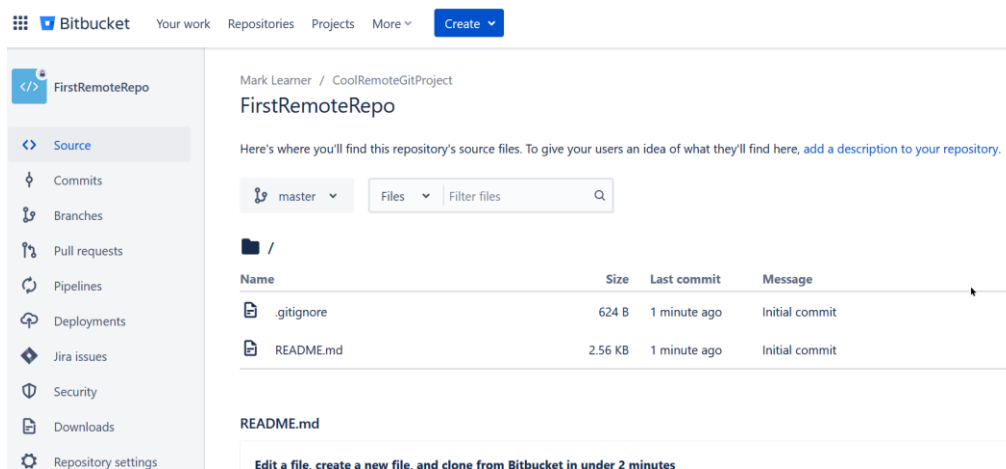
Notice that:

a) The `master` branch is the default branch for a new repo. This will be the active / current branch when the repo is created.

b) The contents of the main project folder are shown in the directory listing immediately below the designated active branch. At the start, it contains a `.gitignore` file and also a [README.md](#) by default. This is a standard text file that introduces and explains a code project in a BitBucket repo (it is also included in other standard Git cloud-hosting services such as GitHub and GitLab). It is typically written in the [Markdown](#) format. The contents of this file are displayed immediately below the directory listing.

You can click on any file in the directory listing to view and edit the contents of the file in a new view. Click on `README.md` and click on Edit to view it in edit mode. Don't make any changes yet. Click on Source in the left pane to return to the Source view.

Click on `.gitignore` and click on Edit to view it in edit mode. Notice the various file extension types that are automatically generated by BitBucket as files that will not be tracked by Git. As you can see, these are a mixture of compiled class types, log files, package files, IDE related files and so on. You can add in additional file extension types that are specific to your application project.

In this lab, we are exploring the most common ways for a single developer to interact with a remote repo created in their own personal BitBucket account. In a future lab, we will explore how a team of developers can collaborate using a shared remote repo using a standard development workflow.

4 Adding content in new commits and new branches

IMPORTANT NOTE: The standard workflow (whether for developers working on their own or collaborating with others in a team) is to clone a new remote repo to a local working copy, make changes there and upload it back to the remote repo (to be demonstrated in upcoming lab sessions). We will rarely ever create new content / branches in the remote repo directly via the browser UI as we are doing here. However, this is being demonstrated here for the sake of completeness.

We will now create a new file and add some content to it and commit these changes.
Select Add file from the vertical three dots menu option.
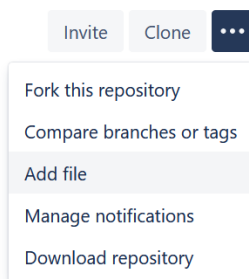
At the top, give the file the name: `countries.txt`

In the file edit tab, enter this new content, making sure to include an empty line at the bottom:

```
Malaysia
Singapore
```

Click Commit on the lower left hand corner to save the changes into a commit on the current branch (master). Enter this for your commit message in the dialog box that appears, then click Commit:

```
Initialized countries with some content
```

You are now transitioned to a commit page where you can view a variety of information regarding this newly created commit (the commit hash, the branch the commit is on, the commit message and a diff between the current commit content and the previous commit in the commit history timeline).

Click on Source in the left pane to return to the Source view. You should be able to see `countries.txt` in the directory listing. Click on this entry and click on Edit button to start editing in the file edit tab. Enter more new content as shown below, making sure to include an empty line at the bottom:

```
Brunei
Thailand
```

Click Commit on the lower left hand corner to save the changes into a commit on the current branch (master). Enter this for your commit message in the dialog box that appears, then click Commit:

```
Added more content to countries
```

Once again, you are transitioned to a commit page where you can view a variety of information regarding this newly created commit (the commit hash, the branch the commit is on, the commit message and a diff between the current commit content and the previous commit in the commit history timeline).

Click on Branches in the left pane to transition to the branches view, where you can see a listing of all branches, including the current active one, master, which is designated as the main development branch.
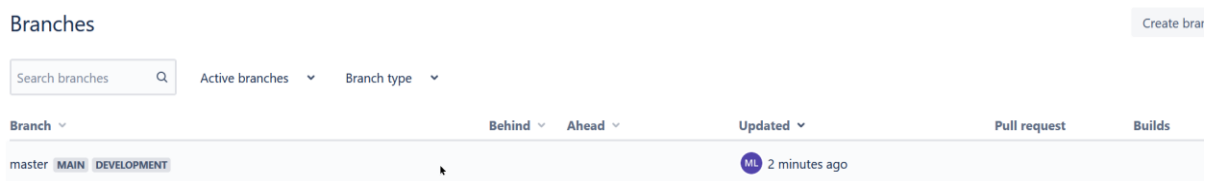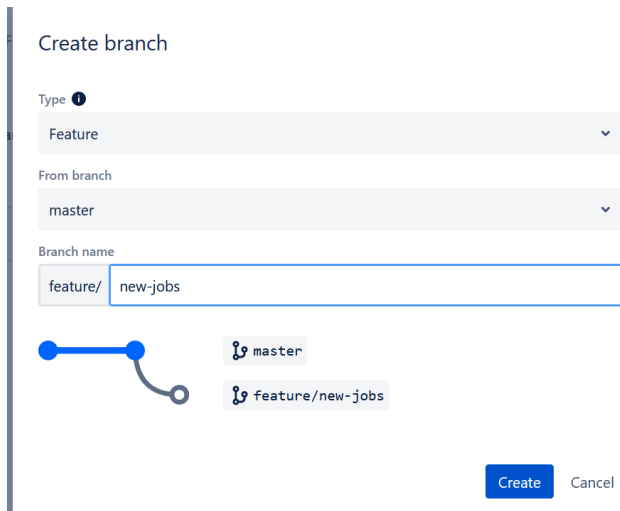

Click on Create Branch in the upper right hand corner. The Type dropdown in the dialog box offers 4 categories to classify branches, as well as Other for branches that don't fall into any category
  a) Bugfix - To fix any bugs discovered in the code, as a result of testing or feedback from users
  b) Feature - To introduce a new feature into the code base
  c) Hotfix - A form of a bug fix, but which can be applied while the system is running without taking the system offline. This is typically found in the Gitflow workflow (and other workflows that derive from it)
  d) Release - This is also part of the Git-flow workflow. The release branch contains all completed features that are ready to be released in the next version of the code base.

For this lab session, we will classify all our branches as Feature to simplify matters: for a real-life Git repository, use the appropriate classification type for the branch that you are creating.

Select a Feature type, and for the new branch name, enter: `new-jobs` and click Create.

In the view for the newly created branch, click on the View Source button to transition back to the Source view


In the Source view, we will now be viewing the contents of the project folder corresponding to the newly created branch: `feature/new-jobs`, which at this point of time will be exactly identical to `master` as we have not added any new commits to it yet.

