Git Lab 4
Collaborative development using 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`

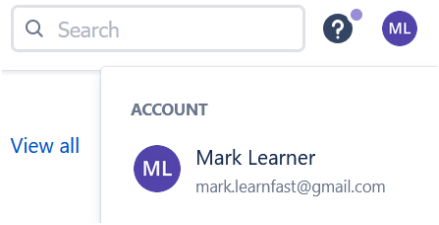
Uploading content to a remote repo	Downloading content from a remote repo
`git push` `git push --set-upstream origin` *branch-name* `git push origin --delete` *branch-name*	`git pull`

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.

You will need to have two valid BitBucket accounts which you have full access to. In the following labs, we will refer to the identities associated with these two accounts as Dev A and Dev B respectively. The actual account names and main email alias can be viewed from the drop down menu accessible from the account Avatar in the upper left hand corner



We will be working with the views for both these accounts in 2 different browsers to prevent any issues with credential / password caching that most modern browsers perform automatically in the background. You can use any standard modern browser for this purpose (Chrome, Edge and FireFox are great choices).

Create two new subfolders `devA` and `devB` in the top-level main folder `labs`. These represent the development work area for the users associated with these two accounts. We will be creating local repos as well as cloning remote repos within these two new subfolders.

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 Collaborative development workflows

Git provides a large multitude of commands and options that allow it to be used effectively for version control in a large variety of situations. This provides a lot of flexibility for users but can also significantly increase the complexity in the use of Git. To simplify and standardize the use of Git, particularly in collaborative development effort, we generally rely on the use of workflows. A workflow is a guideline or recommendation for using Git in a specific way in a software project, so that all developers in the team use Git in a consistent and uniform way.

There are many workflows possible for collaborative development of a shared code base, ranging from the basic to the very complex. Below is a basic workflow that involves synthesis of the key points from centralized and feature branch workflow:

a) Project lead / manager creates an initial software project (in the IDE / framework of choice) and creates a local repo in the root folder of this project. The first few commits of this local repo will contain changes necessary to setup the project for active development. Make sure to include a `.gitignore` file to ignore specific files that don't need to be tracked (e.g. binary executables, log files, IDE specific settings, etc)

b) Project lead / manager creates an empty (bare) remote repo on an internal organizational server or using a hosted Git cloud service such as BitBucket / GitHub. This remote repo will function as the shared central repo for the development team.

c) Project lead / manager populates the empty central repo with the content from the local repo.

d) Project lead / manager sets up user groups and permissions for appropriate access control to the shared central repo to control which users are able to read from (pull) or write to (push) it. The collaborative development workflow can now start.

e) Team members clone central repo to a local repo on their machines. Each developer works on their local repo in the usual manner: make changes to source code files, stage changes and commit to their local repo.

f) The main / master branch on this central repo represents the official production code base. This means that the code base here should be fully tested and ready for shipping release or to be deployed on a production server for consumption by clients. All development work (feature, bugfix, hotfix, etc) should be undertaken in a local branch that starts from the local main / master branch. Branches should be specific and have a very clear purpose. Commits for ongoing, non-complete development work should always be made in this branch, **NEVER** on the main / master branch in either the local or central repo.

KEY POINT: Master / main branch on central repo should ONLY contain tested and working code ready for deployment / shipping. All ongoing development / bugfix work should be in a local branch that starts from master.

g) Team should standardize on the unit of work that constitutes a commit (e.g. some subunit of overall feature working, 1 complete function / class, xxx minutes of coding time). Don't squeeze too much work or wait too long before making a commit. Make sure follow guidelines for writing a good commit message.

h) At a specific point of time, upload the branch under development to the remote repo (creating an identical upstream copy). This is useful as a backup and also provides an opportunity to other team members can also pull this branch to their local repo to inspect it and see how it might affect their own feature branches.

i) Not all local branches need to be pushed to the central repo. Examples might be private branches for experimental work that is being worked on in isolation.

j) When development in the local branch is at an appropriate stage (it could be partially or fully complete) and the latest changes have been pushed to the central repo, the developer responsible for that branch initiates a pull request.

k) The pull request officially kickstarts a code review process involving relevant team members commenting / discussing the changes on this branch. They can download the branch to their local repos to inspect / test it as part of this review. Any further changes resulting from this discussion are committed to the local branch and pushed to its upstream counterpart. All activity (feedback posted, further amendments to the feature) is tracked. These new commits will also be tracked as part of the pull request.

KEY POINT: Use the pull request to document all interaction regarding key development work (feature / bugfix) branches

l) Finally, when all issues are resolved and discussion is complete, the project lead / manager can approve the pull request for the new branch can be considered to be accepted. This implies that the branch would have undergone all appropriate testing (unit, integration, regression testing etc).

m) Project lead / manager will then integrate the remote new branch into the remote master, typically through a merge (but can also be a rebase). Most of the time, this merge itself will be a fast-forward merge, which means it can be performed automatically by Git in the remote repo.

n) When the remote master has been updated, all team members will pull these latest changes to update the local master on their respective repos. If this latest merge represents a significant point in the evolution of the production code base (e.g. release version), mark it with a tag in the local repo.

KEY POINT: Update local repo (master, and other relevant branches) after every merge operation on the shared central repo. This should be done frequently to ensure the local repo is as closely synced to the central repo as possible.

o) If members are working on a local feature / bugfix branch of their own, they will then need to merge the new content from their local master into the local branch to update it as well. This will typically be a 3 way merge that may result in a merge conflict, which will then have to be resolved with the help of other team members, if need be. This resolution can be documented through another pull request for their local branch in a similar way described.

p) If the newly merged-in branch is not required for archival, it can now (or at a later point in the project timeline) be deleted from the central repo. Team members will also delete it from their local repo as well. Branches should only be deleted once their content have being merged into remote master (or some other long-running branch). If no decision can be made on a branch, we should leave it archived in the central repo.

q) Team members assigned to work on new features will again start branches for this feature from the latest commit in the remote master and work will proceed in the usual way as described previously.

There is a list of community acknowledged best practices when working with Git that be used to augment the development workflow outlined above.

In the simple approach outlined above, all branches (feature, bugfix, hotfix, etc) are short-lived branches that are eventually integrated into the main / master branch. However, for more complex projects, you can include additional long-running branches besides main / master which represent key states in the project life cycle such as testing or development state. Just like in main / master, no

commits are placed on these branches directly, instead other feature / bugfix / etc branches are integrated into them over time.

These long running branches have will have a hierarchy: for e.g. main / master is typically the highest-order branch and it only contains working and tested code that is ready to be released and / or deployed on a production server. Below main / master, you might have a development branch in which feature branches are merged into, and then code that has passed testing from the development branch is merged into main / master. This is based on the concept of progressive stability branching - branches are at various levels of stability; when they reach a stable state, they're merged into the branch above them.

A good example of such a workflow is the GitFlow workflow, which has two other long-running branches besides main: develop and release.

In addition, there is also a Forking workflow, whereby each developer in a team has their own individual public repo as well as local repo. There is still a central official repo which holds the production codebase for the project, but only the official project lead / maintainer can push into this central official repo from the public repos of the various team members. This supports a collaborative effort on a project, while at the same time providing better security and access control compared to the simpler case where all team members have full access rights to the central repo. It is well suited for projects that involve large teams with many members (some of whom may not be full trusted) such as is the case for open source community projects.

4 Dev A creates new local repo and populates with content

Step a) in the sample collaborative development workflow

We will assume that Dev A is the project lead and is going to create a local repo and populate it with content.

Inside the `devA` folder, create a new subfolder `firstsharedrepo`

Create a file named as below and populate it as shown using a text editor. Make sure you include a new line after the end of the single line

```
1: developer
```

humans.txt

Initialize a local repo in the current directory with:

```
$ git init
```

Change the local user.name and user.email properties to match the BitBucket account details of Dev A.

```
$ git config --local user.name "BitBucket account name Dev A"

$ git config --local user.email "BitBucket account email Dev A"
```

Verify that these properties have been set correctly with:

```
$ git config --list --show-origin
```

In the list that appears, you should see these two configuration properties listed next to `file:.git/config`, which is the file holding all variables with local scope. As discussed earlier, the values here will override any values for the same variables at global (`C:/Users/UserAccount/.gitconfig`) or system scope (`C:/Program Files/Git/etc/gitconfig`)

Stage this new file with:

```
$ git add --all
```

Create the first commit with:

```
$ git commit -m "Initialized repo with single line in humans"
```

Check that the commit has been added properly:

```
$ git log
```

Create another file named as below and populate it as shown using a text editor. Make sure you include a new line after the end of the single line

```
1: cat
```

animals.txt

Stage this new file with:

```
$ git add --all
```

Create the second commit with:

```
$ git commit -m "Added animals with a single line"
```

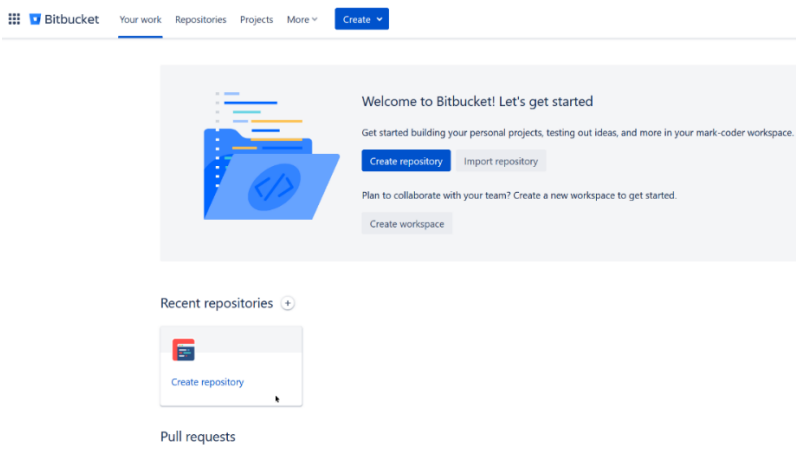
Check that the commit has been added properly:

```
$ git log
```

5 Dev A creates a bare remote repo for shared collaboration

Step b) in the sample collaborative development workflow

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

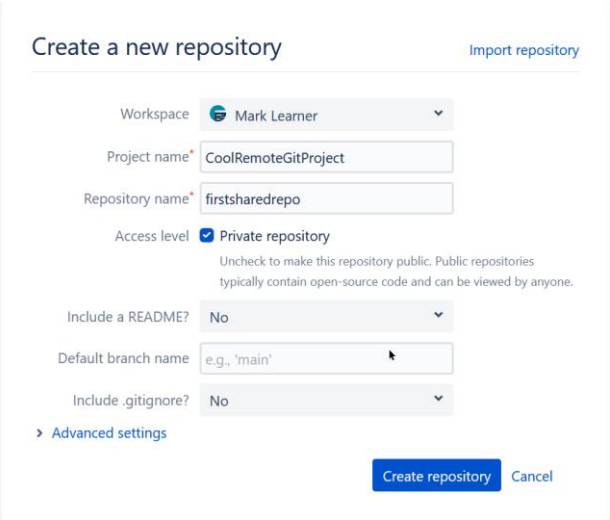


We can proceed to [create a new repository] in this account, which we will make as a bare repository by specifying No to all inclusion options provided (for a README or a .gitignore).

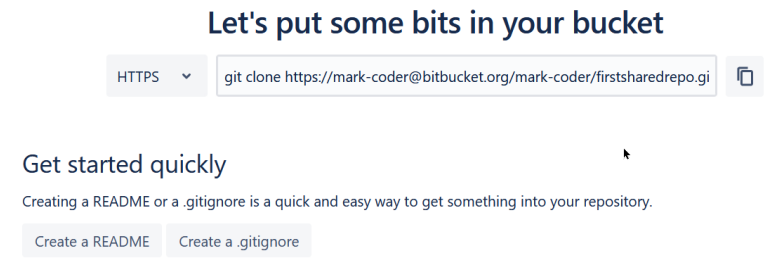
Enter the values below for the following fields. Notice that the new remote repo name is identical to the name of the newly created local repo from the previous topic. This is not compulsory, but will simplify matters when interacting with the remote repo in subsequent labs.

Project Name: `CoolRemoteGitProject`
Repository Name: `firstsharedrepo`

Ensure that the both the include options for README and .gitignore are set to No to create a bare repository.



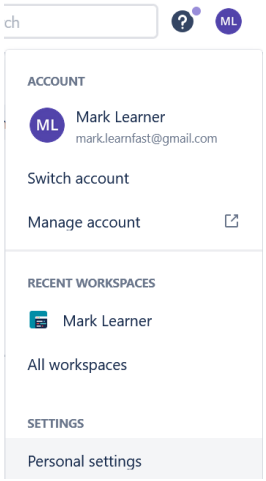
When you are done specifying the values for the fields as shown above, click Create Repository. You will be transitioned to the Source view for the newly created repo, where some instructions will be provided on how to get started.



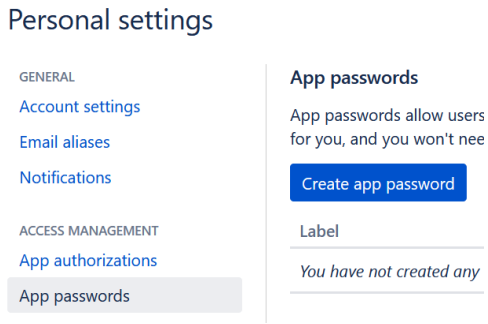
6 Dev A obtains an app password to interact with remote repo

When we use commands from Git Bash to interact with any 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.

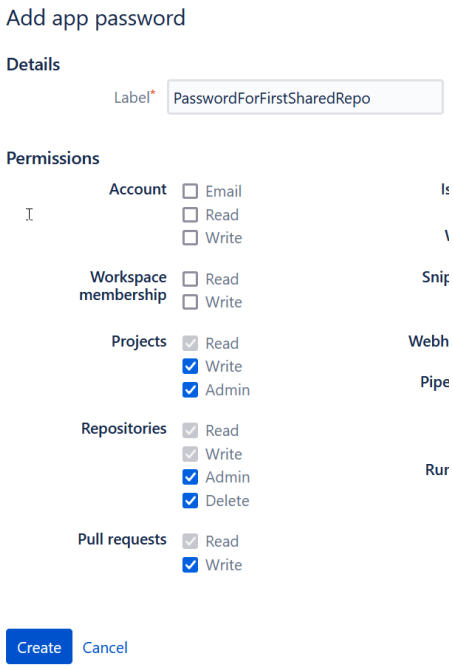


Select App passwords under Access management section on the left. Click the Create app password button

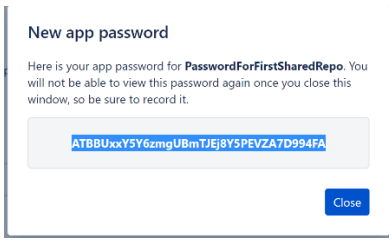


In the Add App password section, provide this info for the label: `PasswordForFirstSharedRepo`

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) and associated it with the Dev A account for use in the later lab sessions. 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 `firstsharedrepo` entry to return to the Source view.

7 Dev A pushes the content of the local repo into the empty remote repo

Step c) in the sample collaborative development workflow

Copy the URL for this new empty remote repo (e.g. `https://xxx@bitbucket.org/yyy/firstsharedrepo.git`) to an empty NotePad++ tab. We will refer to this URL as *`remote-url`* in the commands to follow.

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

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

All remote repos are given a short handle as a shortcut reference to their complete URL. The default short handle for all remote repos is `origin`.

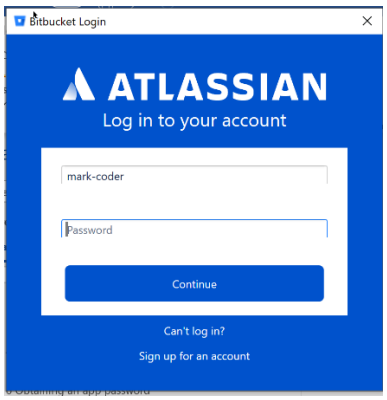
Check that the `origin` handle is set to point to the correct repo URL with:

```
$ git remote --verbose

origin  https://xxx@bitbucket.org/yyy/firstsharedrepo.git (fetch)
origin  https://xxx@bitbucket.org/yyy/firstsharedrepo.git (push)
```

Finally, push the entire contents of the local repo (which currently only has one commit) to the remote repo with:

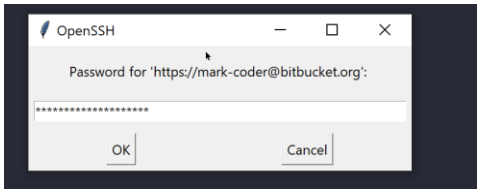
```
$ git push -u origin --all
```

At this point, your initial attempt to upload content into the empty remote repo will require authentication from the BitBucket server. Depending on the state of your BitBucket account, you may initially 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. You can safely 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.

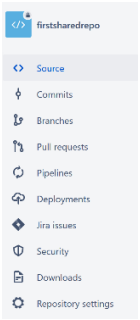


If you enter the correct password and are authenticated successfully, you should see the series of messages below in the Git Bash shell. Otherwise, if for whatever reason you are not able to authenticate, you will need to repeat the previous lab session to obtain a new app password and reuse it here again.

