Department
for Education

DfE Statistics Development Team Workshops

Using git and Dev Ops

Contents

Introduction

We've prepared this walkthrough guide for statistics publication teams as an introduction on how to work collaboratively with git and Dev Ops, using some typical data manipulation as an example. The guide is intended to be step-by-step, building up from the very basics. The plan is to work through this in groups of 3-ish and with access to experienced git users to call on for support. If it starts too basic for your level, then just go through at your own/your group's pace as you see fit. By no means can we cover everything in this walkthrough, so please see it as a prompt to ask follow-up questions as you're working through on anything related to git, Dev Ops and GitHub.

We're focusing on Azure Dev Ops rather than GitHub here, but much of the material is transferable.

GitHub versus Dev Ops

GitHub and Dev Ops effectively provide the same service in terms of creating software via a git repository: they both act as the host for the remote repository, whilst offering important tools to manage bugs and issues, tasks, merging branches, deploying applications and so on.

Dev Ops is part of the Microsoft Azure platform and uses pricate DfE servers. This can allow you to connect or deploy your repository into wider Azure services. This includes SQL databases that you might already be storing data on as well as the DfE's implementation of rsconnect on DfE internal servers, which allows deployment of shiny apps for internal DfE use.

GitHub is hosted on external servers and therefore is more appropriate for making your code or application available for public access and use. For example, from a GitHub repository, you can deploy an R Shiny dashboard to shinyapps.io where members of the public may view and interact with your published data.

Pre-workshop requirements

Technical requirements

First of all, make sure to bring your laptop. This is going to be interactive and require you to do some coding.

Preferably before coming along, you'll need to go through the following list of things you'll need to make sure are set up on your DfE laptop:

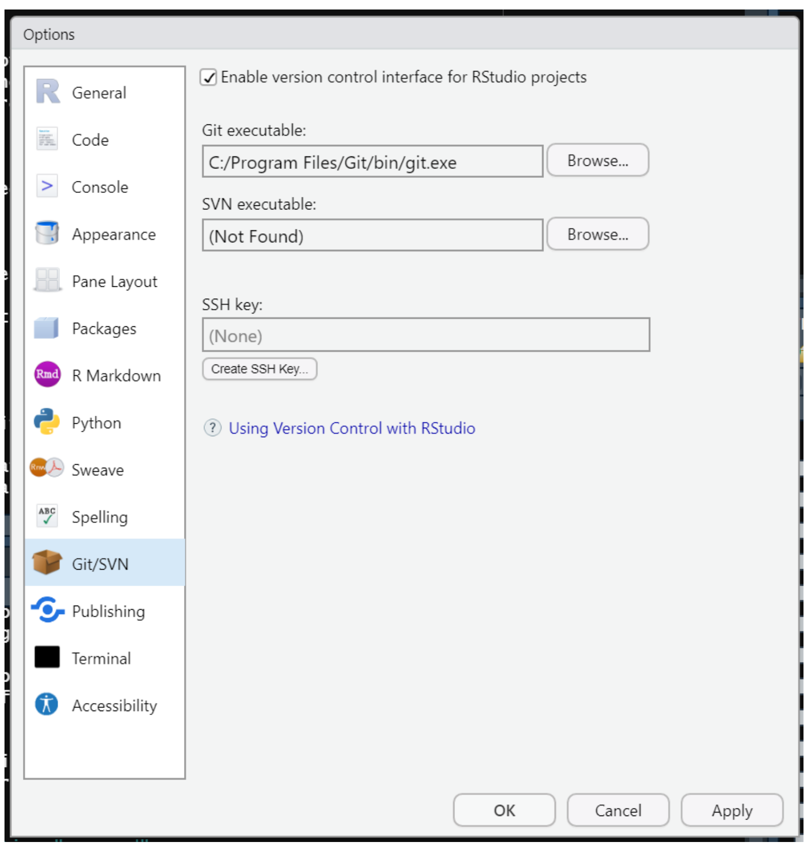
- Set up an Azure Dev Ops Basic account (not a Stakeholder account) at the DfE Service Portal; Either:
- Install git on your laptop: https://git-scm.com/downloads;
- Install R-Studio on your machine: Download **R for Windows (x64)** and **RStudio** from the Software Centre on your DfE laptop.