Let's proceed to add a new file to this branch, add some content to it and commit these changes. Select Add file from the vertical three dots menu option.
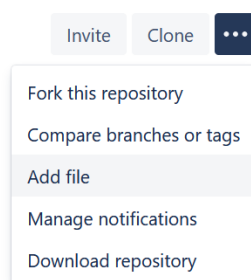
At the top, give the file the name: `jobs.txt`

The file view should clearly indicate you are in the `feature/new-jobs` branch. In the file edit tab, enter this new content, making sure to include an empty line at the bottom:

```
developer
project manager
```

Click Commit on the lower left hand corner to save the changes into a commit on the current branch (master). Enter this for your commit message in the dialog box that appears, then click Commit:

```
Cool software jobs
```

You are again transitioned to a commit page where you can view a variety of information regarding this newly created commit (the commit hash, the branch the commit is on, the commit message and a diff between the current commit content and the previous commit in the commit history timeline).

Click on Source in the left pane to return to the Source view. By default, the Source view shows the contents of the latest commit in the master branch, so you won't be able to see `jobs.txt` (as it was not created in this branch). Select `feature/new-jobs` from the branch drop down menu to switch to this branch, and you should now see `jobs.txt`

Click on Commits in the left pane to switch to the commits view, where you will be shown by default the commit history of all branches in the repo. Select between the branches available from the drop-down menu at the top to view the commit history of a specific branch (at the moment, either `master` or `feature/new-jobs`)

Commits

Author		Commit	Message		Date
ML	Mark Learner	0734495	Cool software jobs	⎇ feature/new-jobs	8 minutes ago
ML	Mark Learner	3ee61c1	Added more content to countries		4 hours ago
ML	Mark Learner	23ffb23	Initialized countries with some content		4 hours ago
ML	Mark Learner	19807af	Initial commit		14 hours ago

Switch back to the Source view, and ensure that master is selected as the current / active branch. We are now ready to clone this remote repo to a local repo on our local machines.

5   Obtaining an app password

When we use commands from Git Bash to interact with the remote repo (e.g. fetch, push and pull), these requests will need to be authenticated to the BitBucket server to ensure that the entity executing the command has the authorization to do so on the specified remote repo. For this purpose, we will need to create an app password beforehand, which we will need to supply with every one of these commands that interact with the remote repo.

Select Personal Settings from the Avatar menu list.
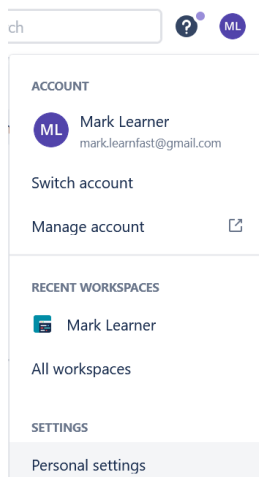
Select App passwords under Access management. Select Create app password.
In the Add App password section, provide this info for the label: `PasswordForFirstRemoteRepo`
and check the following boxes as indicated below:

Then click Create. A dialog box will pop up indicating your new app password.
Make sure you COPY AND PASTE this password into a document (for e.g. empty Notepad++ tab) for use in a later lab session. If you lose this password, you will have to generate a new one - there is no way to retrieve the existing one.

Click on the Repositories option in the main menu to obtain the Repositories view, and then click on the FirstRemoteRepo entry to return to the Source view with master as the current / active branch.

6   Cloning a remote repo

To clone a remote repository, we need to get the full name of the remote repo to be used with the Git clone command.

Select Clone button in the upper right hand corner of the Source view, and copy the command in the dialog box that appears.
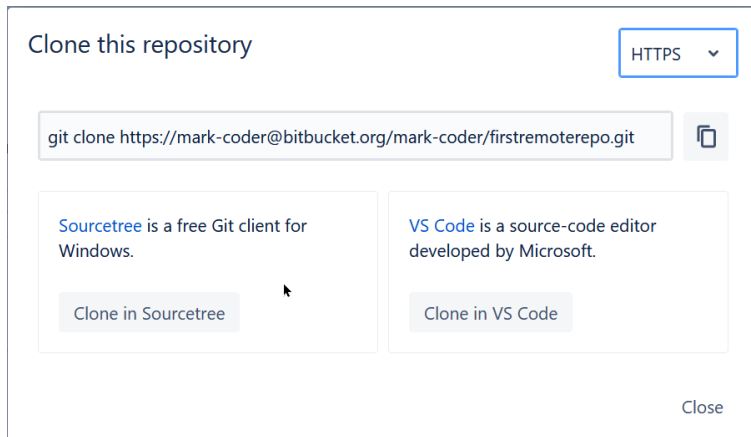
Return to the Git Bash shell that is currently open in `G:\labs`. Paste this command into the shell. You may get a dialog box that looks like the one below:
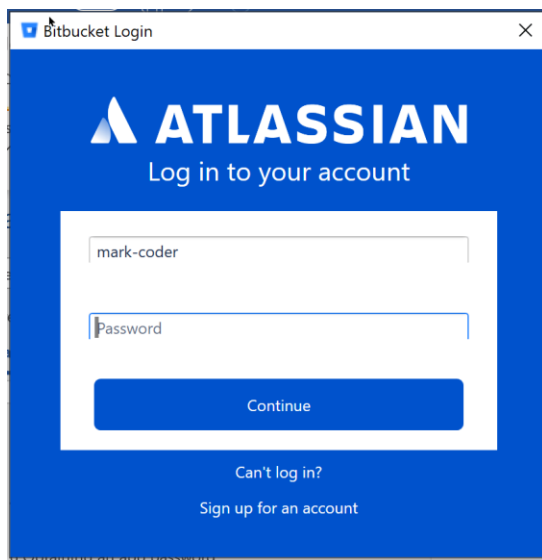
This is the primary authentication mechanism for a BitBucket account, which is no longer actively supported. Close this dialog box

An OpenSSH dialog box next appears, prompting you for the password for the specific BitBucket account. Enter the app password that you created and saved from a previous lab session here.
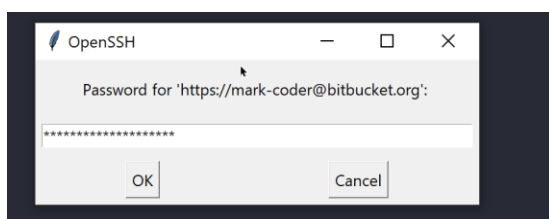
If the password is correct, you should see some messages indicating successful downloading (cloning) of the remote repo to a local repo in a directory with the same name as the remote repo: `firstremoterepo`

```
Cloning into 'firstremoterepo'...
remote: Enumerating objects: 13, done.
remote: Counting objects: 100% (13/13), done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 13 (delta 3), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (13/13), 2.45 KiB | 86.00 KiB/s, done.
```

This new directory is called a working copy (also development copy) of the project folder for that remote repository, and it contains the standard `.git directory` holding the various Git objects of the repository.

7   Getting information on the remote repo

All remote repos are given a short handle as a shortcut reference to their complete URL. The remote repo that we clone from is given the default short handle of `origin`. Remote tracking branches (`origin/xxxx`) are created for all the branches in that remote repo (the remote / upstream branches) when that repo is cloned. These are references to the remote / upstream branches which are stored in the local repo and updated by Git appropriately (for e.g. when a fetch or pull is performed) to reflect the latest state of the remote repo.

In the Git Bash shell, move into the project folder and check the short handle name for the remote repo that the project was cloned from:

```
$ cd firstremoterepo
$ git remote

origin
```

We can also check the short handle names for all remote repos as well as their matching URLs for performing push and fetch operations.

```
$ git remote --verbose

origin              https://mark-coder@bitbucket.org/mark-coder/
firstremoterepo.git (fetch)
origin              https://mark-coder@bitbucket.org/mark-coder/
firstremoterepo.git (push)
```

In the event, there is more than one remote repo associated with a local repo, the convention is to use the handle `origin` for the primary remote repo. For most scenarios, we will typically only have one remote repo associated with a local repo, which is the remote repo that the local repo was cloned from.