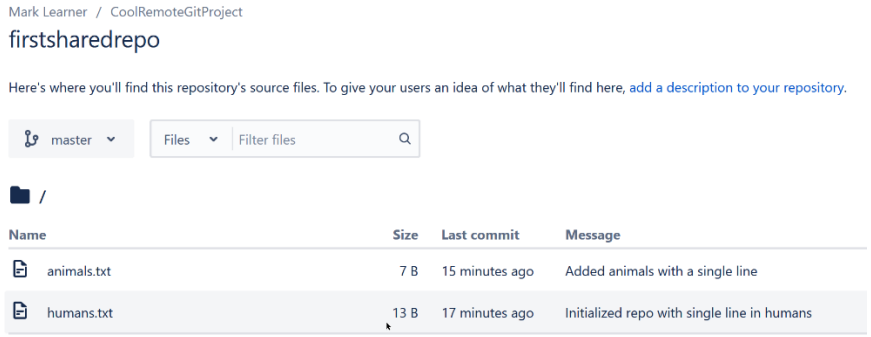
```
Enumerating objects: 6, done.
Counting objects: 100% (6/6), done.
Delta compression using up to 24 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (6/6), 525 bytes | 525.00 KiB/s, done.
Total 6 (delta 0), reused 0 (delta 0), pack-reused 0
To https://bitbucket.org/mark-coder/firstsharedrepo.git
 * [new branch]      master -> master
branch 'master' set up to track 'origin/master'.
```

8 Working with remote repo views

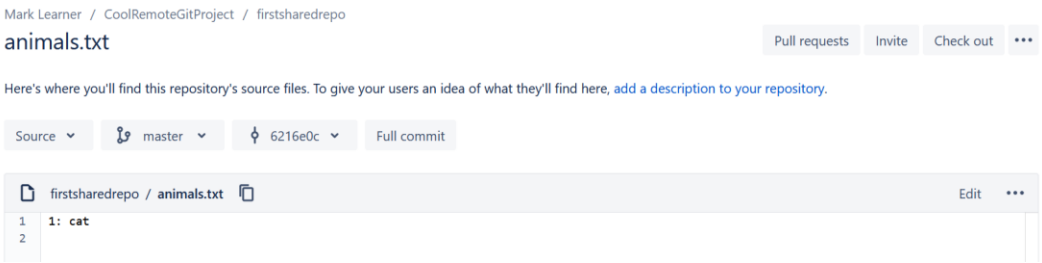
The side browser bar in the main repo view should show a variety of options:



If you refresh the browser in the Source view (or navigate away and navigate back again), you should be able to see the active branch (currently `master`) and the content of the remote repo at this branch. You will see that the content mirrors the content of the local Git repo that it was uploaded from.

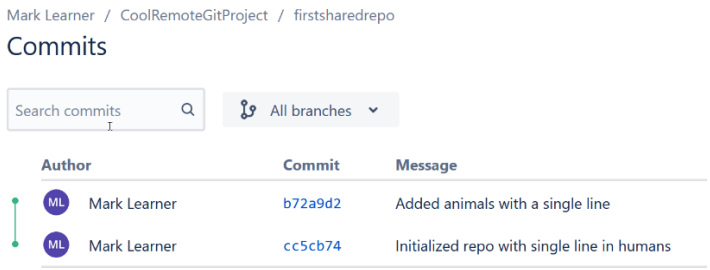


If you click on any of the files in the Source view, you will be transitioned to a view which shows its contents as well as the commit hash.



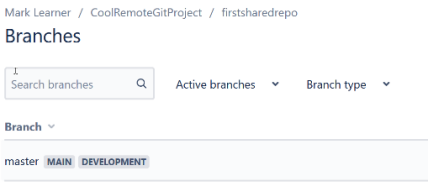
Notice that there is an option to edit the contents of that file directly through in the browser. You can do this, but the more common approach is to make changes in the file(s) concerned in the local repo, commit them and push them to the remote repo as we will see later.

Clicking on the Commits option shows all the commits in all branches in the repo. At the moment, there are only 2.



Notice that the commit hashes are identical to the ones on the local repo (verify for yourself with a `git log`), which is expected as the remote repo is currently simply the uploaded contents from the local repo.

Clicking on the Branches view shows you all the branches available in the repo, and at the moment there is only one (master) which is also marked out as the main development branch. Remember that the master / main branch on central shared repo should ONLY contain production code (that is tested and working code ready for deployment / shipping).



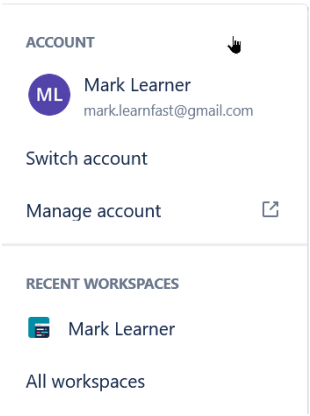
The Branches, Source and Commits view are the 3 main views you will be working frequently with in the duration of a collaborative project.

9 Dev A creates user group to include team members

Step d) in the sample collaborative development workflow

This remote repo will act as the central repo for collaborative teamwork on its codebase. Since Dev A hosts this shared repo in their account, they will therefore be in charge of all relevant admin activities on this repo (although they can also grant access to other team members to perform this on their behalf). The rest of the team members (at the moment this is only a single individual, Dev B - but in practice, this can be any number of other individuals with valid BitBucket accounts) will be enrolled in a user group that has appropriate access rights to this shared repo.

BitBucket uses the concept of workspaces to organize repos and different streams of work in a valid BitBucket Cloud account. Each account has a default workspace whose name is the same as the account name - you can click on the profile avatar to see this in the drop down menu.



You can create a new workspace if you wish, but we will work with the existing one for this simple example.

A workspace can have one or more members (which are users that have valid BitBucket accounts), and these members are organized into user groups. Each user group can be assigned a set of permissions with regards to repository access and workspace access. Any member within a workspace group will automatically have the permissions associated with that group.

The repository permissions listed from the highest level of access to the lowest:

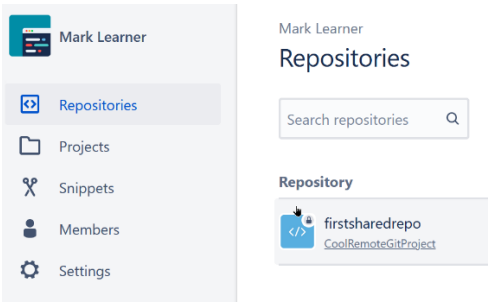
Admin	Allows users to do everything a repository owner can do: change repository settings, update user permissions, and delete the repository.
Write	Allows users to contribute to the repository by pushing changes directly.
Read	Allows users to view, clone, and fork the repository code but not push changes. Read access also allows users to create issues, comment on issues, and edit wiki pages.
None	Prevents those users from seeing anything in the repository.

The workspace access permissions are:

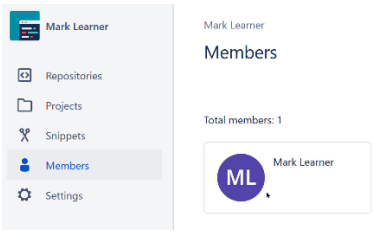
Create repositories	Allows workspace members to create new repositories for the team.
Administer workspace	Allows workspace members to update the workspace's settings and the settings of any repositories within the workspace.

To view existing user groups as well as add new ones, we need to go to the workspace settings page for a selected workspace.

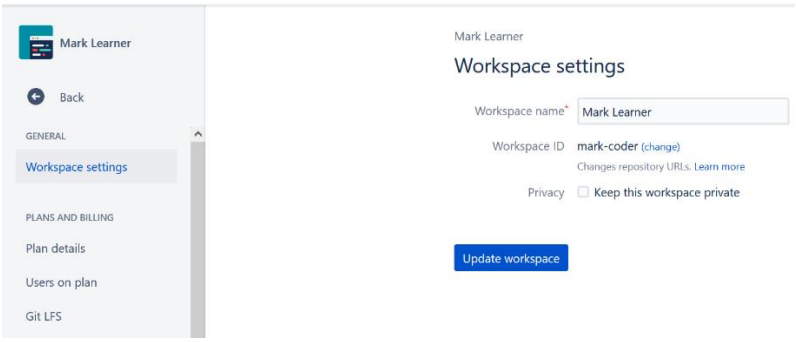
Click on the profile avatar, and in the drop-down menu select the single entry below the RECENT WORKSPACES which is the default and current workspace (it should be the same as the account name of Dev A). You will be provided with an overview page of that workspace.



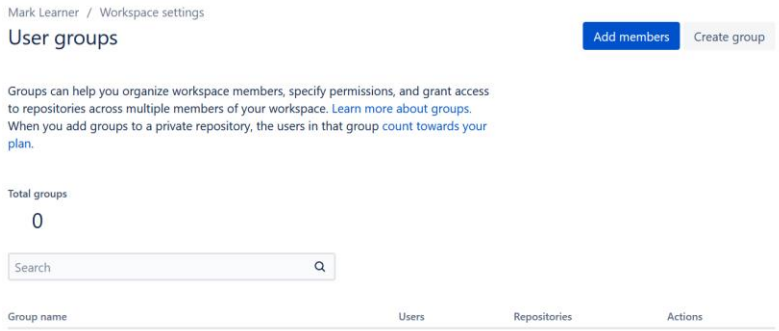
You should be able to see all the repos created in this workspace in the main listing.
Clicking on Members in the left pane will show you the members current associated with this workspace: at the moment, there is only 1 (the account name for Dev A).



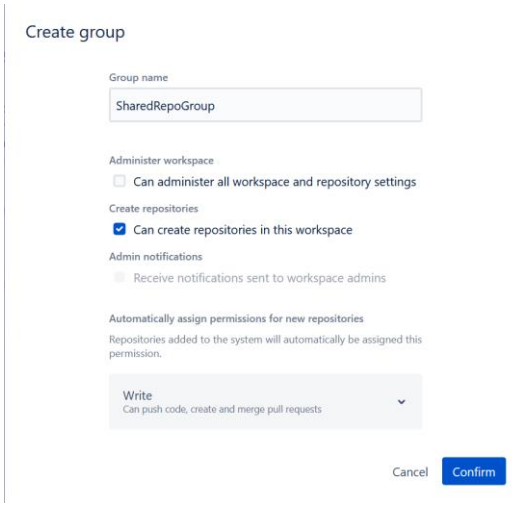
Click on Settings to transition to the workspace settings page, which should look something like this:



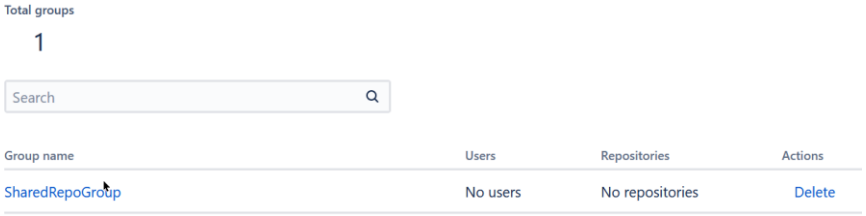
In the access management section on the left pane, click on User Groups.



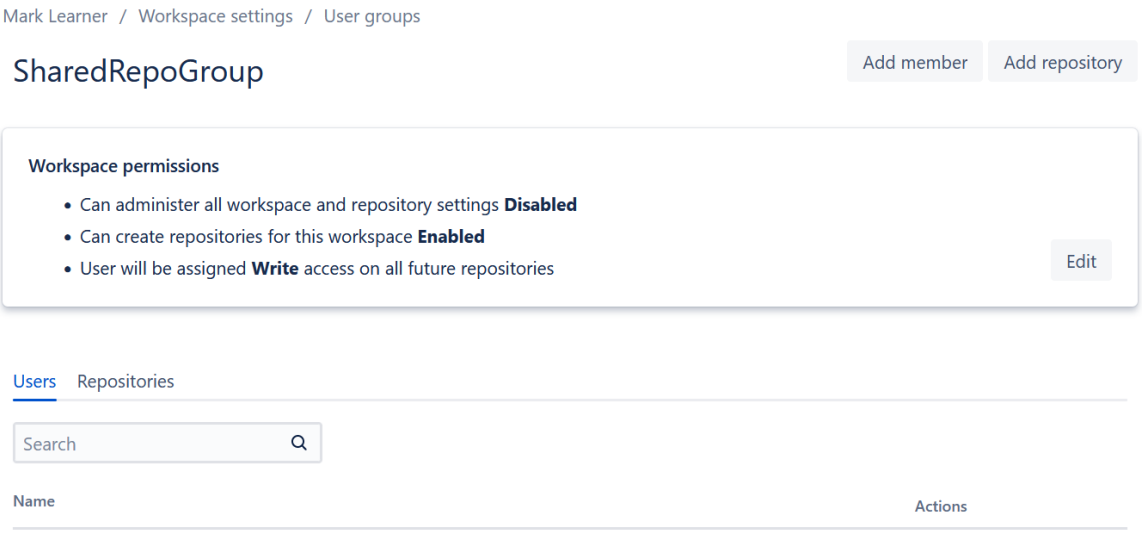
Click on Create Group. In the dialog box that appears, you have the initial 2 settings possible for workspace access permission and the final drop down list provides all the repository permissions.
Provide the group name as: `SharedRepoGroup` and provide the settings as demonstrated below, and then click Confirm. This group is provided standard read / write access to the remote repo, but not admin access: which should be the case for most dev team members.



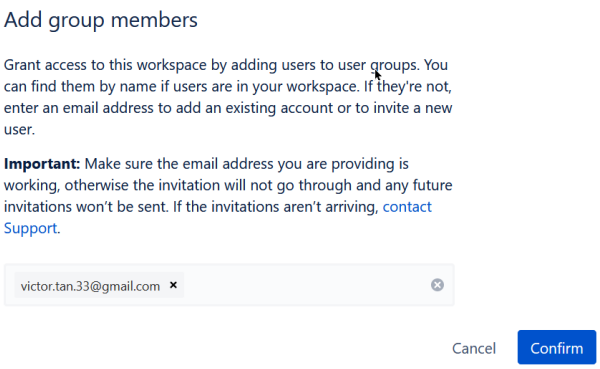
The new user group should now appear in the listing.



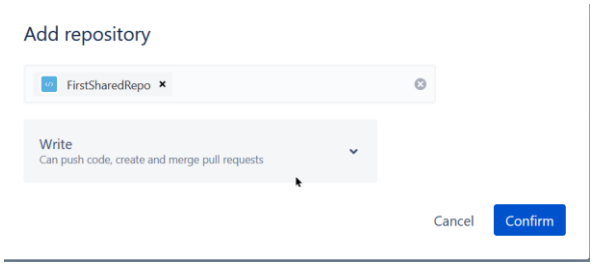
Click on the single group name in the listing.



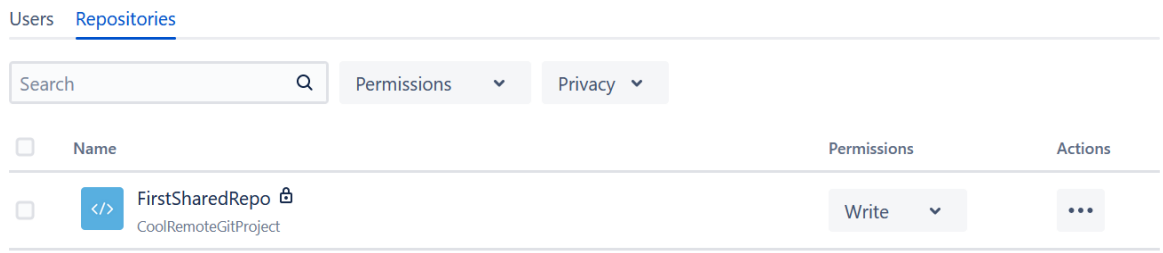
Click on Add Member and type in the primary email address of Dev B, then click confirm.



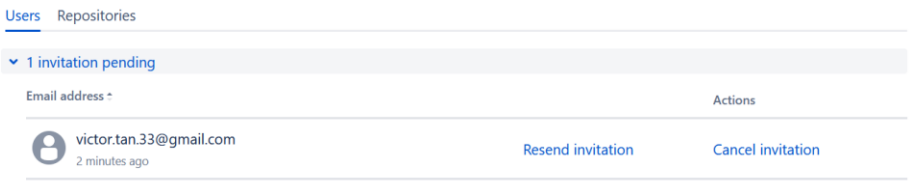
Click on Add Repository, and select the newly created repo (FirstSharedRepo), set the permission to Write and then click Confirm.



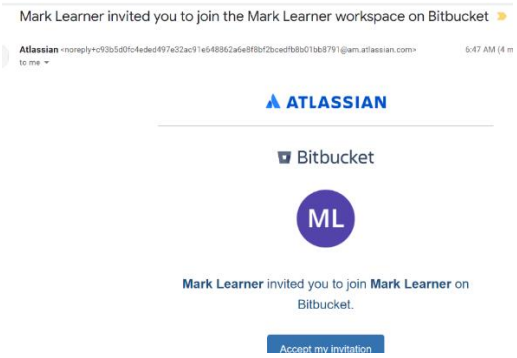
If you click on the Repositories link in the listing, you should be able to see it listed with the correct permissions.