Or:

- If you're on EDAP and used to using R/R-Studio and/or git on there, feel free to just use that.

You'll also need to make sure that git is set up in the git/SVN pane of global options in R-Studio (found in the Tools drop down menu). Make sure the path to your git executable is entered in the git path box and git should automatically be integrated with R-Studio.

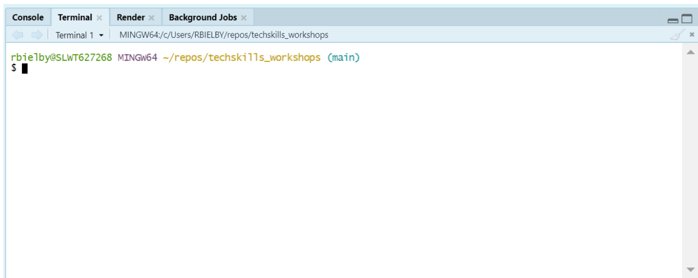
Figure 1: Enter the path to your git executable in the git path option box



Once you open a repository, you'll get an extra panel, named 'git', in the top right pane of

R-Studio and you'll also be able to use git in the 'Terminal' tab at the bottom left (in the same place as the R console).

Figure 2: The 'git BASH' terminal in R-Studio



A useful thing here if you want to use git commands in the terminal is to switch the terminal from the default Windows Command Prompt to `git BASH`. You can do this in the Terminal tab of R-Studio's global options - just select `git BASH` from the 'New terminal opens with' pull down menu. Click apply and then select the Terminal tab (next to the Console tab), click 'Terminal 1' and then select 'New terminal' from the drop down menu. You should see something similar to the terminal screenshot.

Working in teams

To get the most out of git and Dev Ops, you're going to need to work in teams. We're aiming for groups of 3. Some of the tasks we'll work through will require just one of your team to perform, whilst others will require all of your team to perform them. If it's not clear then ask and most importantly, communicate with each other about what you're doing.

Getting started in Dev Ops

Navigating around Dev Ops

There's a lot to Azure Dev Ops, but we're going to focus today on a handful of key elements arund task management and repositories:

- Boards
  - Boards
- Repos
  - Files
  - Commits
  - Branches
  - Pull Requests

If you follow the link below, that should take you to our demo repository for this workshop. Follow the link and explore the tabs on the left of the page that are listed above.

Azure Dev Ops workshop_devopsintro

The key things to note at this stage is that: * The Boards area contains useful project management tools for tracking development of your code. Today we'll just use the Kambam board style functionality, but there's plenty more. * The Repo area contains any repositories you have access to. You can navigate between repositories using the dropdown menu at the top of the page.