To get more detailed info on the remote tracking branches in the local repo and the relationship between the local branches and upstream branches, type:

```
$ git remote show origin

* remote origin
  Fetch          URL:          https://mark-coder@bitbucket.org/mark-
coder/firstremoterepo.git
  Push           URL:          https://mark-coder@bitbucket.org/mark-
coder/firstremoterepo.git
  HEAD branch: master
  Remote branches:
    feature/new-jobs tracked
    master           tracked
  Local branch configured for 'git pull':
    master merges with remote master
  Local ref configured for 'git push':
    master pushes to master (up to date)
```

Here we can see that there are remote tracking branches for the all the remote branches.
```
  Remote branches:
    feature/new-jobs tracked
    master           tracked
```

Also we see that there is a direct tracking relationship between the local master branch and the upstream master branch. This means that we can execute specific commands such as push and pull directly without explicitly specifying the remote branches involved. The local branch will be known as a tracking branch and is said to track the remote or upstream branch, with both branches typically having identical names (this is not compulsory, but it simplifies operations in many situations).

```
  Local branch configured for 'git pull':
    master merges with remote master
  Local ref configured for 'git push':
    master pushes to master (up to date)
```

To get the specific names of the remote tracking branches, we can type:

```
$ git branch --remotes

  origin/HEAD -> origin/master
  origin/feature/new-jobs
  origin/master
```

The standard format for a remote tracking branch name is *repo-handle-name/branchname*
Here we can see that the remote HEAD is pointing to the remote master branch, which makes this the current / active branch on the remote repo.

To get info on all branches (both local and remote tracking branches), we can type:

```
$ git branch --all

* master
  remotes/origin/HEAD -> origin/master
  remotes/origin/feature/new-jobs
  remotes/origin/master
```

Notice that here is only one local branch master, which is the current / active branch in the local repo. The remote tracking branches are all prefixed with the keyword `remotes`. Although there is a remote tracking branch for the upstream branch `feature/new-jobs`, this branch is not actually duplicated locally. This is because when a remote repo is cloned, by default only a single local branch is created to match the content of the current / active branch in the remote repo (typically `master`) and a tracking relationship is established between these two (`master` and `origin/master`).

Check the local project folder to verify that there is only a `countries.txt` inside it and no `jobs.txt` that was created within the `feature/new-jobs` branch in the earlier lab session.

If the remote repo has additional branches beyond the default `master` branch, we can create local branches with identical names and content that track these branches by simply performing a checkout on that branch.

```
$ git checkout feature/new-jobs

Switched to a new branch 'feature/new-jobs'
Branch 'feature/new-jobs' set up to track remote branch 'feature/new-jobs' from 'origin'.
```

Verify now that the project folder now contains the `countries.txt` and that the current active branch in the local repo is `feature/new-jobs`

There may be cases you may wish to duplicate a branch other than the default upstream `master` in the initial clone operation. In that case, you can specify that branch in the initial `clone` command, for e.g.
`git clone --branch feature/new-jobs repo-URL`
This is equivalent to doing a standard clone, and then immediately performing a `git checkout feature/new-jobs` in the newly created repo, as we have just done.

In some cases, the remote repo that you have just cloned may have many upstream branches, and it may be a hassle to duplicate all of them locally with multiple git checkout commands. In that case, we can choose to clone the entire content of the remote repo to the local repo, which includes all tags and upstream branches using this version of the `clone` command
`git clone --mirror repo-URL`

We can view the commit history in the usual way:

```
$ git log --oneline

0734495 (HEAD -> feature/new-jobs, origin/feature/new-jobs) Cool software jobs
3ee61c1 (origin/master, origin/HEAD, master) Added more content to countries
23ffb23 Initialized countries with some content
19807af Initial commit
```

The listing shows that
- Both the local master and upstream master (based on the remote tracking branches) are referencing the commit `3ee61c1`
- Also, the remote HEAD is referencing the remote master at this commit as well, indicating that this is the current / active branch on the remote repo.

- The local HEAD is referencing `feature/new-jobs` which is pointing to the most recent commit in the history (thus making `feature/new-jobs` the current / active branch in the local repo), and this is also where the equivalent upstream is pointing to as well. We say that both upstream and local branches are up to date with each other.

In the browser, navigate to the Commits view and select `feature/new-jobs` from the branch drop-down list and verify that all the commit hashes / messages shown in the listing match the commit history from the local `git log` command.

We can also verify that both the local and upstream `feature/new-jobs` are identical (or up to date) with each other at this point of time with:

```
$ git status

On branch feature/new-jobs
Your branch is up to date with 'origin/feature/new-jobs'.

nothing to commit, working tree clean
```

8 Making local commits and pushing to the remote repo

Add the following content to the end of `jobs.txt`, ensuring you leave a new line at the bottom:

```
salesman
receptionist
```

Add in this new content into a commit with:

```
$ git commit -am "Added new jobs locally"
```

Check the status with:

```
$ git status

On branch feature/new-jobs
Your branch is ahead of 'origin/feature/new-jobs' by 1 commit.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
```

We are informed that the current branch has a new commit that is not currently present in its upstream counterpart. This is based on the latest info in the remote tracking branch `origin/feature/new-jobs`. In the current sequence of labs, we assume that there is only one dev working on both the remote and local repos, so this status report is guaranteed to be correct. For a more typical collaboration scenario where another dev could potentially update the upstream `feature/new-jobs` while the current dev is making changes to the local branch, we would typically execute a `git fetch` to update all the local remote tracking branches before checking on the status.

Since the local `feature/new-jobs` is tracking its remote counterpart, we can directly upload the latest commit with a single command:

```
$ git push
```

Here, you will be prompted again for the app password in the Open SSH dialog box that we saw earlier. Once the correct password has been entered, a series of messages will appear indicating the upload was successful.

```
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 24 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 312 bytes | 312.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
…
…
```

In a push operation, what we are really doing is merging the latest content from the local branch into the upstream branch (technically speaking, both of these are 2 different branches). In other words, Git performs a: `git merge upstream-branch local-branch`

Here, we have a fast forward merge, since there is no divergent history between the upstream and local branches being merged. This simply uploads the latest commits from the local branch into the local repo and then moves the upstream branch to point at the same commit as the local branch.
In the case of a divergent history between both branches, Git will need to perform a 3 way merge which will result in an additional special commit (the merge commit). If there is any conflict, the merge conflict will also need to be resolved manually, as we will see in an upcoming lab.

With this push operation complete, the remote tracking branch `origin/feature/new-jobs` is also automatically updated as well. Verify this with:

```
$ git status
```

```
On branch feature/new-jobs
Your branch is up to date with 'origin/feature/new-jobs'.

nothing to commit, working tree clean
```

Back in the browser, go the Source view and select the `feature/new-jobs` branch and click on the `jobs.txt` to view it and confirm the latest changes that we added locally.

You can also check the commit history in the Commits view. Notice that the author for the commit is the values that you had set in an earlier lab using the `git config --global user.name` command. If this was a randomly created name (e.g. Peter Parker), BitBucket will report that it is not able to match it to valid BitBucket account (which it will try to do by default). We will switch to the same account name locally in the upcoming lab, but for now let's create another new commit in the `master` branch by switching over to it first:

```
$ git checkout master
```

```
Switched to branch 'master'
Your branch is up to date with 'origin/master'.
```

Add the following content to the end of `countries.txt`, ensuring you leave a new line at the bottom:

```
Vietnam
Philippines
```

Add in this new content into a commit with:

```
$ git commit -am "Added new countries locally"
```

Check the status again with:

```
$ git status

On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
```

Upload the new content again with:

```
$ git push
```

Again, you will be prompted again for the app password in the Open SSH dialog box and once the correct password is entered, another series of messages will appear indicating the upload was successful.

Back in the browser, go the Source view and select the `master` branch and click on the `countries.txt` to view it and confirm the latest changes that we added locally.

You can also check the commit history in the Commits view. If you view the commits from all branches, you should see a divergent line in a different color (typically red) indicating the commits from the `feature/new-jobs` branch alongside the `master` branch.

Author	Commit
Peter Parker	afcd09d
Peter Parker	136ec22
Mark Learner	0734495
Mark Learner	3ee61c1
Mark Learner	23ffb23
Mark Learner	19807af

Notice that in both changes that we have just made recently, we can simply use `git push` to upload the changes directly because there is already an existing tracking relationship between the local `master` and `feature/new-jobs` branches and their upstream counterparts, and also because the active / current branch is the one whose commits are to to be uploaded.