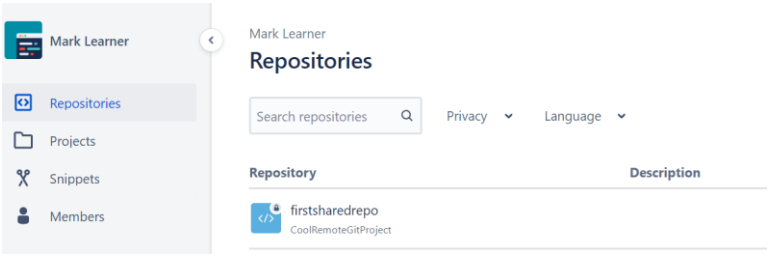
Switching back to the Users view, if you click on the 1 invitation pending link, you will see that an invitation email has been sent out to the email of Dev B.



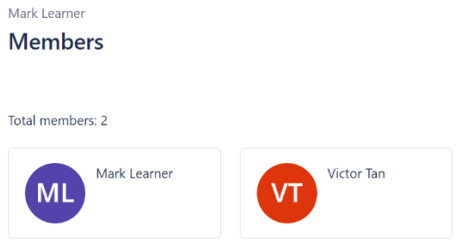
Using a different browser from the one used by Dev A, open Dev B's email account in order to confirm the receipt of an invitation similar to the one shown below. Clicking on Accept My invitation will enrol him automatically into the user group created earlier.



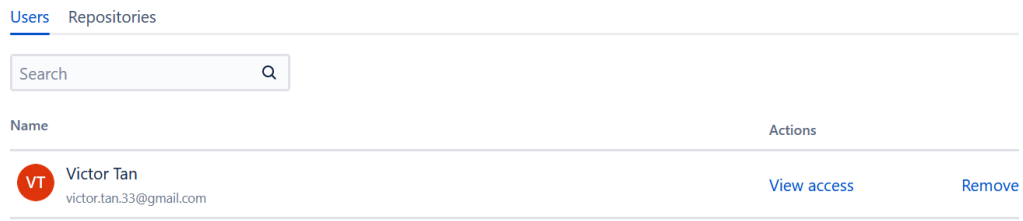
After clicking on the Accept invitation link, Dev B should now be transitioned to the main workspace page for this workspace.



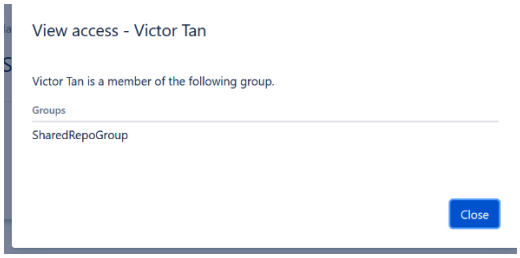
Clicking on the Members item in the left pane shows the current 2 active members in this workspace: Dev A and Dev B.



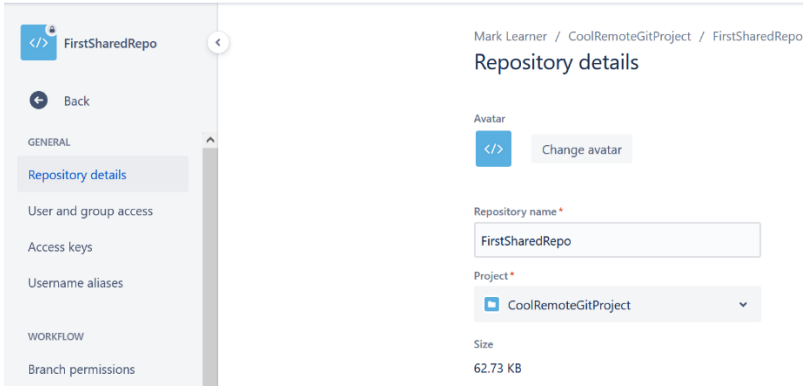
Back in Dev A's browser, click on User Groups and open the same group again. This time you should see that Dev B has now been enrolled in the users listing.



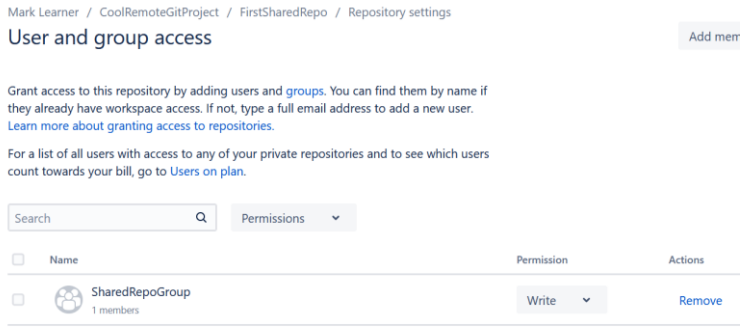
If you click on View Access, you can confirm that Dev B is part of the SharedRepoGroup and therefore has all access rights and permissions accorded to this group.



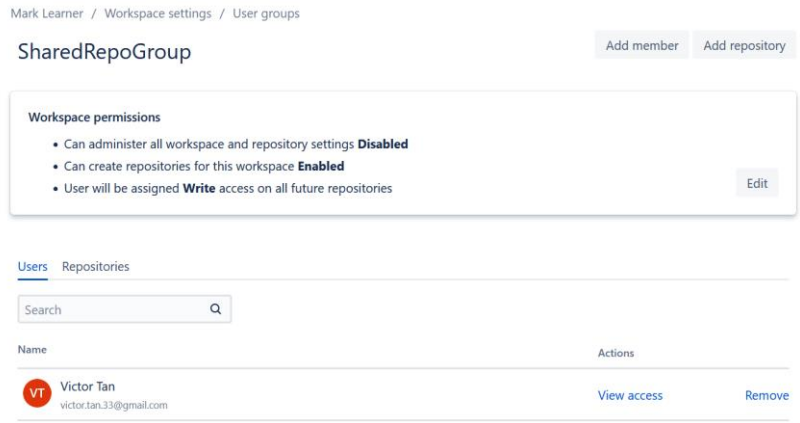
We can double confirm this by checking on the Repository Settings for FirstSharedRepo (which we had just added to this workspace). Click on Repositories option in the top menu listing, and select FirstSharedRepo. In the left hand pane, select Repository Settings.



In the Repository Settings, select User and Group access in the left pane.

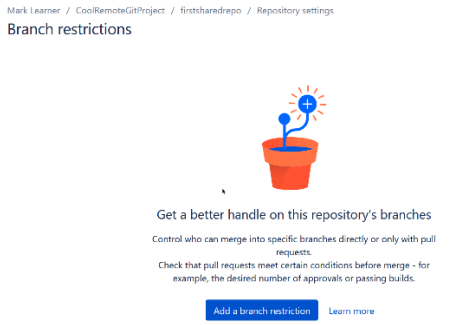


Click on SharedRepoGroup. You will be transitioned back to the same Workspace Settings page that you were at earlier.

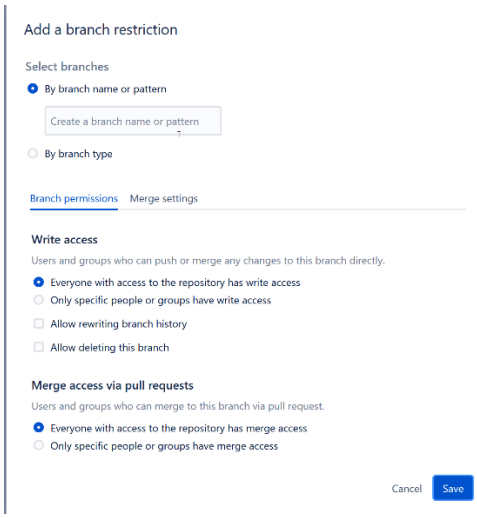


In addition to enforcing access control at the repository level, we can also choose to enforce more fine grained access control at the branch level, for e.g. to determine who has the right to push to a particular remote branch and overwrite it or merge into a remote branch (particularly master).

In the Repository Settings for FirstSharedRepo, you can select Branch restrictions in the Side Panel, and click on Add a branch restriction.



The Dialog box that appears shows you the various options available for enforcing access control on specific branches and the type of access control that can be enforced. We will not be going into that level of detail for this simple lab, but remember that you have this option open to you for a real-life project if necessary. So for now, click on Cancel.



10 Dev B obtains an app password to interact with remote repo

Just like Dev A, Dev B (and all other dev team members) need to obtain an app password in order for them to interact with the shared remote repo from the Git Bash shell, similar to the case in a previous lab.

We can follow the procedure outlined earlier for the case of obtaining an app password for Dev A, but now from the account of Dev B in a separate browser.

11 Cloning the shared repo to local repo for Dev B

Step e) in the sample collaborative development workflow

We are now going to clone the shared remote repo into a local repo in the development work area for Dev B (this would be the `devB` subfolder of the main `labs` folder)

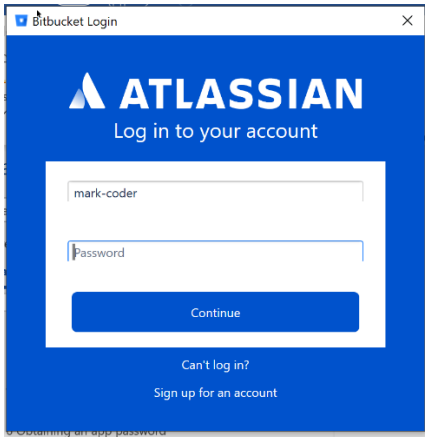
Open a new Git Bash shell in this `devB` subfolder. Keep the previous Git Bash shell in the `devA/firstsharedrepo` subfolder open (Note: you can have as many Git Bash shells open as you want in multiple directories to work with multiple Git repos simultaneously).

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

In Dev A's browser, navigate to FirstSharedRepo from the Repositories main page, and select the Clone button in the upper right hand corner of the Source view, and copy the command in the dialog box that appears.

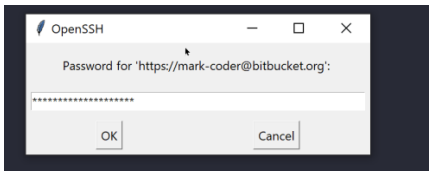
Paste this into the newly opened Git Bash shell in the `devB` subfolder.

As was the case when Dev A initially uploaded content into the empty remote repo a few lab sessions ago, the attempt to clone a populated remote repo into a local repo will also require authentication from the BitBucket server. Depending on the state of your BitBucket account, you may initially 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 for Dev B 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: `firstsharedrepo`

```
Cloning into 'firstsharedrepo'...
remote: Enumerating objects: 6, done.
remote: Counting objects: 100% (6/6), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 6 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (6/6), 505 bytes | 42.00 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.

The terms working directory, working copy, working tree are used interchangeably to refer to the same thing. When you initialize a Git repository in the root folder of your project, that folder hierarchy becomes associated with that repository and is now known as the working directory or working tree. When you clone a local repo from a remote repo, then the working tree is constructed from the Git objects inside the cloned repo. In this case, we could call it a working copy.

12 Getting info 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.

Close the `devB` Git Bash shell, and open a new one in `devB/firstsharedrepo` (or alternatively just navigate into it using `cd firstsharedrepo`).

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

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

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
```

You will be prompted again for your app password for authentication purposes. This will happen every time you interact with the remote repo - so we will assume that you will simply enter the correct password from this point onwards.

```
* remote origin
  Fetch         URL:         https://victor-coder@bitbucket.org/mark-
coder/firstsharedrepo.git
  Push            URL:       https://victor-coder@bitbucket.org/mark-
coder/firstsharedrepo.git
  HEAD branch: master
  Remote branch:
    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)
```

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)
```

The messages above tell us that the local master and remote master branch are fully synchronized, in that they are both pointing to identical commits in their respective repos. We can also verify this info by checking the local commit history with:

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

```
HEAD -> master, origin/master, origin/HEAD) Added animals with a
single line
```

Here we can also see that the local HEAD is pointing to the local master, while the remote HEAD is pointing to the remote master and all these are pointing to the same commit.

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

```
$ git branch --remotes

  origin/HEAD -> origin/master
```

```
  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/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`.

13 Dev B creates and pushes a feature branch to remote repo

Step f) - i) in the sample collaborative development workflow

Before we make any changes, let's set the local user.name and user.email properties to match the BitBucket account details of Dev B. In the Git Bash shell of `devB/firstsharedrepo`, type:

```
$ git config --local user.name "BitBucket account name Dev B"

$ git config --local user.email "BitBucket account email Dev B"
```

Verify that these properties have been set correctly with:

```
$ git config --list --show-origin
```

Note that because we are setting these two important properties at the local level, they only apply at the level of specific Git repos. So if you switch to the Git Bash shell at `devA/firstsharedrepo` and type:

```
$ git config --list --show-origin
```

You should see that the properties for DevA are still in effect for this repo and are not cancelled by the setting of these properties in the local repo for DevB.

Of course, for a real life scenario, each developer will have their own local repo in their own user account, so these variables can be set at the `--global` level.

Switch back to the Git Bash shell of `devB/firstsharedrepo`. Lets create a new branch here with:

```
$ git checkout master

$ git checkout -b feature/new-cars
```

Here we precede the name of the branch with `feature` to identify it as branch that implements a feature (as opposed to a hotfix or bugfix). This is not compulsory, but it is helpful as BitBucket uses this approach to classify all the different branches available in a remote repo.

We assume that Dev B has been assigned to code the implementation for this new feature branch.

Inside `devB/firstsharedrepo`, create a file named as below and populate it as shown using a text editor. Make sure you include a new line after the end of the single line

```
1: honda
```

cars.txt

Stage this new file with:

```
$ git add --all
```

Create the first commit with:

```
$ git commit -m "Added new cars file with a single line"
```

Make a further modification to this new file, ensuring that you include a new line after the end of the single line

```
2: mercedes
```

cars.txt

Create the second commit on this branch with:

```
$ git commit -am "Added 2nd line to the new cars file"
```

To view the existing commit history of all branches in the repo, type:

```
$ git log --oneline --all
```

At this point, if we type

```
$ git branch --all

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

We will see that no information regarding a remote tracking branch for `new-cars`. 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/master
```

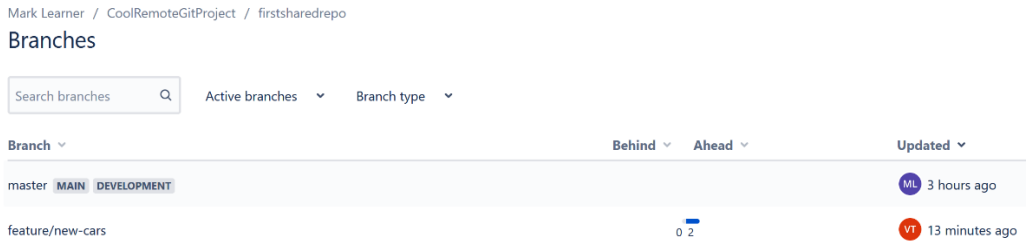
Let's try to directly push this new local branch to the remote repo with:

```
$ git push

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

    git push --set-upstream origin feature/new-cars
```

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:

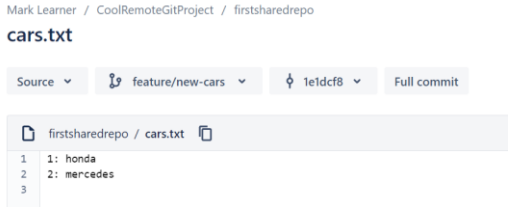
```
$ git push --set-upstream origin feature/new-cars

Enumerating objects: 7, done.
Counting objects: 100% (7/7), done.
Delta compression using up to 24 threads
Compressing objects: 100% (4/4), done.
Writing objects: 100% (6/6), 645 bytes | 645.00 KiB/s, done.
Total 6 (delta 0), reused 0 (delta 0), pack-reused 0
remote:
remote: Create pull request for feature/new-cars:
remote:       https://bitbucket.org/mark-coder/firstsharedrepo/pull-
requests/new?source=feature/new-cars&t=1
remote:
To https://bitbucket.org/mark-coder/firstsharedrepo.git
 * [new branch]      feature/new-cars -> feature/new-cars
branch 'feature/new-cars' set up to track 'origin/feature/new-cars'.
```

Head back to the browser view for Dev A and Dev B to verify that addition of a new upstream branch in the Branches view.



The 0/2 blue line indicates that this new branch is 2 commits ahead of the master branch.