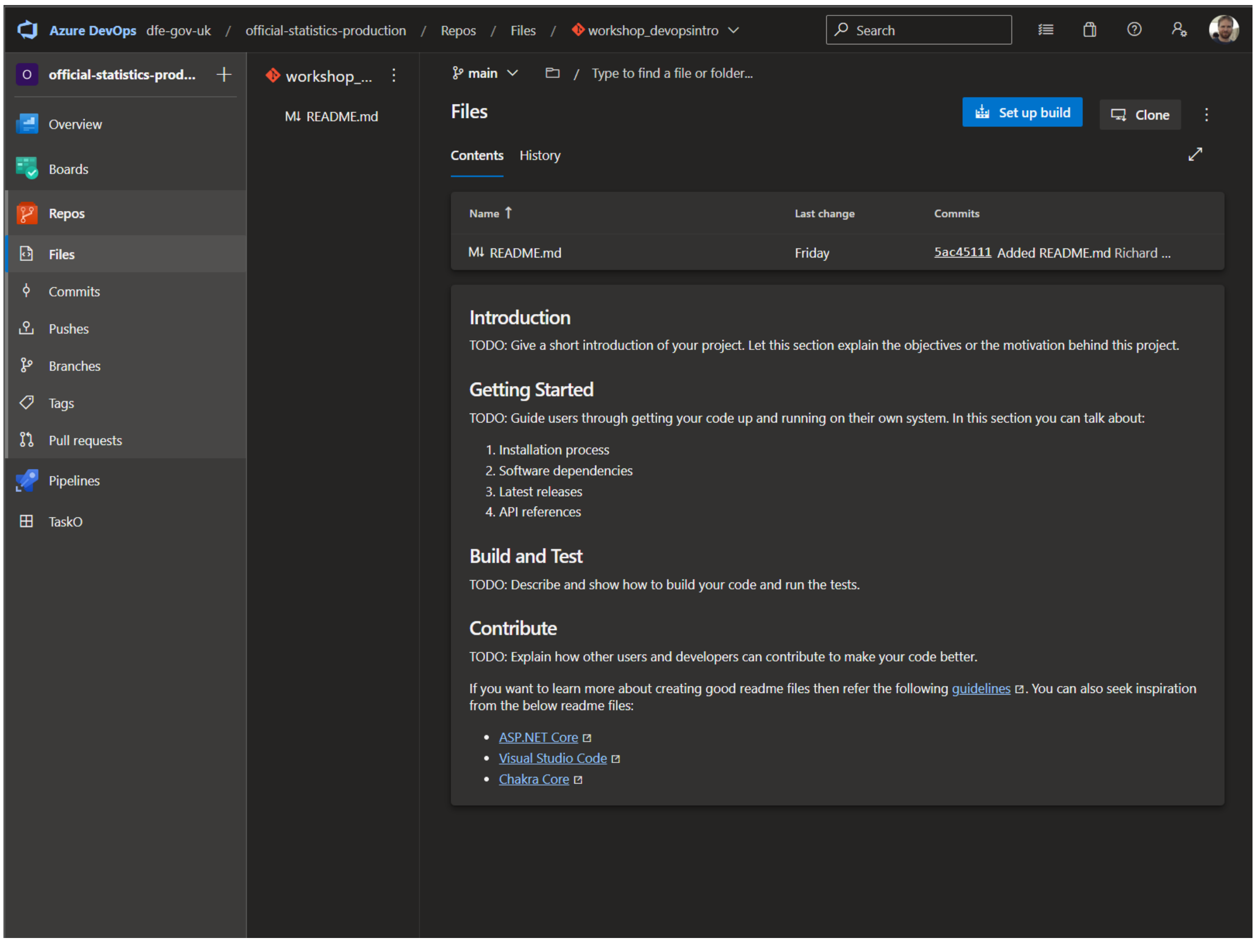
What's a repository?

A repository is a folder containing a related set of code and associated files. So for example, it can contain your full data processing pipeline or it could contain a dashboard app.

The repo on Dev Ops represents the central safe copy of your repository and is called the **Remote** repository. It contains all the history and current code of your code and should also contain at least some associated documentation (primarily in the form of the Readme.md file).

You and your team can also have as many copies (or clones) of the repository on your workstations. These are the **Local** repositories. Any updates that you and your team make to the files in your local copies of the repository should be synced to the remote repository

Figure 3: An example repository front page on Dev Ops



at regular intervals, such that all team members can access the latest versions of the files.

Cloning the repository to your local machine

Cloning the repository refers to creating a copy of the remote repository (i.e. the copy on GitHub or Dev Ops) on the disk on your local machine (i.e. your DfE laptop). For an R project, there are two basic options to choose from for doing this:

- using the R-Studio new project wizard, or
- using `git BASH`.

We'd recommend trying the different options across your working group.

Before starting either option, you'll need to copy the repo url from Dev Ops. To do this, open up the repo front page in Dev Ops (you can use the link from earlier) and click the grey **Clone** button near the top right of the page. Make sure HTTPS is selected and then click the copy button to grab the url.

Cloning in `git BASH`

You can open up a `git BASH` terminal, by typing `git BASH` in the Windows search bar and select `git BASH` when it comes up. With a terminal, you can interact with it just by typing, similar to working in the R console in RStudio. First let's make a directory in which to store our repositories:

```
mkdir repos
```

We can then move into the directory we just created using:

```
cd repos
```

Now grab the repo url and replace `<repo_url>` in the next command with the actual url:

```
git clone <repo_url>
```

You should get some messages letting you know git is connecting to the server and cloning the repository and it should look something like the figure below.