9   Caching / clearing app password

Currently, we have to enter the app password to authenticate our requests every time we initiate an operation (such as push, clone, etc) to a BitBucket remote repo. As a shortcut, we can choose to cache the password using the local credential manager for our system. Type the appropriate command for your system:

For Linux:

```
$ git config --global credential.helper cache
```

For Windows (using Git Bash)

```
$ git config --global credential.helper wincred
```

If you subsequently acquire a new app password for a different BitBucket account, you may need to undo the caching of your previous username/password pair with:

```
git config --global --unset credential.helper
git config --global credential.helper ""
```

At the same time, you may need to remove the previously retained credentials. On Windows, this can be done as follows:

Go to: `Control Panel -> User Accounts -> Manage Windows Credentials`. In the area `Windows  Credentials`, there are some credentials related to BitBucket. Highlight all of them and click remove.

We can also set the local user name and email to match the account details for the current BitBucket account that is hosting the remote repo that we are working with. You can obtain these details by clicking on the account Avatar in the upper right hand corner:
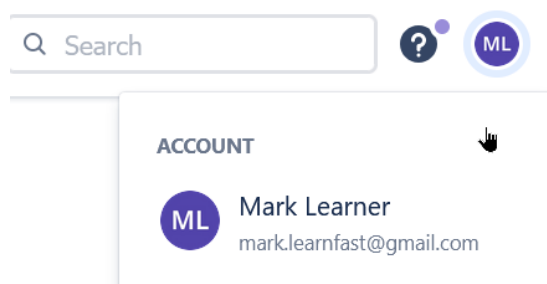

Back in the Git Bash shell, change the user.name and user.email properties with:

```
$ git config --global user.name "BitBucket account name"
```

```
$ git config --global user.email "BitBucket account email"
```

10 Fetching new branches from remote to local repo

Back in the remote repo, switch to the Branches view and click on the Create Branch button. In the Dialog box, create a new feature branch named: `new-fruits` which starts from `master`
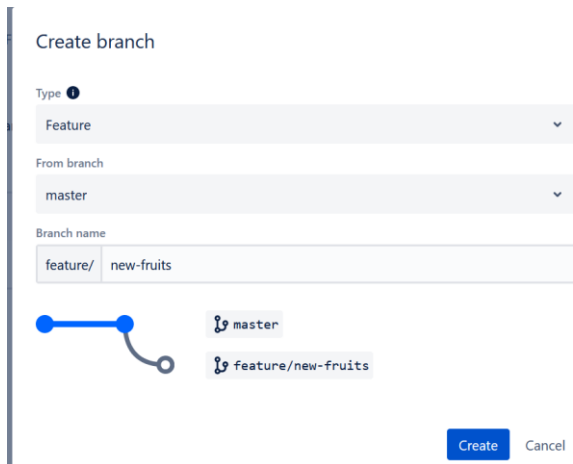
In the view for this newly created branch, select View Source.

Let's proceed to add a new file to this branch, add some content to it and commit these changes. Select Add file from the vertical three dots menu option.
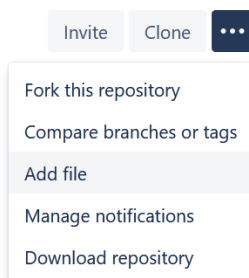
At the top, give the file the name: `fruits.txt`

The file view should clearly indicate you are in the `feature/new-fruits` branch. In the file edit tab, enter this new content, making sure to include an empty line at the bottom:

```
apple
orange
```

Click Commit on the lower left hand corner to save the changes into a commit on the current branch (`feature/new-fruits`). Enter this for your commit message in the dialog box that appears, then click Commit:

```
Some new fruits initialized
```

You are again transitioned to a commit page where you can view a variety of information regarding this newly created commit (the commit hash, the branch the commit is on, the commit message and a diff between the current commit content and the previous commit in the commit history timeline).

In the Git Bash shell, type:

```
$ git branch --all

  feature/new-jobs
* master
  remotes/origin/HEAD -> origin/master
  remotes/origin/feature/new-jobs
  remotes/origin/master
```

Notice that there is no indication of a remote tracking branch for the newly created upstream branch. This is because the local repo is not automatically synchronized to developments in the remote repo. We need to explicitly get information regarding the latest state of the remote repo through:

```
$ git fetch

remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 279 bytes | 39.00 KiB/s, done.
From https://bitbucket.org/mark-coder/firstremoterepo
 * [new branch]      feature/new-fruits -> origin/feature/new-fruits
```

The last message indicates that the existence of a new upstream branch, and the creation of a remote tracking branch corresponding to it (`origin/feature/new-fruits`)

This is a very important point to bear in mind, particularly when working in a typical team collaboration scenario. Here, a team member can potentially update any branch in a shared remote repo without the knowledge of the other team members. It is therefore useful in this kind of scenarios to execute `git fetch` periodically on a local repo to retrieve the latest status of the remote repo.

Now if we check all the branches available on the local repo with:

```
$ git branch --all

  feature/new-jobs
* master
  remotes/origin/HEAD -> origin/master
  remotes/origin/feature/new-fruits
  remotes/origin/feature/new-jobs
  remotes/origin/master
```

You will notice that there is now a remote tracking branch for the new upstream branch (`origin/feature/new-fruits`), but there isn't a local counterpart for it yet. To create a new local branch that tracks the new upstream branch, we simply switch to it with:

```
$ git checkout feature/new-fruits
```

```
Switched to a new branch 'feature/new-fruits'
Branch  'feature/new-fruits'  set  up  to  track  remote  branch
'feature/new-fruits' from 'origin'.
```

Now if we check again on all the branches available on the local repo with:

```
$ git branch --all

* feature/new-fruits
  feature/new-jobs
  master
  remotes/origin/HEAD -> origin/master
  remotes/origin/feature/new-fruits
  remotes/origin/feature/new-jobs
  remotes/origin/master
```

We should now see that we now have a local counterpart to the new upstream branch with an identical name, and this is now the active / current branch locally. You should also be able to see `fruits.txt` in the project folder with its initial content.

11 Merging changes from upstream to local branches

In the browser, return back to the Source view for the `master` branch, and select `countries.txt`. Click on Edit and add this additional content, making sure to include an empty line at the bottom:

```
Cambodia
Myanmar
```

Click Commit on the lower left hand corner to save the changes into a commit on the current branch (master). Enter this for your commit message in the dialog box that appears, then click Commit:

```
More ASEAN countries added
```

You are now transitioned to a commit page where you can view a variety of information regarding this newly created commit (the commit hash, the branch the commit is on, the commit message and a diff between the current commit content and the previous commit in the commit history timeline).

In the Git Bash shell, type:

```
$ git checkout master

Switched to branch 'master'
Your branch is up to date with 'origin/master'.
```

Notice that Git reports that the local `master` branch is up to date with its upstream counterpart, even though a new commit has just been added. Again, this is because we will not have the latest status of the remote repo until we execute:

```
$ git fetch

remote: Enumerating objects: 5, done.
```

```
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 313 bytes | 52.00 KiB/s, done.
From https://bitbucket.org/mark-coder/firstremoterepo
   afcd09d..ae00b48  master      -> origin/master
```

The last statement in the series of messages that appear indicate that one or more new commits have being added to the upstream `master`, which are now updated to the remote tracking branch `origin/master`

Now if we check on the status again with:

```
$ git status

On branch master
Your branch is behind 'origin/master' by 1 commit, and can be fast-
forwarded.
  (use "git pull" to update your local branch)

nothing to commit, working tree clean
```

we are informed that our local branch is outdated by 1 commit and that we can make it equivalent to the upstream branch with a `git pull` operation. This operation is equivalent to combining a git fetch (which we have already one) and a `git merge` *local-branch upstream-branch* together. The merge that is performed here is the exact counterpart of the merge that we perform when we execute `git push`, except now we are merging the latest updated content from the upstream branch into the local branch. We will perform the merge ourselves explicitly now and demonstrate a `git pull` later on.