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



Now if we check the list of remote tracking branches again in `devB/firstsharedrepo` with:

```
$ git branch --remotes

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

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

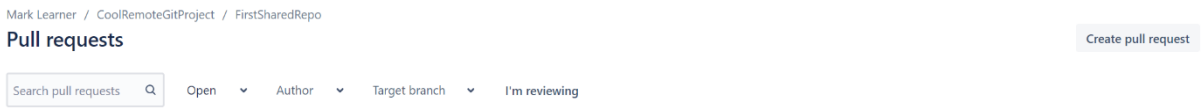
Note that in a real-life scenario, there would probably be several commits in this branch to get the code in it to a reasonably acceptable state before it is pushed to the remote shared repo.

14 Dev B initiates a pull request for this feature branch

Step j) in the sample collaborative development workflow

Pull requests allow a developer to notify members of their team that they have completed a feature (or are midway through completing a feature) for the main project. Once their feature branch is ready for review and published to the shared remote repo, the developer files a pull request via their Bitbucket account. This kickstarts a review process on a dedicated forum related to the that pull request involving the pertinent team members discussing the proposed feature.

In Dev B's browser main view for the FirstSharedRepo, click on Pull Requests, and in the Pull Requests main view, click on the Create Pull Request Button in the upper right hand corner.

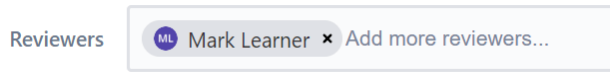


The first part of the Create Pull Request view indicates that the intention is for the newly created branch to be eventually merged into the master branch, which is the standard procedure for most common development workflows.

In the description box, type an appropriate message for this pull request to kickstart the review process. You can use the various formatting tools to style the message if you like.

```
Just added in some new cars. Please check it out to see whether its
appropriate for our app.
```

Then select the reviewers for the pull request. Here, we will only select Dev A, but of course for a real life project, you can select multiple reviewers which would probably be the team members with invested interest in the branch content.

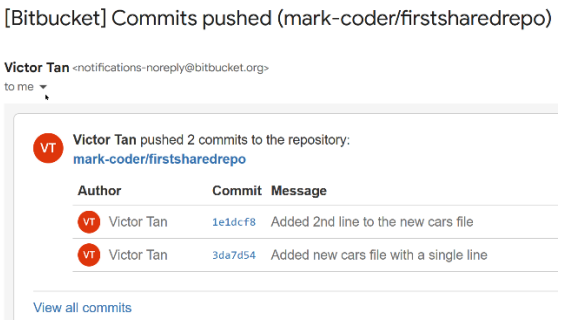


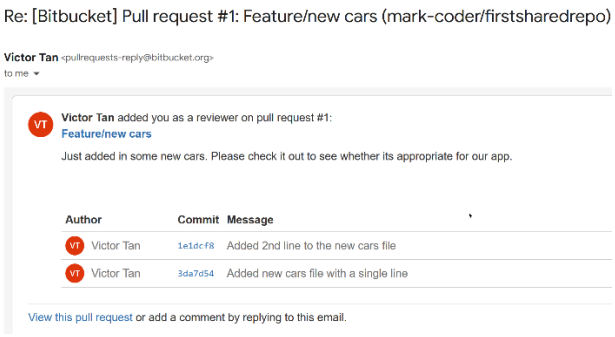
Finally click Create Pull Request.

15 Dev A downloads new feature branch to local repo

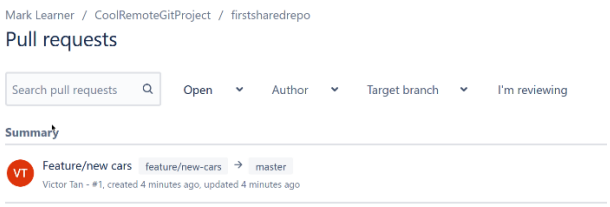
Step k) in the sample collaborative development workflow

At this point of time, Dev A will have received email notifications at their main email alias regarding pushes to shared remote repo as well as the initiation of the pull request (these notifications can be disabled if desired).

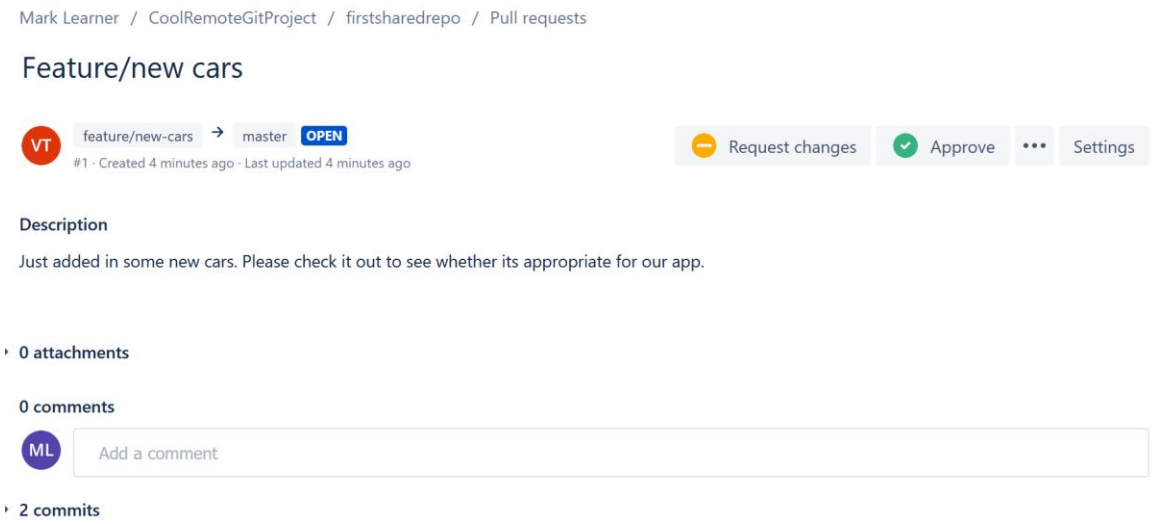




In either Dev A or Dev B's browser, the Pull Requests view should show the new pull request next to circle with the initials of the dev who created this request (in this case, Dev B).



You can click on it to follow the conversation thread on that request in more detail. At the moment, we only see the initial comment from Dev B.



Return to the Git Bash shell for `devA\firstsharedrepo`.

Next, we will download the latest state of the remote shared repo with:

```
$ git fetch
```

If prompted for the app password, remember to enter the correct app password for Dev A (and not Dev B) !!

```
remote: Enumerating objects: 7, done.
remote: Counting objects: 100% (7/7), done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 6 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (6/6), 625 bytes | 52.00 KiB/s, done.
From https://bitbucket.org/mark-coder/firstsharedrepo
 * [new branch]      feature/new-cars -> origin/feature/new-cars
```

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

```
$ git branch --all

* master
  remotes/origin/feature/new-cars
  remotes/origin/master
```

You will notice that there is now a remote tracking branch for the new upstream branch (`origin/feature/new-cars`), 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-cars

Switched to a new branch 'feature/new-cars'
```

```
branch 'feature/new-cars' set up to track 'origin/feature/new-cars'.
```

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

```
$ git branch --all

* feature/new-cars
  master
  remotes/origin/feature/new-cars
  remotes/origin/master
```

We can see that we now have a new local branch, which is also the active branch. Verify this with:

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

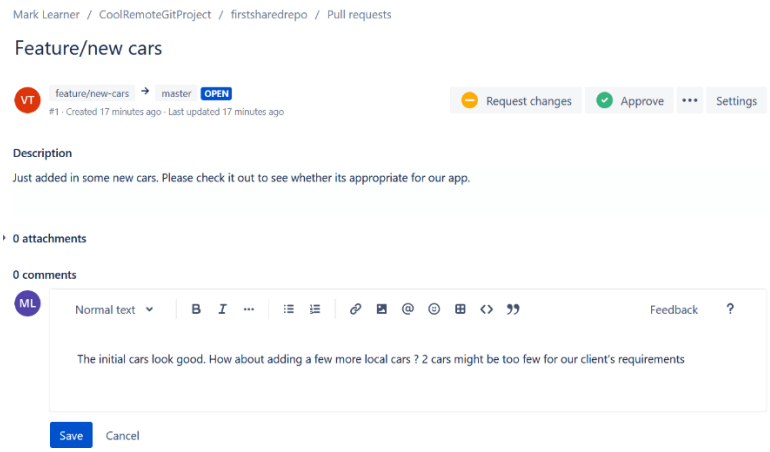
Dev A can now open `cars.txt` using a local IDE / text editor, inspect the content in it, and begin commenting on the pull request. Note that this step is entirely optional since Dev A can also inspect the content directly in the Source and Pull Requests view of the remote repo in the browser.

16 Dev A and Dev B exchange comments on pull request

Step k) in the sample collaborative development workflow

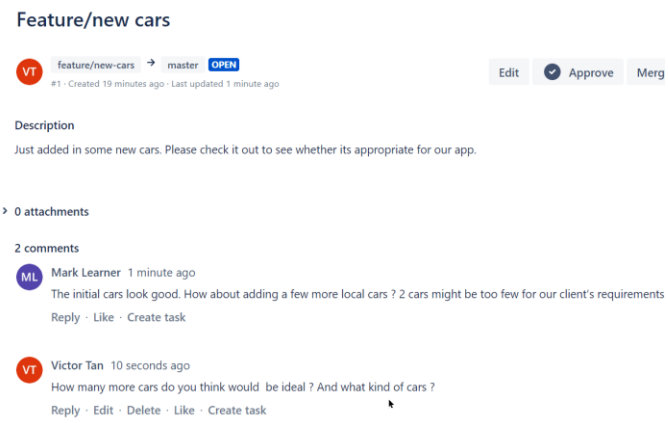
Navigate in Dev A's browser to the detailed view for the new Pull Request view. Add in a suitable comment to kickstart the discussion in the comment section. You can format it using the appropriate formatting buttons. Then click Save.

```
The initial cars look good. How about adding a few more local cars ?
2 cars might be too few for our client's requirements
```



Now, navigate in Dev B's browser to the Pull Request view, and click on the same Pull Request. You should be able to see the latest comment from Dev A, to which Dev B can respond with another comment and then Save as usual.

```
How many more cars do you think would  be ideal ? And what kind of
cars ?
```



Let's repeat the process above and simulate a conversation between Dev A and Dev B:

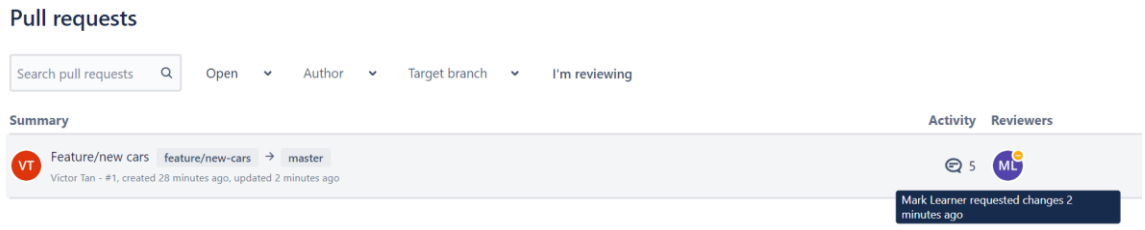
Dev A: `How about 4 cars ? Could we add in an EV as well ?`

Dev B: `I don't think Malaysia has an EV yet. Perodua has one in the pipeline, but won't be ready until another 10 years …..`

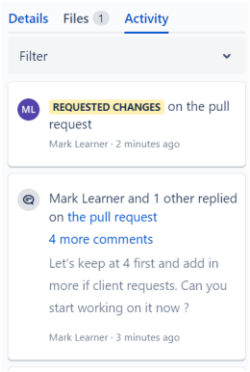
Dev A: `Let's keep at 4 first and add in more if client requests. Can you start working on it now ?`

After adding the last comment, Dev A clicks on Request Changes button in their view. This is an additional signal to Dev B to start making changes to this branch in accordance to the discussion at this point of time

Dev B should be able to see this request in their Pull Requests View.



This can also be seen in the Activity list in the right hand corner of the view for that specific Pull Request.



17 Dev B changes content in response to pull request comments

Step k) in the sample collaborative development workflow

Switch back to `devB\firstsharedrepo`. Remember to type the app password for Dev B for all the password prompts that appear.

Add the following new content to `cars.txt` and save, making sure to include an empty line at the bottom:

```
3: perodua
4: proton
```

Add in this new content into a commit with:

```
$ git commit -am "Added 2 new cars as per request by Dev A"
```

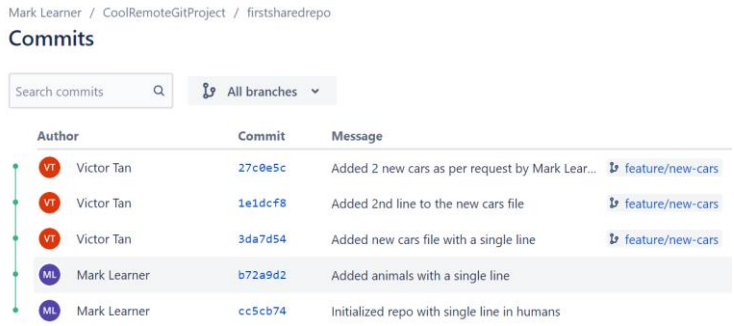
Check the status with:

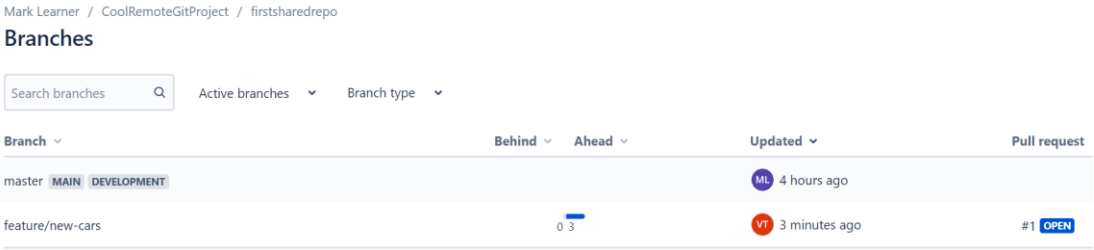
```
$ git status

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

nothing to commit, working tree clean
```

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

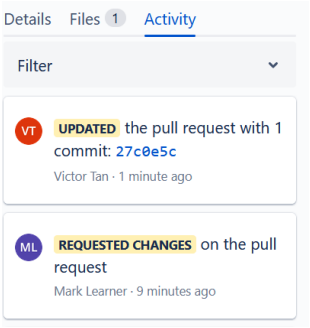
```
$ git push

Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
…
…
remote:
To https://bitbucket.org/mark-coder/firstsharedrepo.git
   1e1dcf8..27c0e5c  feature/new-cars -> feature/new-cars
```

Both Dev A and Dev B should be able to see this latest commit in the Commits and Branches main view.





In addition, we can see the latest commit as an update to the pull request in the Activity list in the right hand corner of the view for this specific Pull Request for both Dev and Dev B.

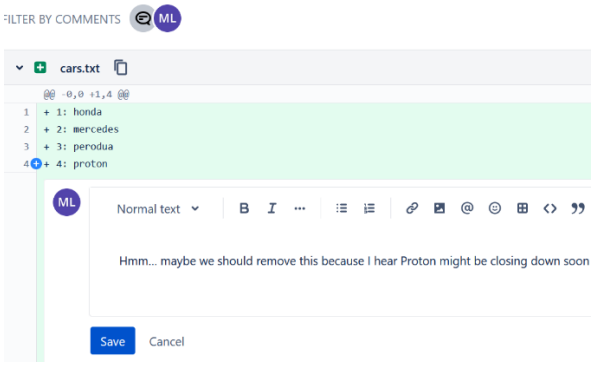


In this way, we can track all subsequent commits to the feature branch under discussion after the initial pull request was initiated.

In this example, Dev B is the one that makes changes to the feature branch as they were the one that created and initiated the pull request, but in practice, any member of the team in an appropriate role can make these changes. For e.g. Dev A can also download the latest updates to `feature/new-cars` to the tracking branch in their local repo, make changes there and push the latest commits back up to the upstream counterpart in the central remote repo (as we will see in an upcoming lab session).

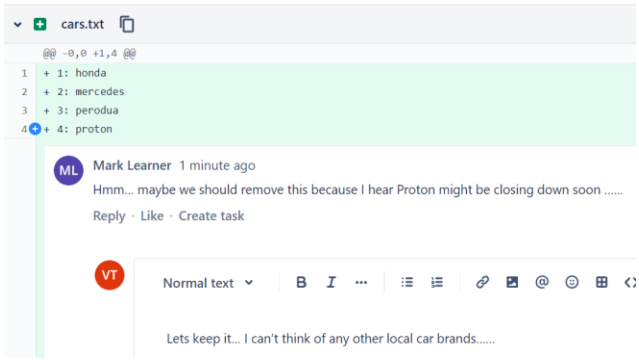
In addition to making comments on the main comments section, we can also add comments to individual lines of code in the diff section at the bottom of the view for this specific Pull Request. For e.g. Dev A can click on line `4 +proton` to add this comment and then Save.

`Hmm… maybe we should remove this because I hear Proton might be closing down soon ……`



And, of course, Dev B can also respond to this comment in the same section by clicking on reply and adding their own comments:

`Lets keep it… I can't think of any other local car brands……`



18 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.