If all went well, you'll now have a complete copy of the repository on your laptop. To open the repository in RStudio, start up RStudio and select Open project. In the file explorer window that opens up, type `C:\Users\` and hit enter (see the screenshot below) and then open up your home folder.

Figure 4: Cloning a repository in git BASH

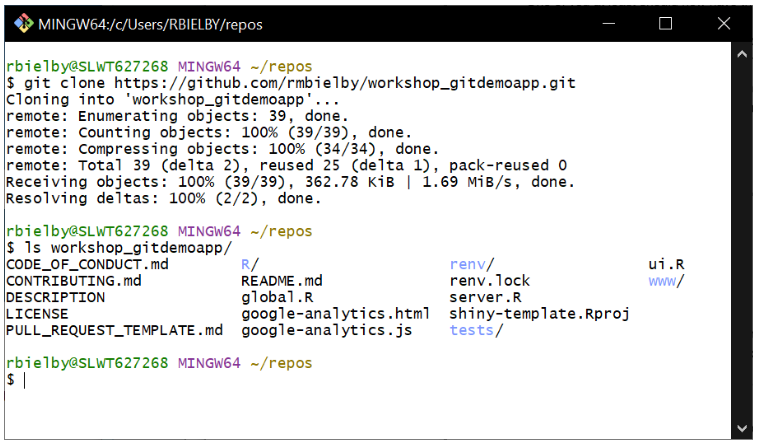
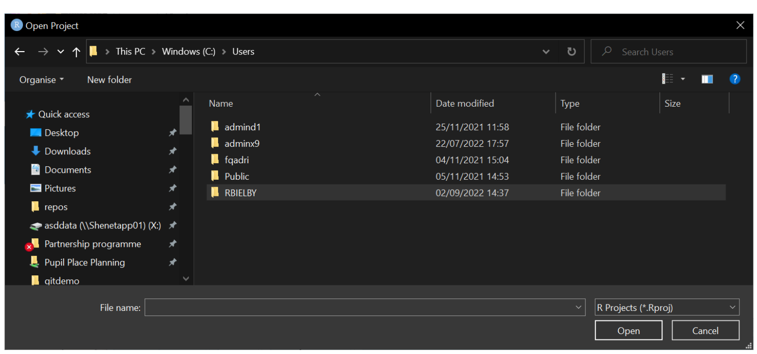


Figure 5: Open a cloned project in RStudio

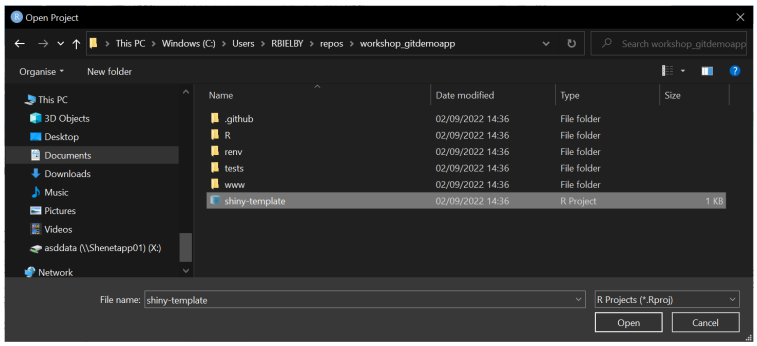


Then navigate into `repos` and the repository folder. The full path should be something along the lines of:

```
This PC > Windows (C:) > Users > <USERNAME> > repos > <REPONAME>
```

Select the R project file and select open.

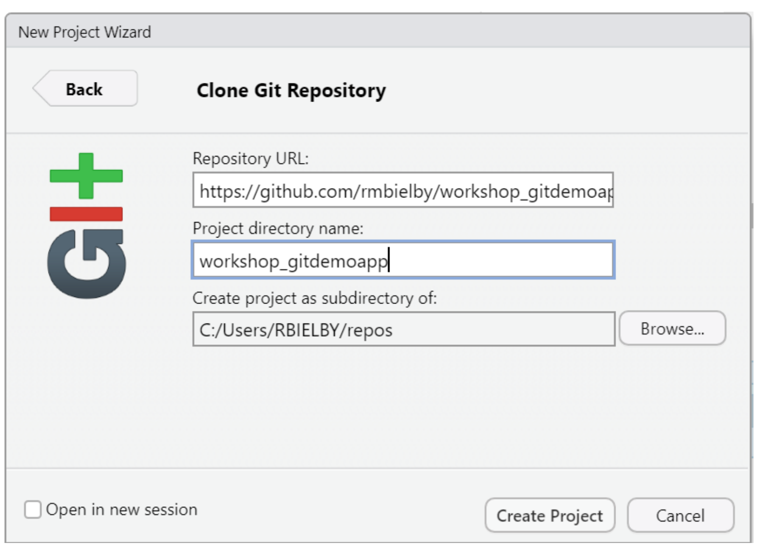
Figure 6: Open a cloned project in RStudio