Before performing an explicit merge, we can inspect the latest updates to the remote `master` in several ways. The first approach is to switch to the remote tracking branch for this branch with:

```
$ git checkout origin/master

Note: switching to 'origin/master'.

You are in 'detached HEAD' state. ….
```

Two key points to note here:

- Switching to the remote tracking branch moves the HEAD pointer to a detached state. This is because the remote tracking branch is not a real local branch, it is simply a pointer to an actual upstream branch in the remote repo. The prefix in the Git Bash prompt should now change to the commit hash of the commit that HEAD is pointing to, which is the latest commit in the upstream `master`.
- Related to this, we should NOT make any changes (for e.g. adding new commits) while in this state, as this will not have any effect for the same reasons mentioned earlier. What we can do is to inspect the commit history for the actual upstream branch that is referenced by this remote tracking branch.

Let's do that now with:

```
$ git log --oneline

ae00b48 (HEAD, origin/master, origin/HEAD) More ASEAN countries added
afcd09d (master) Added new countries locally
3ee61c1 Added more content to countries
23ffb23 Initialized countries with some content
19807af Initial commit
```

Verify that this is the same as the commit history shown in the Commits view for the `master` branch in the browser:

Mark Learner / CoolRemoteGitProject / FirstRemoteRepo

Commits

Author	Commit	Message
ML Mark Learner	ae00b48	More ASEAN countries added
Peter Parker	afcd09d	Added new countries locally
ML Mark Learner	3ee61c1	Added more content to countries
ML Mark Learner	23ffb23	Initialized countries with some content
ML Mark Learner	19807af	Initial commit

Another way to inspect the latest changes from the upstream branch before merging them into the local branch is to do a diff between both branches:

```
$ git checkout master

$ git diff master origin/master

diff --git a/countries.txt b/countries.txt
index f3f453e..540380a 100644
--- a/countries.txt
+++ b/countries.txt
@@ -4,3 +4,5 @@ Brunei
 Thailand
 Vietnam
 Philippines
+Cambodia
+Myanmar
```

Once we have inspected the commit history from the upstream branch as well as the latest changes using these two approaches, we can then make a decision on whether or not to merge in this latest changes into the local branch. If we wish to do this, we simply use the merge command in the usual way that we have seen in a previous lab:

```
$ git merge origin/master

Updating afcd09d..ae00b48
Fast-forward
 countries.txt | 2 ++
```

```
 1 file changed, 2 insertions(+)
```

As we can see, this is a simple fast forward merge as there is no divergent history between the upstream and local branches being merged (just like in the case of `git push` in an earlier lab). This simply downloads the latest commit(s) from the upstream `master` branch into the local repo and then moves the local `master` branch to point at these most recent commit in the sequence of new commits.

At the conclusion of the merge operation, its always good to double check the status to ensure that both the local and upstream branches are completely in synch with each other:

```
$ git status

On branch master
Your branch is up to date with 'origin/master'.

nothing to commit, working tree clean
```

Let's repeat this example but this time using the `git pull` command, which as explained earlier, combines both the `git fetch` and `git merge` operations into a single command.

In the browser, return back to the Source view for the `feature/new-fruits` branch, and select `fruits.txt`. Click on Edit and add this additional content, making sure to include an empty line at the bottom:

```
mango
durian
```

Click Commit on the lower left hand corner to save the changes into a commit on the current branch (master). Enter this for your commit message in the dialog box that appears, then click Commit:

```
Nice Malaysian fruits added
```

You are now transitioned to a commit page where you can view a variety of information regarding this newly created commit (the commit hash, the branch the commit is on, the commit message and a diff between the current commit content and the previous commit in the commit history timeline).

In the Git Bash shell, type:

```
$ git checkout feature/new-fruits

Switched to branch 'feature/new-fruits'
Your branch is up to date with 'origin/feature/new-fruits'.
```

Again, we see that Git reports that this local branch is in synch with its remote counterpart, even though we have just added in a new commit to the upstream branch in the browser. This is because we still have not done a `git fetch` to update the latest status of the remote repo. However, if we are clear that we want to update the local branch with the latest changes from the upstream branch without inspecting it first, we can straight away issue:

```
$ git pull

remote: Enumerating objects: 5, done.
```

```
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 273 bytes | 13.00 KiB/s, done.
From https://bitbucket.org/mark-coder/firstremoterepo
   f561702..c5a4914  feature/new-fruits -> origin/feature/new-fruits
Updating f561702..c5a4914
Fast-forward
 fruits.txt | 2 ++
 1 file changed, 2 insertions(+)
```

As explained earlier, `git pull` essentially combines `git fetch` followed by a `git merge` between the local and upstream branches involved. Note that we could also have performed a `git fetch` first to inspect the latest changes in the upstream branch, followed by `git pull` to merge them into a local branch (there is no problem with executing multiple redundant `git fetch` operations). In fact, this is the most common approach to downloading new content from the remote repo to the local repo.

Finally, we again double check the status to ensure that both the local and upstream branches are completely in synch with each other:

```
$ git status

On branch feature/new-fruits
Your branch is up to date with 'origin/feature/new-fruits'.

nothing to commit, working tree clean
```

At this point, it is good to get an overall view of the remote tracking branches that we have in place as well as the tracking relationship between the various local branches and their upstream counterparts:

```
$ git remote show origin

* remote origin
  Fetch         URL:        https://mark-coder@bitbucket.org/mark-
coder/firstremoterepo.git
  Push          URL:        https://mark-coder@bitbucket.org/mark-
coder/firstremoterepo.git
  HEAD branch: master
  Remote branches:
    feature/new-fruits tracked
    feature/new-jobs   tracked
    master             tracked
  Local branches configured for 'git pull':
    feature/new-fruits merges with remote feature/new-fruits
    feature/new-jobs   merges with remote feature/new-jobs
    master             merges with remote master
  Local refs configured for 'git push':
    feature/new-fruits pushes to feature/new-fruits (up to date)
    feature/new-jobs   pushes to feature/new-jobs   (up to date)
    master             pushes to master             (up to date)
```

Here, we can see that all 3 branches (`feature/new-fruits, feature/new-jobs, master`) have the local branches fully tracking their upstream counterparts. As explained earlier, this simply means that if that given local branch is the active / current branch, then simply running `git pull` or `git push` will either download or upload the latest changes to its respective upstream counterpart. Without a full tracking relationship, we would need to explicitly specify the upstream branch name when we execute either the `git pull` or `git push` command

12 Pushing a new local branch to the remote repo

We will create a new local branch, which we will classify as a bugfix (to distinguish it from the previous 2 branches that we classified as feature):

```
$ git checkout master

Switched to branch 'master'
Your branch is up to date with 'origin/master'.

$ git checkout -b bugfix/new-countries

Switched to a new branch 'bugfix/new-countries'
```

Add the following content to the end of `countries.txt`, ensuring you leave a new line at the bottom:

```
Australia
New Zealand
```

Add in this new content into a commit with:

```
$ git commit -am "Some new Western countries"
```

Check the status with:

```
$ git status

On branch bugfix/new-countries
nothing to commit, working tree clean
```

Notice that there is no information regarding a remote tracking branch. This is because a remote tracking branch will not be created until we push this new branch to the remote repo. We can verify this with:

```
$ git branch --remotes

  origin/HEAD -> origin/master
  origin/feature/new-fruits
  origin/feature/new-jobs
  origin/master
```

If we check the Branches view in the browser for all branches, we also will not see this newly created local branch yet.

Let's try to directly push this new local branch to the remote repo with:

```
$ git push

fatal: The current branch bugfix/new-countries has no upstream branch.
To push the current branch and set the remote as upstream, use

    git push --set-upstream origin bugfix/new-countries
```

Notice the error message that we get back. To push a new local branch to a remote repo and establish a remote tracking branch for it, we need to type:

```
$ git push --set-upstream origin bugfix/new-countries

Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
….
….
To https://bitbucket.org/mark-coder/firstremoterepo.git
 * [new branch]      bugfix/new-countries -> bugfix/new-countries
Branch 'bugfix/new-countries' set up to track remote branch
'bugfix/new-countries' from 'origin'.
```
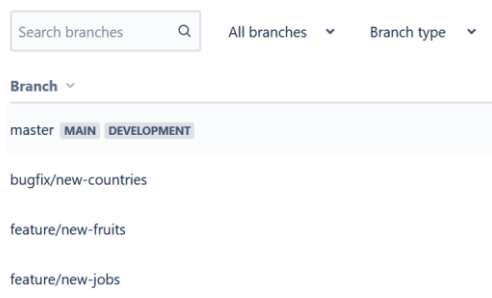
Head back to the browser and check the Branches view and list all branches to verify that a new upstream branch has been created.