In the Git Bash shell for `devA/firstsharedrepo` and `devB/firstsharedrepo`, type the appropriate command for your system:

For Linux:

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

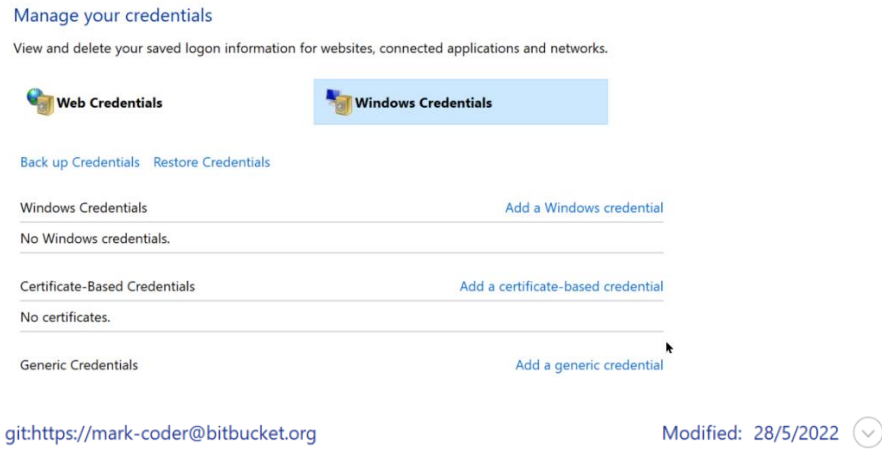
For Windows (using Git Bash)

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

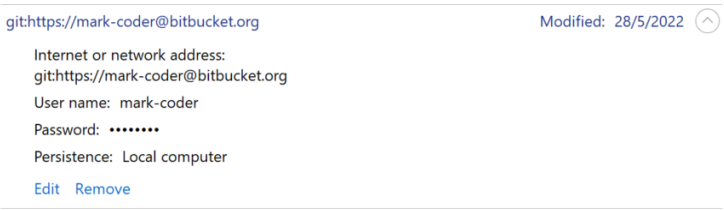
If there are issues with this password caching at any point later in the lab sessions, you can elect to undo the caching of the previous username/password pair with:

```
git config --local --unset credential.helper
git config --local 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.



Click to expand the entry and then click Remove.



19 Merging updates from upstream to local branches (Dev A)

Step k) in the sample collaborative development workflow

Return to the Git Bash shell for `devA\firstsharedrepo`.

Switch to the `feature/new-cars` branch (if you are not already there) with:

```
$ git checkout feature/new-cars

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

Notice that Git reports that the local `feature/new-cars` 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
```

If prompted for the app password, remember to enter the correct app password for Dev A (and not Dev B) !!

```
remote: Enumerating objects: 5, done.
….
   1e1dcf8..27c0e5c  feature/new-cars -> origin/feature/new-cars
```

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

Now if we check on the status again with:

```
$ git status

On branch feature/new-cars
Your branch is behind 'origin/feature/new-cars' 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. The `git pull` operation is equivalent to combining a `git fetch` (which we have already performed) followed by a `git merge` *`local-branch upstream-branch`* together. We will perform the merge ourselves explicitly now and demonstrate a `git pull` in a subsequent lab.

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/feature/new-cars

Note: switching to 'origin/feature/new-cars '.

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

Two key points to note here:

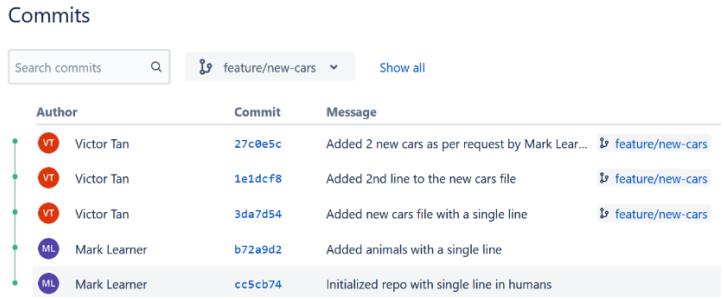
- Switching to the remote tracking branch moves the HEAD pointer to a detached state. This is because the remote tracking branch is not a real local branch, it is simply a pointer to an actual upstream branch in the remote repo. The prefix in the Git Bash prompt should now change to the commit hash of 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

27c0e5c (HEAD, origin/feature/new-cars) Added 2 new cars as per
request by Mark Learner
1e1dcf8 (feature/new-cars) Added 2nd line to the new cars file
3da7d54 Added new cars file with a single line
b72a9d2 (origin/master, master) Added animals with a single line
cc5cb74 Initialized repo with single line in humans
```

Verify that this is the same as the commit history shown in the Commits view for the `feature/new-cars` branch in the browser view:



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 feature/new-cars

$ git diff feature/new-cars origin/feature/new-cars

diff --git a/cars.txt b/cars.txt
index c18da4e..147984d 100644
--- a/cars.txt
+++ b/cars.txt
@@ -1,2 +1,4 @@
 1: honda
 2: mercedes
+3: perodua
+4: proton
```

Once we have inspected the commit history from the upstream branch as well as the latest changes using either of these two approaches, we can then make a decision on whether or not to merge in these 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/feature/new-cars

Updating 1e1dcf8..27c0e5c
Fast-forward
 cars.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. This simply downloads the latest commit(s) from the upstream branch into the local repo and then moves the local 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 feature/new-cars
Your branch is up to date with 'origin/feature/new-cars'.

nothing to commit, working tree clean
```

You can now open `cars.txt` in `devA\firstsharedrepo` to confirm its contents are now reflective of the latest changes.

20 Approving a pull request and merging into remote master

Step l) - m) in the sample collaborative development workflow

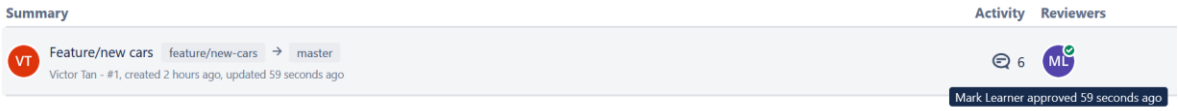
The discussion process above can be repeated with changes being made to the feature branch by Dev B and pushed to the shared repo, which can then be subsequently downloaded by Dev A (and all other team members for inspection).

At some point however in the discussion, a decision will be made to either approve or reject the pull request. In theory, the decision can be made by any team member; in practice, it is made by the project / technical lead. This usually occurs when the code in the branch involved in the pull request is adequately tested and considered ready to be merged into the production code base in the main / master branch. Remember that all commits in the main / master branch should contain code that is guaranteed to be working and ready to be deployed / released for client consumption.

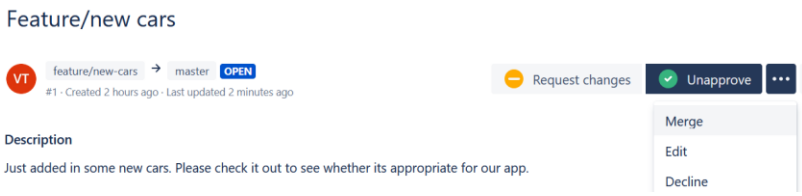
Let's assume here that Dev A chooses to approve the request by clicking on the Approve button in the view for this Pull Request.

The view will now change to reflect this

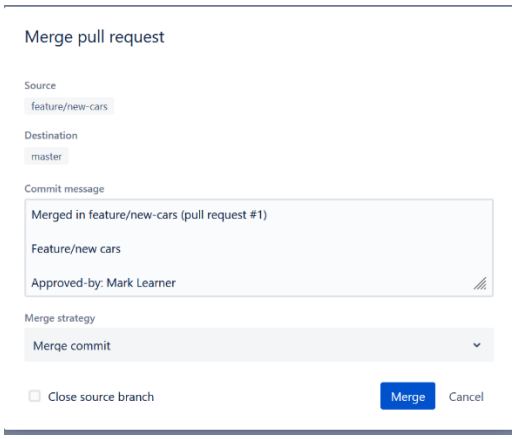
This new status will be reflected in the entry in the Pull Requests list view.



At this point, either the project / technical lead (Dev A) or the developer in charge of the branch (Dev B) can merge this branch into the remote main / master. Let's assume that Dev A does this by selecting the Merge option from the triple dot drop down menu.

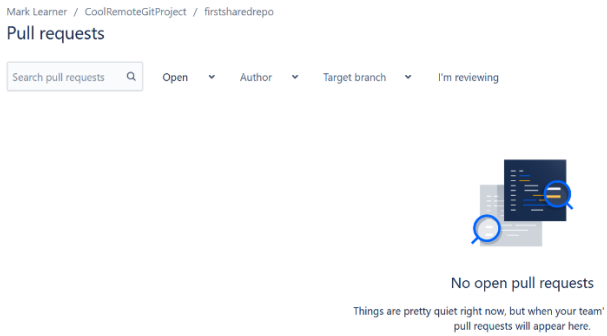


This brings up the Merge Pull request dialog box where you can provide the commit message and fix the merge strategy. You can also decide whether to close the feature branch once the merge is complete. For now, lets accept all the defaults and click Merge.

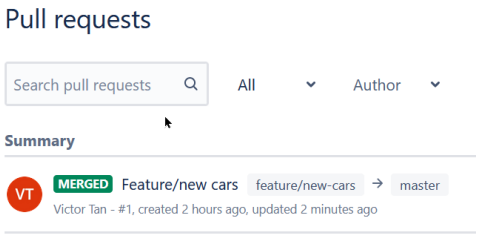


After a while, the merge should complete successfully in the background and the main Pull Request view should transition to indicate this:

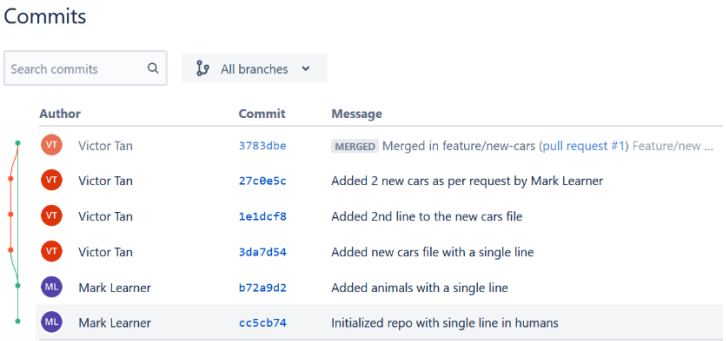
Now the main Pull Requests list is empty (because by default this only shows open requests).



Select the All or Merged option in the Drop down list to list all requests or merged requests. You will be able to see the newly merged pull request.



If we head back to the main Commits view for the master branch, we should be able to see the merged in commits from the branch, as well as a link to the related pull request.



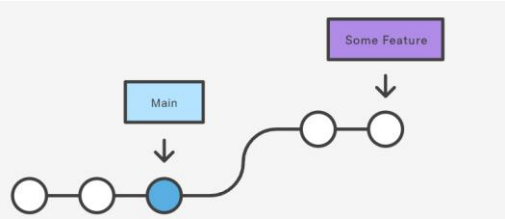
Notice that we have a new commit with the message: `Merged in feature/new-cars …..` in the `master` branch timeline.

We can check the contents of `cars.txt` from either the Source view or Commits view to verify that it now contains the latest changes.

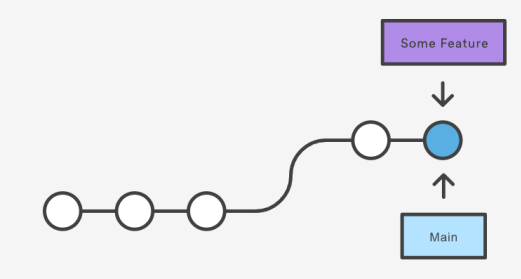
An important thing to note here is that the merging of the feature branch into the remote master succeeded without any need for manual intervention because this was a simple fast forward merge. Keep in mind the key features of the standard development workflow:

- All branches (feature / hotfix / bugfix / etc) will start from the remote main / master
- All development work (new commits) are performed in branches. We **NEVER** create new commits independently in the remote or local main / master
- When development work in a branch is complete, it is merged back into the remote main / master.

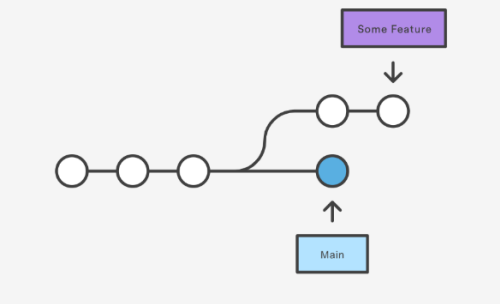
With this in mind, the main and feature branch will always look something like this (i.e. they are non-divergent branches)



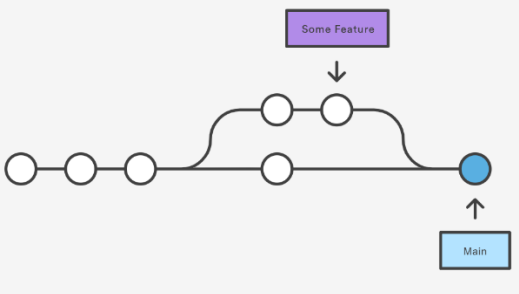
As a result, the merge of the feature branch into main will **ALWAYS** be a fast forward merge:



If on the other hand, we directly created independent commits in main / master at the same time as the ongoing development work in the feature branch (for e.g. a developer might work on her local master and push the changes to its remote counterpart), then we would have situation like this (i.e. divergent branches)



When we attempt to merge a divergent branches together, we will need to perform a 3 way merge, creating a new merge commit in the process.



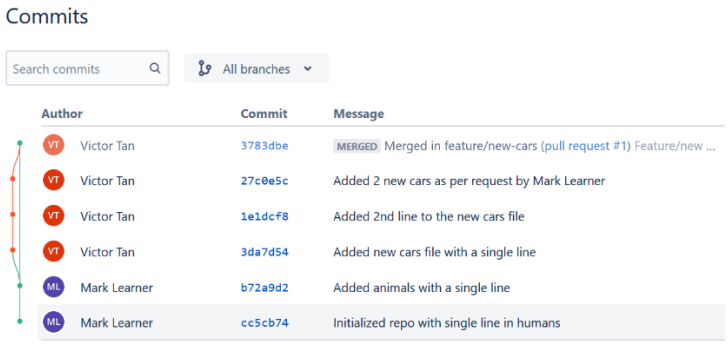
We have also seen in a previous lab that with a 3 way merge, there is a potential for a merge conflict occurring. Merge conflicts have to be resolved locally: they cannot be resolved within the remote repo. This means some developer will need to pull down the latest updates to both the remote main

and feature branch and then perform the merge in their local repo, resolve the conflict and push the updated local main to its remote counterpart. In addition to the complexities involve in resolving a merge conflict, allowing any developer to push new content from the local main to the remote main runs the risk of corrupting the remote main, which should only hold working production code that is ready to be released.

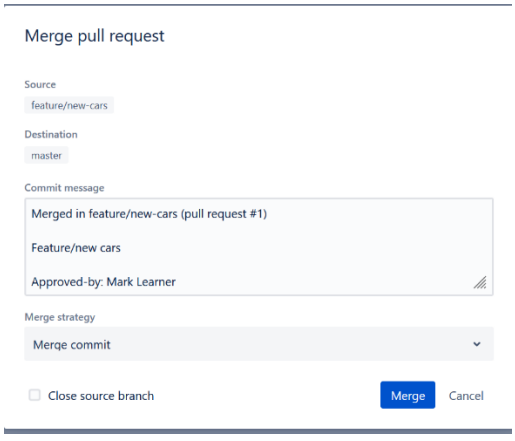
All of these difficulties is the reason why we try to avoid a merge conflict when merging a remote branch into the remote master, which we accomplish if we follow these 2 rules to the letter:

- All branches (feature / hotfix / bugfix / etc) will start from the remote master
- All development work (new commits) are performed in branches. We **NEVER** create new commits independently in the remote or local master

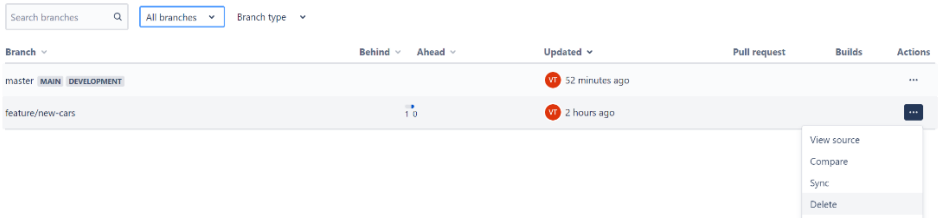
Another important thing to note here: if we look at the Commits view for the master branch, we will see a new merge commit as the latest commit.



However, it was mentioned earlier that the merge that occurred was a fast-forward merge, so by right, there should not be an extra merge commit (merge commits are only created for the case of 3 way merges). However, here we have an extra merge commit because we chose our merge strategy as merge commit in the earlier merge pull request dialog box.



Having this extra merge commit can be useful to explicitly indicate to us that a merge occurred at that particular point in the master timeline if we ever choose to delete that feature branch.
For e.g. we could choose to delete a feature branch from the Branches view via the drop down menu.



21 Merging updates from remote master into local repos

Step n) in the sample collaborative development workflow

Once the merge has been completed, it is important that all team members pull the latest version of the remote master to update the master branches in their local repo.

Return to the Git Bash shell for `devA/firstsharedrepo`.

Switch to the `master` branch (if you are not already there) with:

```
$ 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 we just completed a merge operation on it. Again, this is because we will not have the latest status of the remote repo until we execute:

```
$ git fetch
```

At this point, you should no longer be prompted for the app password if you have completed the correct caching of app password from an earlier lab session.

```
remote: Enumerating objects: 1, done.
remote: Counting objects: 100% (1/1), done.
remote: Total 1 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (1/1), 266 bytes | 133.00 KiB/s, done.
From https://bitbucket.org/mark-coder/firstsharedrepo
   b72a9d2..3783dbe  master     -> origin/master
```

If we now check the status:

```
$ git status

On branch master
Your branch is behind 'origin/master' by 4 commits, and can be fast-
forwarded.
  (use "git pull" to update your local branch)

nothing to commit, working tree clean
```

At this point, we could choose to perform a merge explicitly as we did in a previous lab. Instead, we will perform a pull instead:

```
$ git pull

Updating b72a9d2..3783dbe
Fast-forward
 cars.txt | 4 ++++
 1 file changed, 4 insertions(+)
 create mode 100644 cars.txt
```

A `git pull` basically combines a `git fetch` (which we had just executed earlier) followed by a `git merge`, where we merge the latest updates from the remote branch into the local branch. As expected, this results in a fast forward merge.