Cloning using the RStudio wizard

If that looks like a bit too much text based effort, RStudio offers a way to clone a repository with it's New project wizard. To do this navigate the menu bar to **File > New Project. . .** , select **Version Control** and then Git. This opens up a dialogue box to enter the repository url and select where to save it. As with the git BASH version, copy and paste your remote repo URL here and set a directory where you want it saved on your laptop.

Figure 7: Clone a project using the RStudio git wizard



A note on local repository clones and OneDrive

> Note that saving a repository within your OneDrive folder structure can cause some awkward issues. If you use git to perform version control on a repository saved within a OneDrive folder, you may start receiving warning messages that large numbers of files have been removed from OneDrive. In addtion, it can put a heavy burden on your internet connection as OneDrive tries to keep up with changes to the files managed by git. Best practice therefore is to store your repositories somewhere outside your OneDrive file structure. We recommend creating a `repos` directory within your base User directory (i.e. `C:\Users\<USERNAME>\repos\`. Windows sometimes tries to make it awkward for you to navigate to places on your laptop outside of the OneDrive folders, so a useful tip is to add your `repos` folder to your Quick access list in File Explorer.

Using Dev Ops Boards

We're going to work through a demo of tidying up some data and running a quick QA on it.

To start with, we'll create some tasks on the Kambam Board in Dev Ops. The tasks to create are:

- Loading and tidying the data
- Creating the wide data plot
- Creating the tidy data plot

Go to the *Boards* > *Boards* tab on Dev Ops and make sure you're in the right board, i.e. workshop_devopsintro_N. Now create tasks for each of these using the New item button.

Summary

In this section, we've had a quick look around Dev Ops and the remote repository, cloning it to your local drive (using both the BASH terminal and the RStudio wizard) and creating and assigning tasks on the Dev Ops Boards area.

In the next section, we'll cover some of the basics of using git to log changes to your code and sync them between the remote repo and local copies. If you open up one of the tasks, you'll see you can add descriptions, comments, tags, priorities and story points (effectively an estimate of the effort or time required) among other things. We can leave those for the purpose of this demo and for now decide who's going to do what and drag each task into the Active column. Note that whoever drags a given task across will automatically be assigned as the person working on that task. If that's not the right person, then click on the name and select who should be doing it.

We'll take a break from Dev Ops now and move on to taking a look at using git.

Basics of git

We'll now take a look at updating repositories using some simple processing and QA code as an example.

The git log

In order to move quickly between different versions of files and code, git is built around indexing and a log file that track the changes in a repository. To view the log of any repository, we can simply go into that repository and run the command `git log` from the BASH terminal. For example, if you run it in the repository you've created from the template, you'll get something along the lines of the following:

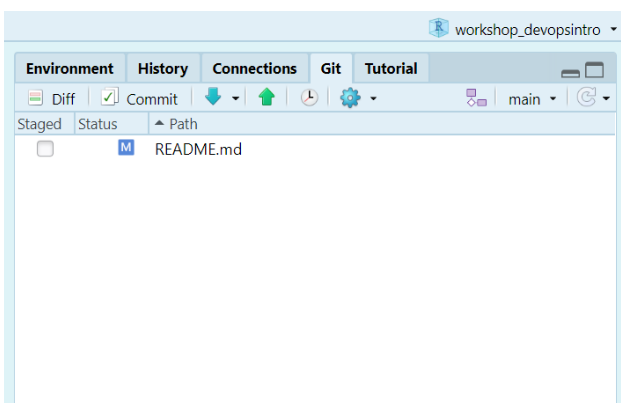
```
rbielby@SLWT627268 MINGW64 ~/repos/workshop_gitdemoapp (main)
$ git log
commit 467df8800a108c0e1c98cb8db5ff0f377db8ffbb (HEAD -> main, origin/main, origin/HEAD)
Author: Rich Bielby <richbielby@e.email>
Date:   Fri Aug 19 17:27:36 2022 +0100

    Initial commit
```

The log shows all "commits" that have been made to the repository. We'll go into making commits in the next section.