Switch to the Source view, switch to the new `bugfix/new-countries` branch and click on `countries.txt` to verify that it contains the latest content that we added locally.

Now if we check the list of remote tracking branches again with:

```
$ git branch --remotes

  origin/HEAD -> origin/master
  origin/bugfix/new-countries
  origin/feature/new-fruits
  origin/feature/new-jobs
  origin/master
```

we now see that a new remote tracking branch (`origin/bugfix/new-countries`) has been created for this new local branch since there is now a new upstream branch for it.

13 Resolving merge conflicts in pushing and pulling changes

So far, all the merging of content that resulted from a `git push` and `git pull` have been simple fast forward merges which do not require user intervention. However, as we have seen before in a previous lab where we studied merging in detail, occasionally merge conflicts will occur which will then require manual resolution. We will see how to repeat this process here.

In the browser, return back to the Source view for the `bugfix/new-countries` branch, and select `countries.txt`. Click on Edit and add this additional content, making sure to include an empty line at the bottom:

```
Egypt
Kuwait
```

Click Commit on the lower left hand corner to save the changes into a commit on the current branch (master). Enter this for your commit message in the dialog box that appears, then click Commit:

```
Some really hot countries !
```

You are now transitioned to a commit page where you can view a variety of information regarding this newly created commit (the commit hash, the branch the commit is on, the commit message and a diff between the current commit content and the previous commit in the commit history timeline).

Back in the Git Bash shell, we will now add a new commit with conflicting content in the same local branch:

```
$ git checkout bugfix/new-countries
```

Add the following content to the end of `countries.txt`, ensuring you leave a new line at the bottom:

```
Finland
Norway
```

Add in this new content into a commit with:

```
$ git commit -am "Some really cold countries !"
```

Check the status with:

```
$ git status

On branch bugfix/new-countries
Your branch is ahead of 'origin/bugfix/new-countries' by 1 commit.
  (use "git push" to publish your local commits)
```

```
nothing to commit, working tree clean
```

Notice that Git states that the local branch is ahead of the upstream branch by 1 commit, although both upstream and local branches each have a new commit. This is again because we have not yet executed a `git fetch` to obtain the latest state of the remote repo. Let's do this now with:

```
$ git fetch

remote: Enumerating objects: 5, done.
….
…..
From https://bitbucket.org/mark-coder/firstremoterepo
   24d987d..512d425   bugfix/new-countries  ->  origin/bugfix/new-
countries

$ git status

On branch bugfix/new-countries
Your branch and 'origin/bugfix/new-countries' have diverged,
and have 1 and 1 different commits each, respectively.
  (use "git pull" to merge the remote branch into yours)

nothing to commit, working tree clean
```

After the `git fetch`, the `git status` command reveals the correct situation with regards to the local and upstream branches. We can now go ahead and attempt to merge the upstream branch into the local branch with:

```
$ git pull

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

As expected, the merge operation fails due to conflicting content in the new commits in both the local and upstream branches.

We will resolve this manually. Open up `countries.txt` locally. You should see the source of the conflict at the bottom:

```
…
…
Australia
New Zealand
<<<<<<< HEAD
Finland
Norway
=======
Egypt
Kuwait
>>>>>>> 512d425d2315166a45132e8053aa03e7d6490621
```

The <<<<<<< indicates that the content below is from the current branch that is being merged into. In this case, it is the branch that HEAD is pointing to: the local `bugfix/new-countries`

The >>>>>>> indicates that the content below is from the branch that is being merged from. In this case, we have the hash of the latest commit from the remote `bugfix/new-countries`
The ======= is a divider between the conflicting content

Change the conflicting part to the content below and save it:

```
 Italy
 Spain
```

Next, commit the changes for this merge commit in the usual way, with an appropriate commit message:

```
$ git commit -am "Resolved hot and cold with temperate countries !"
```

Now if we check the status:

```
$ git status

On branch bugfix/new-countries
Your branch is ahead of 'origin/bugfix/new-countries' by 2 commits.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
```

we can see that we are now ahead by 2 commits in the local branch, with the addition of the newly created merge commit.

To see the complete commit history for the latest merge commit, we can type:

```
$ git log --oneline --graph

*   ea97343 (HEAD -> bugfix/new-countries) Resolved hot and cold with
temperate countries !
|\
| * 512d425 (origin/bugfix/new-countries) Some really hot countries !
* | 237881a Some really cold countries !
|/
* 24d987d Some new Western countries
* ae00b48 (origin/master, origin/HEAD, master) More ASEAN countries
added
….
….
```

Notice that the merge commit has two parent commits, as explained in detail in a previous lab.

Finally, we can push the updated local branch with its latest merge commit to update its upstream counterpart. In this case, we will be performing a simple fast forward merge, so should not be any issue

```
$ git push
```

```
Enumerating objects: 10, done.
………
……..
remote:
To https://bitbucket.org/mark-coder/firstremoterepo.git
   512d425..ea97343  bugfix/new-countries -> bugfix/new-countries
```

In the browser, navigate to the Commits view and check the commit history for `bugfix/new-countries`. Again, notice the two divergent histories for the latest merge commit.

Let's repeat the process above with a different branch: `feature/new-fruits`

In the browser, return back to the Source view for the `feature/new-fruits` branch, and select `fruits.txt`. Click on Edit and add this additional content, making sure to include an empty line at the bottom:

```
avocado
guava
```

Click Commit on the lower left hand corner to save the changes into a commit on the current branch (master). Enter this for your commit message in the dialog box that appears, then click Commit:

```
Nice fruits from Spain
```

You are now transitioned to a commit page where you can view a variety of information regarding this newly created commit (the commit hash, the branch the commit is on, the commit message and a diff between the current commit content and the previous commit in the commit history timeline).

Back in the Git Bash shell, we will now add a new commit with conflicting content in the same local branch:

```
$ git checkout feature/new-fruits
```

Add the following content to the end of `fruits.txt`, ensuring you leave a new line at the bottom:

```
pear
peach
```

Add in this new content into a commit with:

```
$ git commit -am "Nice fruits from Italy"
```

Check the status with:

```
$ git status

On branch feature/new-fruits
Your branch is ahead of 'origin/feature/new-fruits' by 1 commit.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
```

As expected, Git states that the local branch is ahead of the upstream branch by 1 commit, although both upstream and local branches each have a new commit. This is again because we have not yet executed a `git fetch` to obtain the latest state of the remote repo. Let's do this now with:

```
$ git fetch

remote: Enumerating objects: 5, done.
….
…..
From https://bitbucket.org/mark-coder/firstremoterepo
   c5a4914..6cce700  feature/new-fruits -> origin/feature/new-fruits

$ git status

On branch feature/new-fruits
Your branch and 'origin/feature/new-fruits' have diverged,
and have 1 and 1 different commits each, respectively.
  (use "git pull" to merge the remote branch into yours)

nothing to commit, working tree clean
```

After the `git fetch`, the `git status` command reveals the correct situation with regards to the local and upstream branches. Let's see what happens if we now try to push changes from the local branch to its upstream counterpart:

```
$ git push

To https://bitbucket.org/mark-coder/firstremoterepo.git
 ! [rejected]        feature/new-fruits -> feature/new-fruits (non-
fast-forward)
error: failed to push some refs to 'https://bitbucket.org/mark-
coder/firstremoterepo.git'
hint: Updates were rejected because the tip of your current branch is
behind
hint: its remote counterpart. Integrate the remote changes (e.g.
hint: 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for
details.
```

This example demonstrates a very important point: If a push operation results in a merge conflict, Git will prevent the operation from going ahead because there is no guarantee that the current user can access the remote repo to resolve the conflict properly. Instead, Git insists on a pull operation (as we

have done previously) to resolve the conflict on the local repo. In other words, if there is a merge conflict whenever we try to do a `git push`, we **MUST** resolve the conflict locally with a `git pull`.

The rest of the operations that we perform next are identical to what we did previously.