We can check the commit history of `master` with:

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

You should see that the commit history here now matches that shown in the Commits view in the browser.

We will repeat the same procedure for Dev B (and for all other dev team members, who can perform this update simultaneously without any issue):

Return to the Git Bash shell for `devB/firstsharedrepo`.

Switch to the `master` branch (if you are not already there) with:

```
$ git checkout master
```

In this situation, we can immediately execute the git pull since this already combines a `git fetch` and `git merge` together:

```
$ git pull

remote: Enumerating objects: 1, done.
remote: Counting objects: 100% (1/1), done.
……
…..
1 file changed, 4 insertions(+)
 create mode 100644 cars.txt
```

Notice that the output messages indicate that both the `git fetch` and `git merge` was performed.

As before, we can check the commit history of `master` with:

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

You should see that the commit history here now matches that shown in the Commits view in the browser.

To summarize, `git pull` is equivalent to performing `git fetch` followed by `git merge`.

a) We use `git pull` when we are very confident about the status of the local branch and the latest updates in the remote branch that will be updated into the local branch
b) We use two separate commands (`git fetch` and `git merge`) when we are unsure about the latest updates in the remote branch that will be merged into the local branch. The first (`git fetch`) allows us to get info about the latest updates via the remote tracking branch, and once we are satisfied, we can merge the remote tracking branch into the local branch.

The comments above apply for all branches, regardless of whether we are working with the main / master branch or any feature / hotfix / bugfix branches.

22 Merging latest updates from local master into ongoing local branch

Step o) in the sample collaborative development workflow

One other important thing to keep in mind is that that most (if not all) dev team members may have an ongoing feature branch under development when the remote master is updated through a merge from an approved pull request. Therefore, after merging these changes into their local master, they need to go one further step and merge these updates from their local master into their local feature branch. This is important to ensure that their current development work on their feature branch incorporates the latest developments in the main codebase.

Once their local feature branches have being updated, they can then push these changes to the corresponding upstream branch in the remote repo (assuming that they are already published).

Let's demonstrate this with an example that combines all the steps from the previous lab sessions.

Return to the Git Bash shell of `devB/firstsharedrepo`. Let's create another new branch here with:

```
$ git checkout master

$ git checkout -b feature/more-humans
```

Inside `devB/firstsharedrepo`, add the following lines to this existing file. Make sure you include a new line after the end of the single line

```
2: CEO
```

humans.txt

Add this into the first commit on this new branch with:

```
$ git commit -am "Added a rich person"
```

Add another extra line to this existing file. Make sure you include a new line after the end of the single line

```
3: project manager
```

humans.txt

Add this into the second commit on this new branch with:

```
$ git commit -am "Added a poor person"
```

To view the commit history of this new branch, type:

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

At this point, if we type

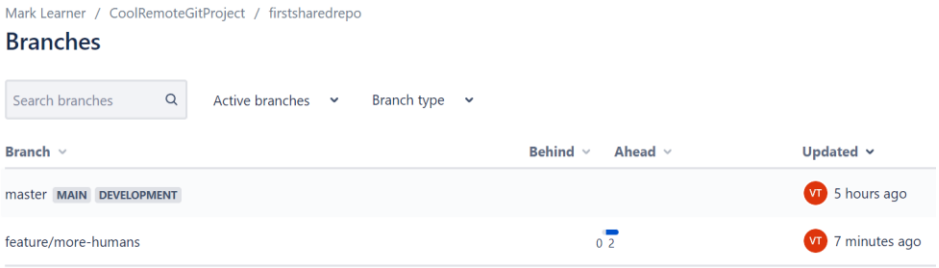
```
$ git branch --all

* feature/more-humans
  feature/new-cars
  master
  remotes/origin/HEAD -> origin/master
  remotes/origin/feature/new-cars
  remotes/origin/master
```

We will see that no information regarding a remote tracking branch for `more-humans`. This is because a remote tracking branch will not be created until we push this new branch to the remote repo, as we have seen previously.

To push this new local branch to a remote repo and establish a remote tracking branch for it, we need to type:

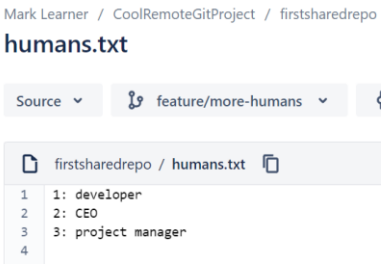
```
$ git push --set-upstream origin feature/more-humans
```

Head back to the browser view for Dev A and Dev B to verify that addition of a new upstream branch in the Branches view.



The 0/2 blue line indicates that this new branch is 2 commits ahead of the master branch.

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



Now if we check the list of remote tracking branches again in `devB/firstsharedrepo` with:

```
$ git branch --remotes

  origin/HEAD -> origin/master
  origin/feature/more-humans
  origin/feature/new-cars
  origin/master
```

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

Let's repeat this creation of a new feature branch for Dev A. This reflects the real life situation of dev team members working on their feature branches together in parallel.

Return to the Git Bash shell of `devA/firstsharedrepo`. Let's create another new branch here with:

```
$ git checkout master

$ git checkout -b feature/more-animals
```

Inside `devA/firstsharedrepo`, add the following lines to this existing file. Make sure you include a new line after the end of the single line

```
2: dog
```

animals.txt

Add this into the first commit on this new branch with:

```
$ git commit -am "This is a nice pet"
```

Add another extra line to this existing file. Make sure you include a new line after the end of the single line

```
3: snake
```

animals.txt

Add this into the second commit on this new branch with:

```
$ git commit -am "That is a dangerous one !"
```

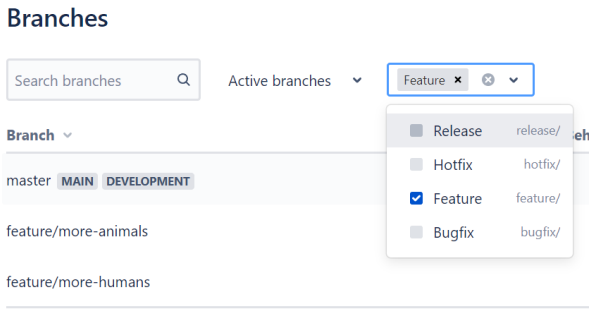
To view the commit history of this new branch, type:

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

To push this new local branch to the remote repo and establish a remote tracking branch for it, we need to type:

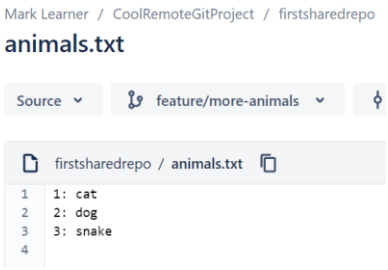
```
$ git push --set-upstream origin feature/more-animals
```

Head back to the browser view for Dev A and Dev B to verify that addition of a new upstream branch in the Branches view. Note that we can use the Branch type drop down list to filter the list of branches to display based on the first part of the branch name (the reason why we precede all our branch names with `feature/`).



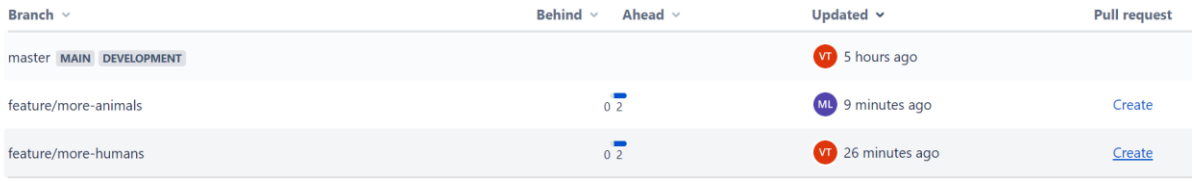
The 0/2 blue line indicates that this new branch is 2 commits ahead of the master branch.

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



Now that we have remote versions of the local branches that both Dev A and Dev B are working on, let's get Dev B to initiate Pull Request for their new remote feature branch.

In Dev B's browser main view for Branches, select Create in the Pull Request column in the row for `feature/more-humans`.

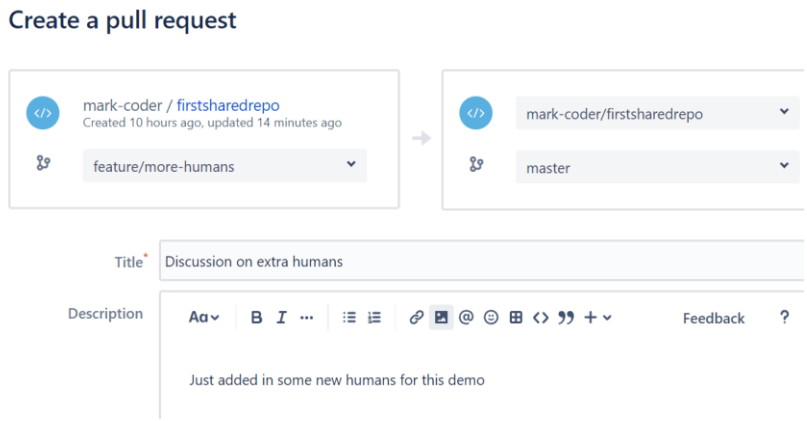


In the Create Pull Request view, enter the following for the title:

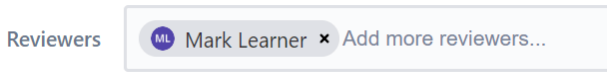
```
Discussion on extra humans
```

In the description box, type an appropriate message for this pull request to kickstart the review process. You can use the various formatting tools to style the message if you like.

```
Just added in some new humans for this demo
```



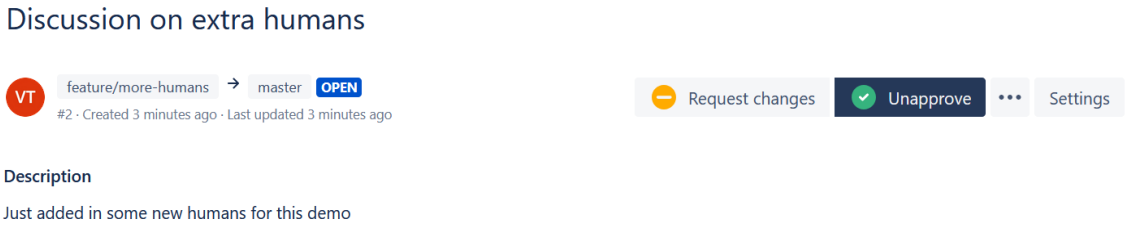
Then select Dev A as the reviewer for this request.



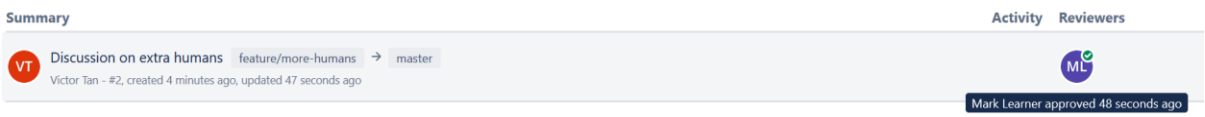
Finally click Create Pull Request.

As seen before previously, all dev team members can proceed to engage in discussion on this pull request after which any one of the team members can approve the merging of the contents of this new feature branch into master.

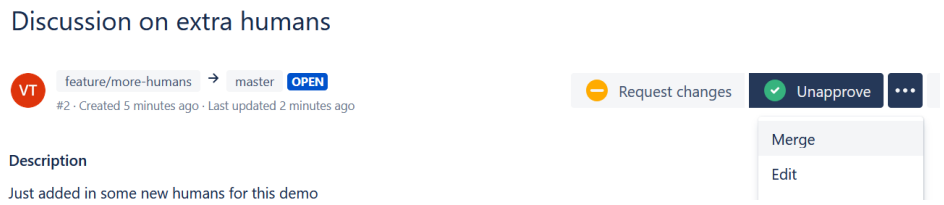
We will shortcut this process for this example and assume here that Dev A chooses to approve the request by clicking on the Approve button in the view for this Pull Request. The view will now change to reflect this



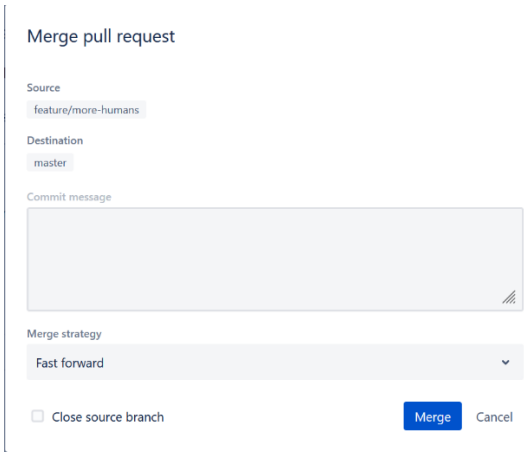
This new status will be reflected in the entry in the Pull Requests list view.



Next, Dev A will merge this branch into the remote main / master by selecting the Merge option from the triple dot drop down menu.

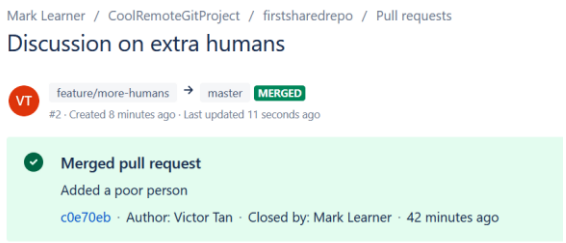


This brings up the Merge Pull request dialog box where you can provide the commit message and fix the merge strategy. You can also decide whether to close the feature branch once the merge is complete.
This time we will select a Fast Forward merge strategy. In this case, notice that the commit message box is greyed out because a new merge commit will not be created and there is no need to obtain a commit message from the user for this case.

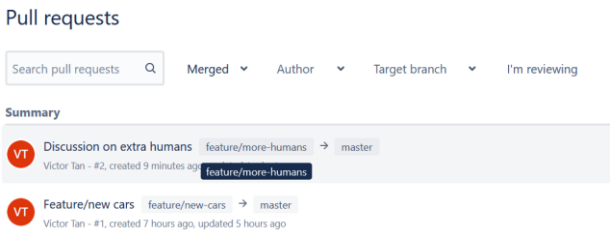


Go ahead and click Merge.

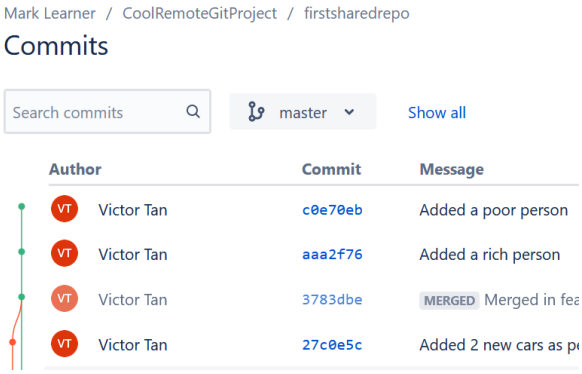
After a while, the merge should complete successfully in the background and the main Pull Request view should transition to indicate this:



Now the main Pull Requests list is empty (because by default this only shows open requests). Select the All or Merged option in the Drop down list to list all requests or merged requests. You will be able to see the newly merged pull request.



If we head back to the main Commits view for the master branch, we should be able to see the merged in commits from the branch.



Notice that now, unlike the previous merge that we did in the previous lab session, we do not have an extra merge commit on the `master` branch timeline. This is because we now explicitly specified a Fast Forward merge strategy in the previous Pull Request dialog box when performing the merge.