`git bash` versus the *R-Studio* git panel

Figure 8: The *R-Studio* git panel provides all the common day to day git commands such as Stage/Add, Commit, Push and Pull, switch branches, view history.



There are two main ways you can run git commands within R-Studio, either using the `git bash` terminal or the *R-Studio* git panel Each offers some advantages, but the main ones are that `git bash` offers the full range of tools for controlling your repository, whilst the

Figure 9: The *R-Studio* git bash terminal provides access to all git functionality via a command line interface.

R-Studio git panel offers the most common basic commands but with a simpler (and usually quicker) interface. We'll use both in this workshop.

Branches

One of the most important elements of using `git` is **branches**. These provide a method to keep multiple different copies of your code in a single repository. This is usually intended to maintain a working base copy alongside one or many development branches with new features, updates or bugfixes.

In this workshop, we've created a branch for each team, so you need to start by switching to the branch for your group - one of workshop_group_1, workshop_group_2, workshop_group_3 etc. As in most cases here, you can use either the RStudio git panel or `git bash` to switch branches.

Switching branches in `git bash`

In `git bash`, you can change branches using either the `switch` or `checkout` command. Go to the bash terminal in R-Studio and enter the command:

```
git checkout workshop_group_1
```

Updating the number to your group number. To switch back to main, you can simply use:

```
git checkout main
```

Switching branches in the *R-Studio* git panel

In the git panel, the name of the current branch is given next to the purple create branch symbol. Click on the name of the current branch (which should be *main* at present) and a drop down list will appear. Select your group's branch from this list and R-Studio will automatically switch to that branch. Anything you add and commit to the repository will now be written to the branch you've selected and not *main*.

Adding, commiting and pushing

To have something to work with, we need some data. There should be a data folder in the repository already, so all we need to do is grab some data and save it there. For this workshop, we'll use a file from a publication on Explore Education Statistics.

For this section, just one of your group should run through the following steps in your group's branch:

Go to key stage 4 performance publication and download the **KS4 subject timeseries data (csv, 364 Kb)** file, extract the data csv file (*2021_subject_timeseries_data.csv*) and save it into the repo's data folder (you can just use the normal Windows File Explorer to do this).

Now to add this to the git tracking: run the following commands:

```
git add .
```

This searches the repo for any files that have been modified since the last commit and creates a log of the changes. If you want to check that the command has worked, then you can type `git status` or `git st` and you'll get a summary of files that have been staged and are ready to commit.

```
git commit -m "Added data file into repository."
```

This adds an entry on to the log, updating it with the file changes that you've just made. Note that the text after the `-m` is a comment used to describe the changes to make it easier for someone looking back from the log to see what changes have happened. Those are the two key steps for tracking changes to the files and folders in your repository.

Now we'll add in some simple code to read in the file. Create a script in the repository's root directory called `main.R`. The add the following code to it:

```
dfKS4 <- read.csv('data/2021_subject_timeseries_data.csv')
colnames(dfKS4)[1] <- "time_period"
```

Let's do a quick commit to log that change. In the termainl run `git add .` and `git commit -m "Added new data and reading it in."`.

If we now run `git log` again, we get something along the lines of:

```
rbielby@SLWT627268 MINGW64 ~/repos/workshop_gitdemoapp (main)
$ git log
commit a57f66898df18eb43a685a414b9ab805301a6c01 (HEAD -> main)
Author: Rich Bielby <richard.bielby@education.gov.uk>
Date:   Tue Sep 6 18:30:30 2022 +0100

    Renamed data file.

commit 8d4ef984664a23d64fe0fbfa296729af0c23094c
Author: Rich Bielby <richard.bielby@education.gov.uk>
Date:   Tue Sep 6 18:28:44 2022 +0100

    Added data file into repository.

commit 467df8800a108c0e1c98cb8db5ff0f377db8ffbb (origin/main, origin/HEAD)
Author: Rich Bielby <richbielby@e.email>
Date:   Fri Aug 19 17:27:36 2022 +0100

    Initial commit
```

Here we can see, in reverse order, the commits that have been made, who made them, when they made them, and the messages that have been recorded with them.