```
$ git pull

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

As expected, the merge operation fails due to conflicting content in the new commits in both the local and upstream branches.

We will resolve this manually. Open up `fruits.txt` locally. You should see the source of the conflict at the bottom:

```
…
…
durian
<<<<<<< HEAD
pear
peach
=======
avocado
guava
>>>>>>> 6cce700edc5543339a34be07f0c0c73e3b6f869a
```

The `<<<<<<<` indicates that the content below is from the current branch that is being merged into. In this case, it is the branch that HEAD is pointing to: the local `feature/new-fruits`
The `>>>>>>>` indicates that the content below is from the branch that is being merged from. In this case, we have the hash of the latest commit from the remote `feature/new-fruits`
The `=======` is a divider between the conflicting content

Change the conflicting part to the content below and save it:

```
avocado
guava
```

Next, commit the changes for this merge commit in the usual way, with an appropriate commit message:

```
$ git commit -am "Viva espanol"
```

Now if we check the status:

```
$ git status

On branch feature/new-fruits
Your branch is ahead of 'origin/feature/new-fruits' by 2 commits.
  (use "git push" to publish your local commits)
```

```
nothing to commit, working tree clean
```

we can see that we are now ahead by 2 commits in the local branch, with the addition of the newly created merge commit.

To see the complete commit history for the latest merge commit, we can type:

```
$ git log --oneline --graph

*   2ca3fac (HEAD -> feature/new-fruits)  Viva espanol
|\
| * 6cce700 (origin/feature/new-fruits) Nice fruits from Spain
* | 69679cc  Nice fruits from Italy
|/
* c5a4914 Nice Malaysian fruits added
* f561702 Some new fruits initialized
….
….
```

Notice that the merge commit has two parent commits, as explained in detail in a previous lab.

Finally, we can push the updated local branch with its latest merge commit to update its upstream counterpart. In this case, we will be performing a simple fast forward merge, so should not be any issue

```
$ git push

Enumerating objects: 10, done.
………
……..
remote:
To https://bitbucket.org/mark-coder/firstremoterepo.git
   6cce700..2ca3fac  feature/new-fruits -> feature/new-fruits
```

In the browser, navigate to the Commits view and check the commit history for `feature/new-fruits`. Again, notice the two divergent histories for the latest merge commit.

14 Merging branches into master in remote and local repo

In many development workflows, once we have completed implementation for a particular branch (feature, bugfix, hotfix, release, etc), we will integrate it into the main / master branch and then delete that branch. We will see how to do this on the remote and local repo, as well as ensuring that both repos remain in sync after these changes have been made.

We will start with the local repo.

```
$ git checkout master
```

```
$ git merge bugfix/new-countries

Updating ae00b48..ea97343
Fast-forward
 countries.txt | 4 ++++
 1 file changed, 4 insertions(+)
```

As expected, merging in the changes from the `bugfix/new-countries` branch is a straight forward process with a fast forward merge.

Check the new commit history with:

```
$ git log --oneline --graph

*     ea97343 (HEAD -> master, origin/master, origin/bugfix/new-
countries, origin/HEAD) Resolved hot and cold with temperate countries
!
|\
| * 512d425 Some really hot countries !
* | 237881a Some really cold countries !
|/
* 24d987d Some new Western countries
……
……
```

Notice the divergent commit histories starting from the most recent merge commit.

We can delete the local branch with:

```
$ git branch --delete bugfix/new-countries

Deleted branch bugfix/new-countries (was ea97343).
```

We can upload the latest updates in the `master` branch to the remote repo with:

```
$ git push
```

In the browser, navigate to the Commits view and check the commit history for `master`. Again, notice the two divergent histories for the latest merge commit.

We can delete the upstream branch with:

```
$ git push origin --delete bugfix/new-countries
```

```
To https://bitbucket.org/mark-coder/firstremoterepo.git
 - [deleted]          bugfix/new-countries
```

In the browser, navigate to the Branches view and check that this branch no longer exists.

We will work with the `feature/new-jobs` branch next. Notice at the moment that this branch is 6 commits behind and 2 commits ahead of `master` (all branches are automatically compared with the `master` branch because the expectation is that we will merge into `master` or merge from `master` at some point in the development workflow).

Then, in the same view, click on `master`. In the drop down list, select the `feature/new-jobs` branch. We now have a comparison (diff) this branch as well as an option to merge this branch into `master` (the `sync now` link).

Click on the `sync now` link. A dialog box appears. Click Sync to accept the default commit message.
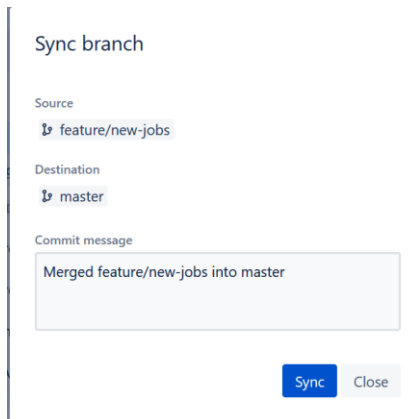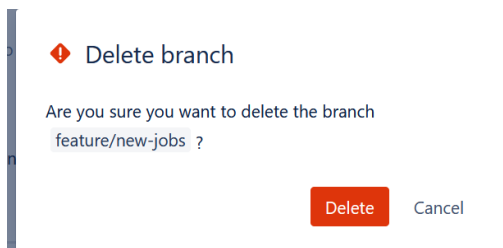
When this is complete, you can return to the Source view and check that the project folder now contains `jobs.txt` with the latest content from `feature/new-jobs`

Next, return to the Branches view and select the Delete action for the `feature/new-jobs` branch
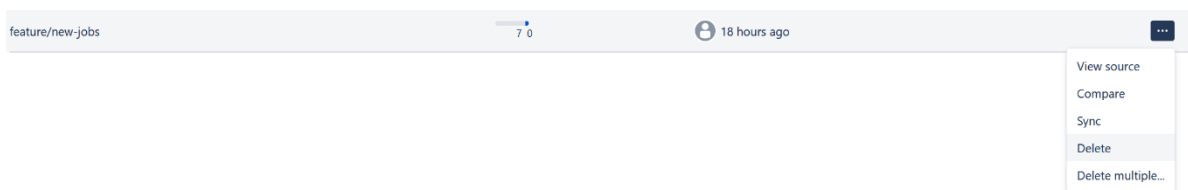
Select Delete from the pop up box that appears.

A pop up message indicating successful deletion appears in the lower left hand corner. You should no longer see this branch in the list in the main Branches view.

Back in the Git Bash shell, we can update our remote tracking branches with:

```
$ git fetch
```

We can also perform a quick diff with the latest content from the upstream `master` with:

```
$ git diff origin/master

diff --git a/jobs.txt b/jobs.txt
deleted file mode 100644
index 186e6e3..0000000
--- a/jobs.txt
+++ /dev/null
@@ -1,4 +0,0 @@
-developer
```

```
-project manager
-salesman
-receptionist
```

As expected, we see the content of the new `jobs.txt` that was added in the previous merge commit. We can now proceed to update our local master with:

```
$ git pull
```

This will be a straightforward fast forward merge.
Next, we follow up on the local `feature/new-jobs` branch

```
$ git checkout feature/new-jobs

$ git fetch

$ git status

On branch feature/new-jobs
Your branch is up to date with 'origin/feature/new-jobs'.

nothing to commit, working tree clean

$ git branch --all

  feature/new-fruits
* feature/new-jobs
  master
  remotes/origin/HEAD -> origin/master
  remotes/origin/feature/new-fruits
  remotes/origin/feature/new-jobs
  remotes/origin/master
```

Notice here that even after performing a git fetch to get the latest update on the state of the remote repo, Git still indicates our local `feature/new-jobs` is up to date with its upstream counterpart and the remote tracking branch for it (`origin/feature/new-jobs`) is still available, even though this branch has already been deleted in the remote repo.