The advantage of not having an extra merge commit is that it does not clutter up the timeline of the `master` branch, particularly when we are merging in many feature / bug-fix branches into this branch over its lifetime. The disadvantage is that we can no longer clearly see from the view which feature branch contributed to the particular sequence of commits on the `master` branch.

Once the merge has been completed, it is important that all team members pull the latest version of the remote master to update the master branches in their local repo.

Return to the Git Bash shell for `devA/firstsharedrepo`.

Switch to the `master` branch (if you are not already there) with:

```
$ 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 we just completed a merge operation on it. Again, this is because we will not have the latest status of the remote repo until we execute:

```
$ git fetch
```

Next, we check the status:

```
$ git status

On branch master
Your branch is behind 'origin/master' by 2 commits, and can be fast-
forwarded.
  (use "git pull" to update your local branch)

nothing to commit, working tree clean
```

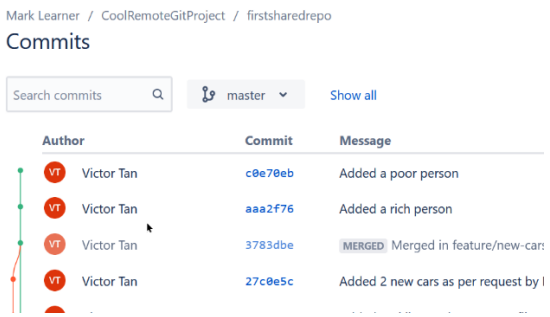
Next, we perform a pull:

```
$ git pull
```

We can check the commit history of `master` with:

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

You should see that the commit history here now matches that shown in the Commits view in the browser for the `master` branch



Now the final step is to merge in the latest updates from the local `master` into the local `feature/more-animals`. This is important to ensure that this feature branch that Dev A is currently working on incorporates the latest changes as a result of updates from other devs in the team.

```
$ git checkout feature/more-animals

$ git merge master
```

As we are performing this as a 3-way merge locally, the default editor will open up prompting for a new commit message. We can accept the default message and close the editor.

Next, we can check the commit history structure of all branches with:

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

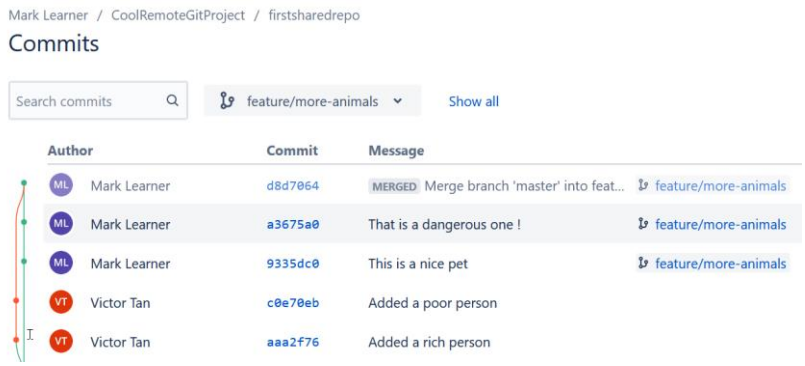
We should be able to see the new merge commit with both HEAD and `feature/more-animals` pointing to this commit, which is exactly the expected behavior for the case of a 3-way merge.

The important point to note here is that 3-way merge does not result in a conflict as both Dev A and Dev B were modifying two different files in the code base.

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

```
$ git push
```

You should see that the Commits view in the browser for the `feature/more-animals` branch commit history now matches that of the local branch.



Before we merge `feature/more-animals` branch into the remote `master`, let's add one more commit to it.
Add some extra lines to this existing file. Make sure you include a new line after the end of the single line

```
4: zebra
5: lion
```

animals.txt

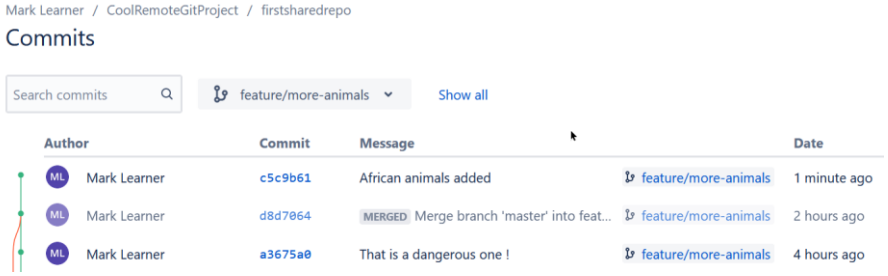
Add this into a new commit on this branch with:

```
$ git commit -am "African animals added"
```

Next, we push these latest changes up to the remote repo with:

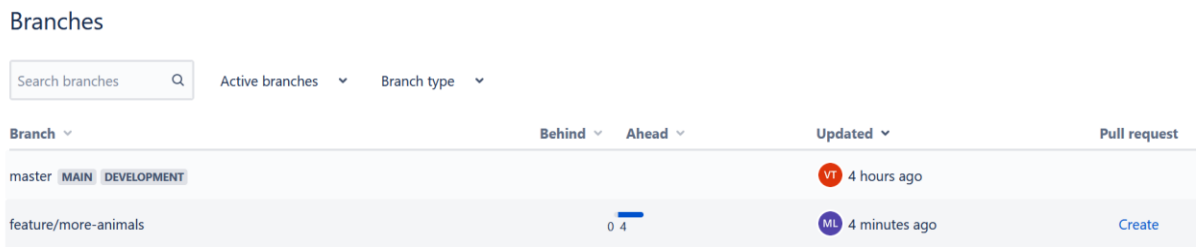
```
$ git push
```

You should see that the Commits view in the browser for the `feature/more-animals` branch commit history now includes the latest commit.



Finally, let's get Dev A to initiate Pull Request for their remote feature branch.

In Dev A's browser main view for Branches, select Create in the Pull Request column in the row for `feature/more-animals`.

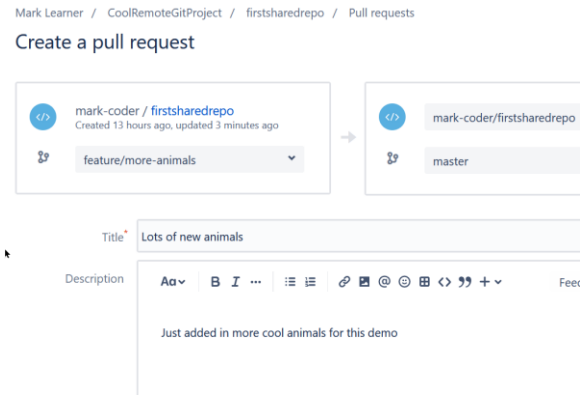


In the Create Pull Request view, enter some random text for the title:

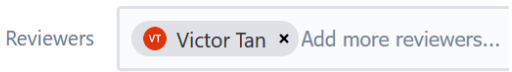
```
Lots of new animals
```

In the description box, type an appropriate message for this pull request to kickstart the review process. You can use the various formatting tools to style the message if you like.

```
Just added in more cool animals for this demo
```



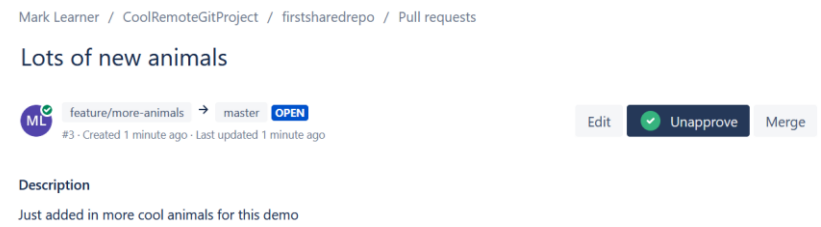
Then select Dev B as the reviewer for this request.



Finally click Create Pull Request.

As seen before previously, all dev team members can proceed to engage in discussion on this pull request after which any one of the team members can approve the merging of the contents of this new feature branch into master.

We will shortcut this process again and assume here that Dev A chooses to approve the request by clicking on the Approve button in the view for this Pull Request. The view will now change to reflect this

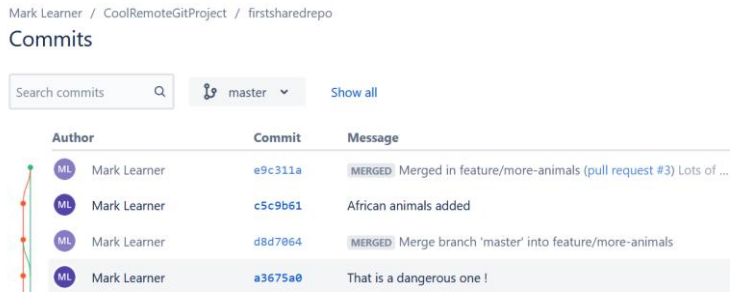


Next, Dev A will merge this branch into the remote main / master by clicking the Merge option next to the triple dot drop down menu.

This brings up the Merge Pull request dialog box where you can provide the commit message and fix the merge strategy. In this case, we will accept the defaults (with a merge strategy of merge commit) and go ahead and click merge.

After a while, the merge should complete successfully in the background and the main Pull Request view should transition to indicate this:

If we head back to the main Commits view for the `master` branch, we should be able to see the merged in commits from this feature branch.



This time, we have an extra merge commit on the `master` branch timeline.

Once the merge has been completed, we again get all team members to pull the latest version of the remote master to update the master branches in their local repo.

Return to the Git Bash shell for `devA/firstsharedrepo`.

Switch to the `master` branch (if you are not already there) with:

```
$ git checkout master
```

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

We can straight away perform the pull operation (skipping the optional `git fetch` and `git status`):

```
$ git pull
```

We can check the commit history of `master` with:

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

You should see that the commit history here now matches that shown in the Commits view in the browser for the `master` branch

We will repeat the same operation for Dev B

Return to the Git Bash shell for `devB/firstsharedrepo`.

Switch to the `master` branch (if you are not already there) with:

```
$ git checkout master
```

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

We can straight away perform the pull operation (skipping the optional `git fetch` and `git status`):

```
$ git pull
```

We can check the commit history of `master` with:

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

You should see that the commit history here now matches that shown in the Commits view in the browser for the `master` branch

If we now do a:

```
$ git branch --all
```

```
  feature/more-humans
  feature/new-cars
* master
  remotes/origin/HEAD -> origin/master
  remotes/origin/feature/more-animals
  remotes/origin/feature/more-humans
  remotes/origin/feature/new-cars
  remotes/origin/master
```

we can see that we still have not yet downloaded the remote `feature/more-animals` into a local branch, even though we already have a tracking branch for it. However, since this particular

branch was already merged into the remote master and we have already merged the changes from the remote master to the local master, we don't really need to do this.

23 Resolving merge conflicts from different feature branches

Step o) in the sample collaborative development workflow

In the previous example, we saw that when we merged from the updated local master into an ongoing feature branch on the local repo for Dev A, the resulting 3-way merge did not result in a conflict. This was because both Dev A and Dev B were modifying two different files in the code base in their respective ongoing branches. If we adhere to this principle in the collaborative dev workflow (i.e. that each dev works on an individual source code file that is also not worked on by any other dev in their branch), then we can eliminate any possible merge conflicts.

However, the situation may arise whereby two (or more) developers will need to make changes to the same part of a given source code file in their respective feature branches. In that case, merging from the local master into an ongoing feature branch will result in a merge conflict which will then need to be resolved accordingly.

Let's simulate that situation using Dev A and Dev B again as examples.

Return to the Git Bash shell of `devB/firstsharedrepo`. Let's create another new branch here with:

```
$ git checkout master

$ git checkout -b feature/cool-cars
```

Inside `devB/firstsharedrepo`, add the following lines to this existing file. Make sure you include a new line after the end of the single line

```
5: Aston Martin
6: Porsche
```

cars.txt

Add this into the first commit on this new branch with:

```
$ git commit -am "Cool cars added here"
```

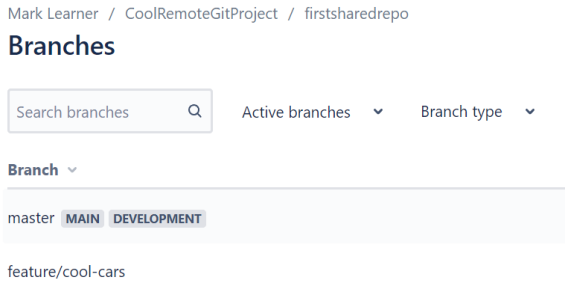
To view the commit history of this new branch, type:

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

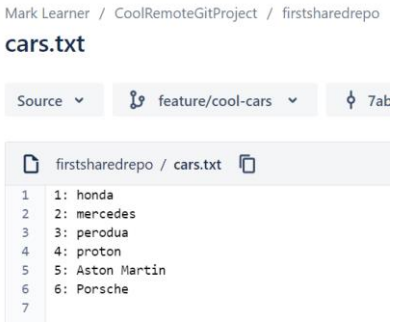
To push this new local branch to a remote repo and establish a remote tracking branch for it, we to type:

```
$ git push --set-upstream origin feature/cool-cars
```

Head back to the browser view for Dev A and Dev B to verify that addition of a new upstream branch in the Branches view.



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



We will repeat this example now with Dev A making changes to the same file in the same location on a feature branch in their local repo.

Return to the Git Bash shell of `devA/firstsharedrepo`. Let's create another new branch here with:

```
$ git checkout master
```

```
$ git checkout -b feature/boring-cars
```

Inside `devA/firstsharedrepo`, add the following lines to this existing file. Make sure you include a new line after the end of the single line

```
5: mitsubishi
6: hyundai
```

cars.txt

Add this into the first commit on this new branch with:

```
$ git commit -am "Boring cars added here"
```

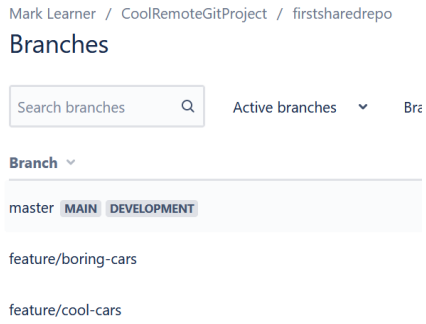
To view the commit history of this new branch, type:

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

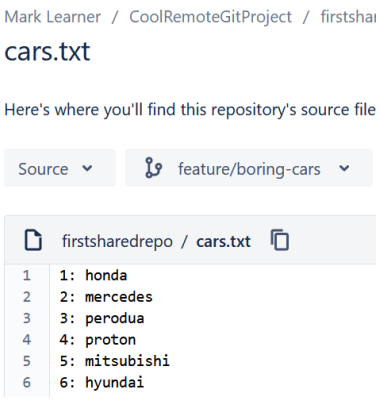
To push this new local branch to a remote repo and establish a remote tracking branch for it, we to type:

```
$ git push --set-upstream origin feature/boring-cars
```

Head back to the browser view for Dev A and Dev B to verify that addition of a new upstream branch in the Branches view.

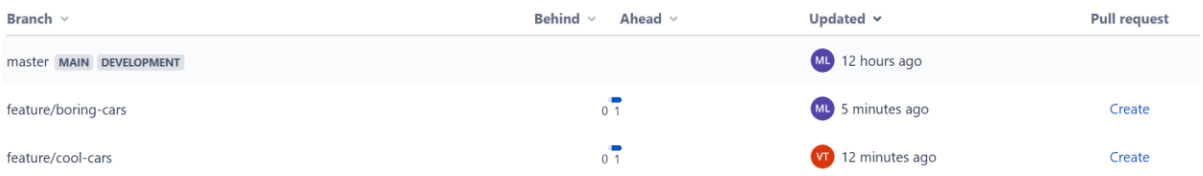


Switch to the Source view, switch to the new `feature/boring-cars` branch and click on `cars.txt` to verify that it contains the new content that we added locally.



Now that we have remote versions of the local branches that both Dev A and Dev B are working on, let's get Dev B to initiate Pull Request for their new remote feature branch.

In Dev B's browser main view for Branches, select Create in the Pull Request column in the row for `feature/cool-cars`.



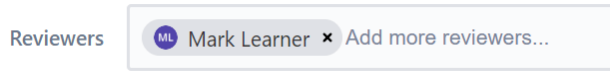
In the Create Pull Request view, enter the following for the title:

```
We all need cool cars
```

In the description box, type an appropriate message for this pull request to kickstart the review process. You can use the various formatting tools to style the message if you like.