Finally, it's important to note that what we've done so far is only being applied to the local copy of the repository (i.e. the copy on your laptop). To apply your changes to the remote repository (i.e. on GitHub or Dev Ops), you need to "push" the changes. This can be done a couple of different ways: a) in the terminal type `git push` or b) on the toolbar in the *R-Studio* git panel press the green up button! Once you've done this, open up a browser and go to your remote repository on Dev Ops and, once you've switched on to your branch, you should now see the data file stored there. You should also see that it's not been added on to the main branch.

Pulling from the remote repository

Now that you've made changes, the rest of your team need to update their own local copies of the repository with your updates by pulling from the remote. Similarly to pushing, they can do this by either a) typing `git push` in the BASH terminal or b) pressing the down arrow in the toolbar of the git panel in *R-Studio*.

Summary of git basics

We've quickly tried out a quick cycle of adding and committing, which is used to log changes into the local repository and then we've pushed and pulled to and from the remote repository and local copies on different laptops. The table below gives a summary of the relevant commands in the BASH terminal and the corresponding buttons in the

RStudio git panel.

Process	git BASH	RStudio git panel
Add	`git add .`	Stage using tickbox next to each modified file.
Commit	`git commit -m "Commit message."`	“Commit” button in toolbar.
Push	`git push`	Green up arrown in toolbar.
Pull	`git pull`	Blue down arrow in toolbar.
View the log	`git log`	Clock icon in toolbar.

Working collaboratively with git

Git only really makes proper sense once multiple people start working on a project collaboratively. Solo working, git is useful for version control and syncing your work to a remote repository site like Dev Ops and GitHub, but may not feel like it offers all that much more beyond that. Once we start working collaboratively however, the benefits of using git (alongside GitHub or Dev Ops) become more apparent. We'll now look further into this with some worked examples.

Branches and splitting tasks

Task management

One useful management tool that we can use from Dev Ops is the *Boards* area. Here we can create individual tasks, assign them to team member and then create new **branches** from those tasks. You can think of **branches** as self contained copies of the repository that can contain complementary or conflicting differences with all other **branches** in the repository. These allow you to work on different multiple tasks on your code independently of any other changes you might be making. Bringing these different **branches** or tasks together is then managed using **merges** or **pull requests** (PRs).

We'll demonstrate this by performing 3 related tasks on 3 different branches. Each task should be done by only one of your group, so split the following 3 tasks between your group and each follow the relevant instructions in the subsections below:

- 1a) Loading and tidying the data
- 1b) Creating the wide data plot
- 1c) Creating the tidy data plot

Once you've decided who's doing what, each of you should jump to the relevant section below. And remember that you're not working in independent silos here, what you do can impact what other people are doing so communication needs to happen along the way.

Loading and tidying the data This task underpins the next two. You need to read in the data file that we've already downloaded. Start by creating a new branch. Note that when you create a new branch, git will create it as a copy of the current branch that you're in, so make sure that you're in your groups working branch (i.e. workshop_group_X). In the `BASH` terminal, and type:

```
git checkout -b featReadData
```

This creates and moves you into a new branch called `featReadData`. Anything you do here has no affect on the contents of the `main` or `workshop_group_X` branch until you explicitly tell git to merge your changes. Note as well that this only creates the branch in your local copy of the repository, for it to be added to the remote, you'll need to push:

```
git push --set-upstream origin featReadData
```

If you only enter `git push` without the `--set-upstream` flag, git will suggest the full command for you, which you can copy and paste.