This is one of the main drawbacks of deleting a branch via the browser UI: we lack clear indication in the local repo that a particular upstream branch has been deleted. Now if we perform a:

```
$ git remote show origin

…….
……
  HEAD branch: master
  Remote branches:
    feature/new-fruits                    tracked
    master                                tracked
    refs/remotes/origin/feature/new-jobs  stale  (use  'git  remote
prune' to remove)
  Local branches configured for 'git pull':
    feature/new-fruits merges with remote feature/new-fruits
    feature/new-jobs   merges with remote feature/new-jobs
```

```
   master               merges with remote master
  Local refs configured for 'git push':
    feature/new-fruits pushes to feature/new-fruits (up to date)
    master             pushes to master            (up to date)
```

Here, we get a clue that there is an issue with the upstream `feature/new-jobs` branch. The remote tracking branch for it (`origin/feature/new-jobs`) is shown to be stale (outdated) and there is recommendation to remove it. Let's go ahead and do that with:

```
$ git remote prune origin

Pruning origin
URL: https://mark-coder@bitbucket.org/mark-coder/firstremoterepo.git
 * [pruned] origin/feature/new-jobs
```

An alternative to `git remote prune origin` is `git fetch --prune`. This essentially gets an update on all the remote tracking branches first before removing the stale (outdated) references.

Now if we run the various pertinent commands again, we get a clear indication that the upstream counterpart of the local `feature/new-jobs` has been deleted:

```
$ git status

On branch feature/new-jobs
Your branch is based on 'origin/feature/new-jobs', but the upstream
is gone.
  (use "git branch --unset-upstream" to fixup)

nothing to commit, working tree clean

$ git branch --all

  feature/new-fruits
* feature/new-jobs
  master
  remotes/origin/HEAD -> origin/master
  remotes/origin/feature/new-fruits
  remotes/origin/master
```

Let's delete this local branch in the usual way:

```
$ git checkout master
```

```
$ git branch --delete feature/new-jobs
```

And finally if we check again with:

```
$ git remote show origin
```

There is no longer any reference to `feature/new-jobs` anywhere

So far, we have demonstrated merging the contents of a branch into `master`. Sometimes, we may also wish to merge the contents of `master` into an existing branch if we still want to continue development on that branch but we want it to be updated with the latest content from `master`.

In the browser, navigate to the Branches view and select `feature/new-fruits`. We can see it is 8 commits behind master.
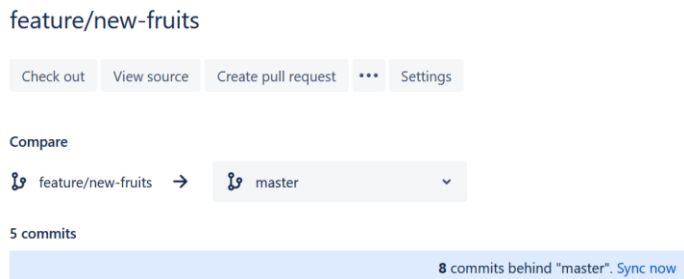
Click on Sync Now to merge the new content from master into this branch. Click on Sync in the dialog box to accept the default commit message.

You should now be able to see the new merge commit in the view. In the main branches listing, it clearly indicates that this branch is 0 commits behind master.
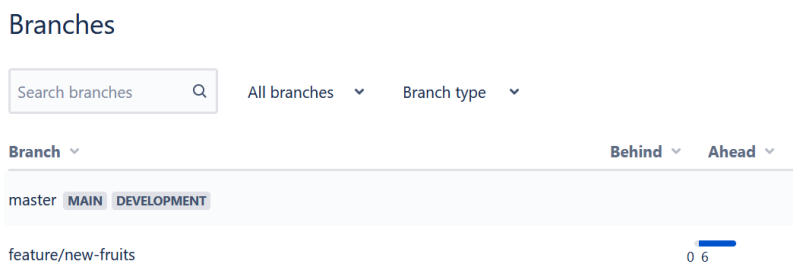
We can now continue to add content to this branch and perhaps merge it into `master` when we are complete sometime in the future.

We will synchronize the updated content here with its local counterpart as well:

```
$ git checkout feature/new-fruits

$ git fetch

$ git status

On branch feature/new-fruits
Your branch is behind 'origin/feature/new-fruits' by 9 commits, and
can be fast-forwarded.
  (use "git pull" to update your local branch)

nothing to commit, working tree clean

$ git pull

Updating 2ca3fac..ed08fff
```

```
Fast-forward
 countries.txt | 6 ++++++
 jobs.txt      | 4 ++++
 2 files changed, 10 insertions(+)
 create mode 100644 jobs.txt

$ git status

On branch feature/new-fruits
Your branch is up to date with 'origin/feature/new-fruits'.

nothing to commit, working tree clean
```

15 Publishing a local repository to a remote repository

In this lab, we started off with an empty (or nearly empty) remote repository, which team members can then clone to a local repository on their respective machines and perform their development work there. The other typical scenario is that we may have an existing local repository that was initially private that we now wish to upload to a remote repository so that can be shared with others in a collaborative team effort.

We will use the `firstlab` project folder from a previous lab to demonstrate this scenario. First, we will need to create an empty remote repository to which we can upload our existing local repository to. To keep things simple, we will create it with the same name as the local repository, although it can be different as well.

Go ahead and repeat the process of [creating a new repository](#) that we did earlier, but this time we will create a bare repository with no content to facilitate the process of pushing the contents of our local repository to it:
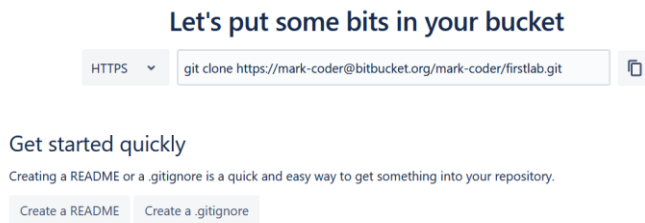
Enter the values for the following fields.
Project Name: `CoolRemoteGitProject`
Repository Name: `firstlab`

When you are done specifying the values for the fields as shown above, click Create Repository. You will be transitioned to the Source view for the newly created repo, where some instructions will be provided on how to get started.

Let's put some bits in your bucket

| HTTPS ˅ | git clone https://mark-coder@bitbucket.org/mark-coder/firstlab.git | 🗋 |

Get started quickly

Creating a README or a .gitignore is a quick and easy way to get something into your repository.

Create a README   Create a .gitignore

Click on the Clone button to copy the URL (e.g. https://xxx@bitbucket.org/yyy/firstlab.git) for this new remote repo to an empty document. We will refer to this URL as `remote-url` in the commands to follow.

Open a Git Bash shell in `firstlab`, and type:

```
$ git remote add origin remote-url
```

Check that the origin handle is set to point to the correct repo URL with:

```
$ git remote --verbose
```

Finally, push the entire contents of the local repo (including all its branches) to the remote repo with:

```
$ git push -u origin --all

Enumerating objects: 39, done.
Counting objects: 100% (39/39), done.
….
…..
* [new branch]      master -> master
Branch 'master' set up to track remote branch 'master' from 'origin'.
```

Return back to the browser and check through the Source, Commits and Branches main view to verify that these mirrors the status of the local repo that you just pushed up into it. Note that you may need to navigate out to the main Repositories tab in the main menu, and then click on the `firstlab` repo entry in order for it to refresh properly and show the uploaded content. This may take some time to complete correctly.

Back in the Git Bash shell in `firstlab`, type the following commands to verify that the remote tracking branches have been set up and that the local and upstream master are both in sync with each other.

```
$ git remote show origin
```

```
$ git status
```

Other users can now clone this remote repo in the same way that we have just demonstrated previously.