```
Cool cars make life more interesting !
```

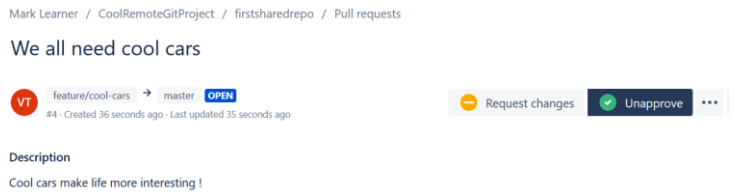
Then select Dev A as the reviewer for this request.



Finally click Create Pull Request.

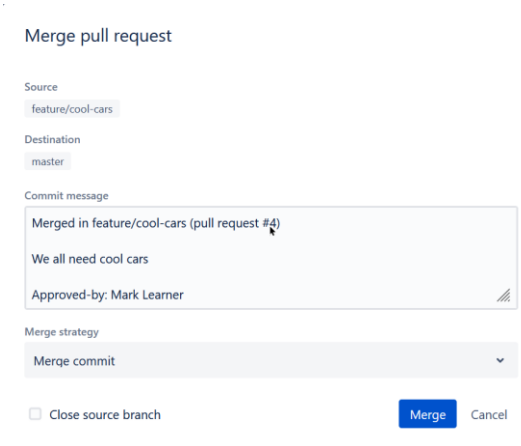
As seen before previously, all dev team members can proceed to engage in discussion on this pull request after which any one of the team members can approve the merging of the contents of this new feature branch into master.

We will shortcut this process for this example and assume here that Dev A chooses to approve the request by clicking on the Approve button in the view for this Pull Request. The view will now change to reflect this



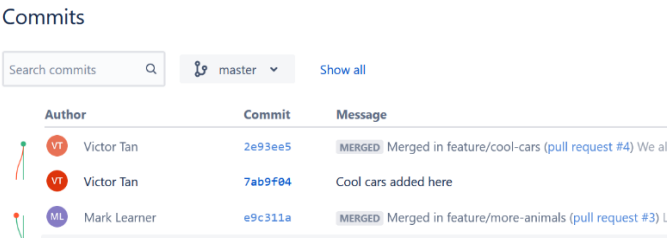
Next, Dev A will merge this branch into the remote main / master by selecting the Merge option from the triple dot drop down menu.

This brings up the Merge Pull request dialog box where you can provide the commit message and fix the merge strategy. You can also decide whether to close the feature branch once the merge is complete. We will accept the default options provided and click Merge.



After a while, the merge should complete successfully in the background and the main Pull Request view should transition to indicate this:

If we head back to the main Commits view for the master branch, we should be able to see the merged in commits from the branch.



Once the merge has been completed, it is important that all team members pull the latest version of the remote master to update the master branches in their local repo.

Return to the Git Bash shell for `devA/firstsharedrepo`.

Switch to the `master` branch (if you are not already there) with:

```
$ 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 we just completed a merge operation on it. Again, this is because we will not have the latest status of the remote repo until we execute:

```
$ git fetch
```

Next, we check the status:

```
$ git status

On branch master
Your branch is behind 'origin/master' by 2 commits, and can be fast-
forwarded.
  (use "git pull" to update your local branch)

nothing to commit, working tree clean
```

Next, we perform a pull:

```
$ git pull
```

We can check the commit history of `master` with:

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

You should see that the commit history here now matches that shown in the Commits view in the browser for the `master` branch

Now the final step is to merge in the latest updates from the local `master` into the local `feature/boring-cars`. This is important to ensure that this feature branch that Dev A is currently working on incorporates the latest changes as a result of updates from other devs in the team.

```
$ git checkout feature/boring-cars

$ git merge master

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

This time, the 3-way merge results in a conflict and the Git Bash shell changes to indicate this status as we have already seen in a previous lab.

Open the file containing the conflict (`cars.txt`). You should see a section of the file where Git has automatically flagged the conflict:

```
<<<<<<< HEAD
5: mitsubishi
6: hyundai
=======
5: Aston Martin
6: Porsche
>>>>>>> master
```

The `<<<<<<<` indicates that the content below is from the current branch that is being merged into.
The `>>>>>>>` indicates that the content below is from the branch that is being merged from
The `=======` is a divider between the conflicting content

At this point, both Dev A and Dev B will need to work together using suitable collaboration tools to resolve the conflict appropriately.

Let's assume the conflict is resolved as shown below.

```
5: mitsubishi
6: Porsche
```

cars.txt

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

```
$ git commit -am "Merged cool and boring cars successfully !"
```

Notice now that the prompt no longer has the additional term MERGING at the end of it, indicating that the merge operation is now complete.

Check the commit history:

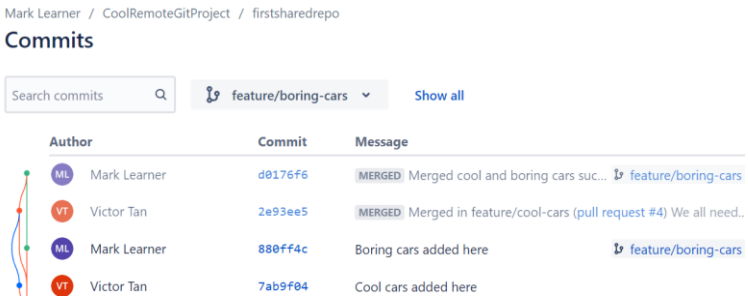
```
$ git log --oneline -n 4 --graph
```

Once again, we see that the latest merge commit has two parent commits from the two branches being merged.

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

```
$ git push
```

You should see that the Commits view in the browser for the `feature/boring-cars` branch commit history now matches that of the local branch.



The possibility of merge conflicts arising when merging the latest changes from the remote / local master into an ongoing feature branch is a very good reason why all team members should keep track of activity in all ongoing pull requests. This way, they can be alerted in advance of possible content updates to the feature branch related to that pull request which may possibly conflict with their own feature branches. They can then proactively engage with the dev concerned in the ongoing discussion on that pull request to prepare for the inevitable conflict resolution process ahead in advance.

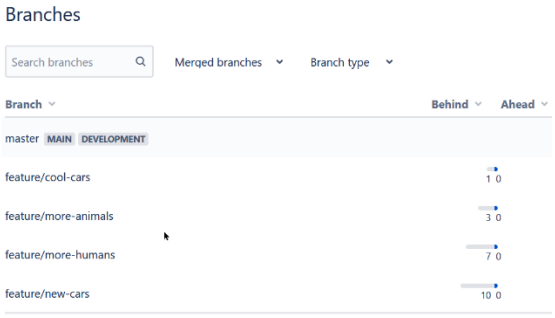
24 Deleting merged feature branches from remote / local repos

Step p) in the sample collaborative development workflow

So far, we have worked with a variety of feature branches that were uploaded to the remote repo and subsequently merged into the remote master. We can archive these feature branches in the local / remote repos for later study and reference, however we can also choose to delete them to prevent a proliferation of unused branches.

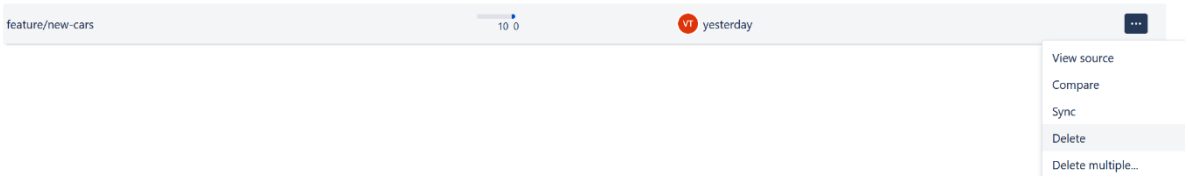
Typically, the project lead / manager will perform the deletion of these branches, although the dev responsible for that branch can also delete it.

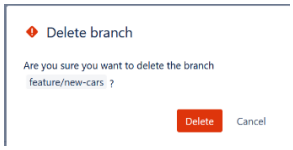
In Dev A's browser, check in the main Branches view for Merged branches (from the drop down list). We specifically search for merged branches since we will not typically delete a branch until it has been merged into the remote master.



Notice that all these branches are behind the master branch by different number of commits and none of them are ahead of the master branch (which is expected since there was no further work on them after they were merged into the remote master).

We will select the `feature/new-cars` branch for deletion via the drop down menu available from the Actions column in that specified row.





Return to the Git Bash shell for `devA/firstsharedrepo`.

Switch to the `feature/new-cars` branch and check on the status with:

```
$ git checkout feature/new-cars

$ git fetch

$ git status

$ git branch --all
```

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

Now if we perform a:

```
$ git remote show origin
```

```
…..
…..
    refs/remotes/origin/feature/new-cars  stale  (use  'git  remote
prune' to remove)
```

Here, we get a clue that there is an issue with the upstream `feature/new-cars` branch. The remote tracking branch for it (`origin/feature/new-cars`) 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/firstsharedrepo.git
 * [pruned] origin/feature/new-cars
```

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

```
$ git status

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

nothing to commit, working tree clean

$ git branch --all

  feature/boring-cars
  feature/more-animals
* feature/new-cars
  master
  remotes/origin/feature/boring-cars
  remotes/origin/feature/cool-cars
  remotes/origin/feature/more-animals
  remotes/origin/feature/more-humans
  remotes/origin/master
```

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

```
$ git checkout master

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

And finally if we check again with:

```
$ git remote show origin

$ git branch --all
```

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

Next, we will see how to delete remote branches using the Git Bash shell. Let's delete the local and remote version of `feature/more-animals`.

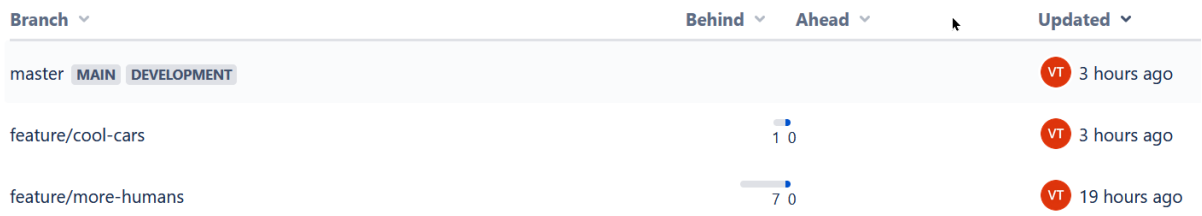
In the Git Bash shell for `devA/firstsharedrepo`, we will delete the local branch first with:

```
$ git checkout master

$ git branch --delete feature/more-animals
```

Then we delete the upstream branch with:

```
$ git push origin --delete feature/more-animals

To https://bitbucket.org/mark-coder/firstsharedrepo.git
 - [deleted]         feature/more-animals
```

Now if we check the main Branches view in the browser, we should no longer be able to see this branch:



Let's repeat this process for Dev B.

In the Git Bash shell for `devB/firstsharedrepo`, we can check on the status of the remote branches with:

```
$ git fetch

$ git branch --all
```

As expected, we can see that the remote tracking branches `remotes/origin/feature/more-animals` and `remotes/origin/feature/new-cars` are still present even though they are already deleted from the remote repo.

Again, if we perform a:

```
$ git remote show origin

* remote origin
…
…
    master                                      tracked
    refs/remotes/origin/feature/more-animals  stale  (use  'git  remote
prune' to remove)
```

```
    refs/remotes/origin/feature/new-cars      stale (use 'git remote
prune' to remove)
```

we get the indication of which branches are stale. We can then prune them with:

```
$ git remote prune origin
```

after which the following command provides the correct info:

```
$ git branch --all
```

We can then proceed to delete the local version of `feature/new-cars`

```
$ git checkout master

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

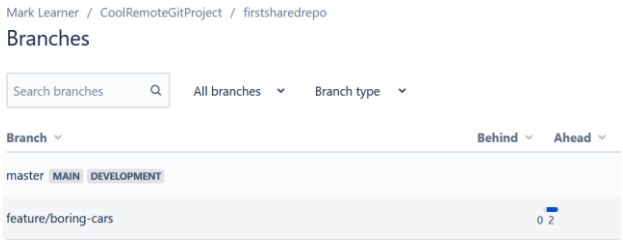
Finally, we can proceed to delete the local and remote versions of `feature/cool-cars` and `feature/more-humans`

```
$ git branch --delete feature/more-humans

$ git push origin --delete feature/more-humans


$ git branch --delete feature/cool-cars
```

You will get some error messages about not yet being merged into HEAD (which is pointing to master) when deleting `feature/cool-cars` . This is because even though the upstream version of this branch was merged into the remote master, we had not yet updated the local master in `devB/firstsharedrepo` before performing the delete operation.

```
$ git push origin --delete feature/cool-cars
```

At this point, when we check all branches in the main Branches view in the browser, we should see there is only a single remaining unmerged remote branch `feature/boring-cars`:



When we check the main Commits view, we will only be able to see two branches that we can still access (`master` and `feature/boring-cars`), but we will still be able to see the colored lines representing the history of different branches in the overall commit history.

25 Resolving merge conflicts from parallel work on feature branch

So far we have seen, merge conflict arising from work by different devs on different feature branches which involve changes to the same file content. Another possibility for merge conflict occurring is when two devs are working on the same remote feature branch simultaneously, but are not coordinating their efforts properly (for e.g. through the pull request discussion forum).

Let's simulate a situation such as this involving Dev A and Dev B.

We will work using the `feature/boring-cars` branch created by Dev A in a previous lab that still remains unmerged.
In the Git Bash shell for `devB/firstsharedrepo`, first update the content of the local version of this branch (if it has not yet been updated):

```
$ git checkout feature/boring-cars

$ git pull
```

Inside `devB/firstsharedrepo`, add the following lines to this existing file. Make sure you include a new line after the end of the single line

```
7: Rolls-Royce
8: Jaguar
```

cars.txt

Add this into the new commit on with:

```
$ git commit -am "British cars added here"
```

To view the commit history of this new branch, type:

```
$ git log --oneline -n 4
```

Upload this commit to the upstream branch at the remote repo with:

```
$ git push
```

Now if Dev A is also collaborating on this same feature branch, they should pull down the latest updates from the upstream branch first before making any new changes to their local branch. Let's assume that they forgot to do this, and made a new commit on their local branch first:

In the Git Bash shell for `devA/firstsharedrepo`, switch over to this branch:

```
$ git checkout feature/boring-cars
```

Inside `devA/firstsharedrepo`, add the following lines to this existing file. Make sure you include a new line after the end of the single line

```
7: Saab
8: Volvo
```

cars.txt

Add this into the new commit on with:

```
$ git commit -am "Swedish cars added here"
```

To view the commit history of this new branch, type:

```
$ git log --oneline -n 4
```

At this point of time, the content of this branch is divergent from its upstream counterpart and there is conflicting content as well.
Let's see what happens when Dev A attempts to upload this commit to the upstream branch with:

```
$ git push

To https://bitbucket.org/mark-coder/firstsharedrepo.git
…
…
hint: See the 'Note about fast-forwards' in 'git push --help' for
details.
```

The error messages that appear essentially point out that there is one or more existing commits in the upstream branch that does not exist in the local branch (the `British cars added here` created by Dev B earlier). The attempt to merge these two divergent branches results in a merge conflict WHICH MUST be resolved locally. There is no way to resolve a merge conflict in a remote repo.

We will attempt this merge locally instead now with:

```
$ git pull

remote: Enumerating objects: 5, done.
….
….
Auto-merging cars.txt
CONFLICT (content): Merge conflict in cars.txt
Automatic merge failed; fix conflicts and then commit the result.
```

The standard error messages regarding a 3-way merge conflict appear and the Git Bash shell changes to indicate this status as we have already seen in a previous lab.
Open the file containing the conflict (`cars.txt`). You should see a section of the file where Git has automatically flagged the conflict:

```
<<<<<<< HEAD
7: Saab
8: Volvo
=======
7: Rolls-Royce
8: Jaguar
>>>>>>> bcf1484ffcfb08f486e516870e1bf66fb8e0a83f
```

The `<<<<<<<` indicates that the content below is from the current branch that is being merged into.
The `>>>>>>>` indicates that the content below is from the branch that is being merged from
The `=======` is a divider between the conflicting content

At this point, both Dev A and Dev B will again need to work together using suitable collaboration tools to resolve the conflict appropriately.

Let's assume the conflict is resolved as shown below.

```
7: Saab
8: Volvo
9: Rolls-Royce
10: Jaguar
```

cars.txt

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

```
$ git commit -am "British and Swedish cars together. Finally !"
```

Notice now that the prompt no longer has the additional term MERGING at the end of it, indicating that the merge operation is now complete.

Check the commit history:

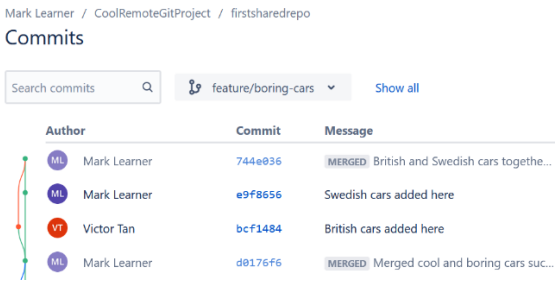
```
$ git log --oneline -n 4 --graph
```

Once again, we see that the latest merge commit has two parent commits from the two branches being merged.

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

```
$ git push
```

You should see that the Commits view in the browser for the `feature/boring-cars` branch commit history now matches that of the local branch.



To avoid this issue, simply ensure that a feature branch is only ever worked on by a single dev until its complete and merged into the master branch. If this cannot be achieved, then ensure the devs collaborating on the master branch coordinate to take turns adding and committing changes sequentially to the feature branch involved, rather than all of them working on it in parallel.