Then copy and paste the following code into the script `R/utils.R`:

```
tidy_subject_timeseries <- function(dfin){
  # Pivot long and then update the string values in the new filter column.
  dftidied <- dfin  %>%
    pivot_longer(!c(time_period,time_identifier,geographic_level,
                    country_code,country_name,version,characteristic_gender,
                    subject,entries),
                 names_to="level",
                 values_to="percentage_pupils")
  colnames(dftidied) <- gsub("characteristic_","",colnames(dftidied))
  return(dftidied)
}
```

And then go back to `main.R` and add the lines:

```
source('R/utils.R')
dftidy <- tidy_subject_timeseries(dfKS4)
```

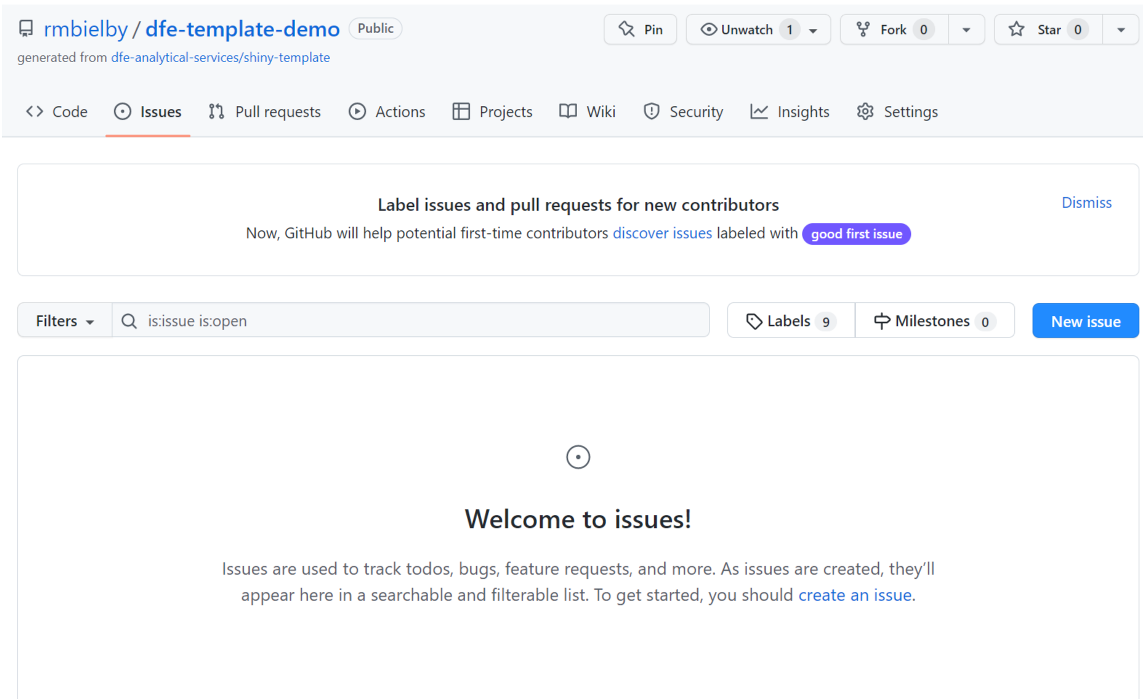
Now you could try sourcing `main.R` in the R console and that should create the data frame dftidy (and hopefully not produce any errors!).

Once you're happy, then run another `add/commit/push` cycle and flag to your team that you've finished the code to read in the data. Then scroll down this guide to the create a pull request section.

Creating the wide data plot Whilst the first task adds in some data, reads it in and does some processing, this task builds a quick chart based on the data.

Create a new branch using GitHub. Open up the repository on GitHub in your web browser and select the Issues tab. You should see something similar to shown below.

Figure 10: The Issues panel in GitHub. This can be used to keep track of jobs that need doing on a repository and create new branches linked to individual jobs.



On the issues tab, select *New issue* and then click **Get started** alongside **Feature request**. This will open a new issue panel, with boxes to put in some details about the feature being requested. Let's put in a title of *"Update inputs for new data"*. You'll see some handy guidance on what information to enter for an issue. Have a quick read of this, but then for brevity here, let's ignore it and put in a short description along the lines of "The data file being read in has been updated and the inputs needed updating to match the column headers etc.". You've got options to add assign this task and add labels, link to projects and milestones. We don't need much of that, but maybe add yourself as the assignee. When you're ready, click **Submit new issue**.

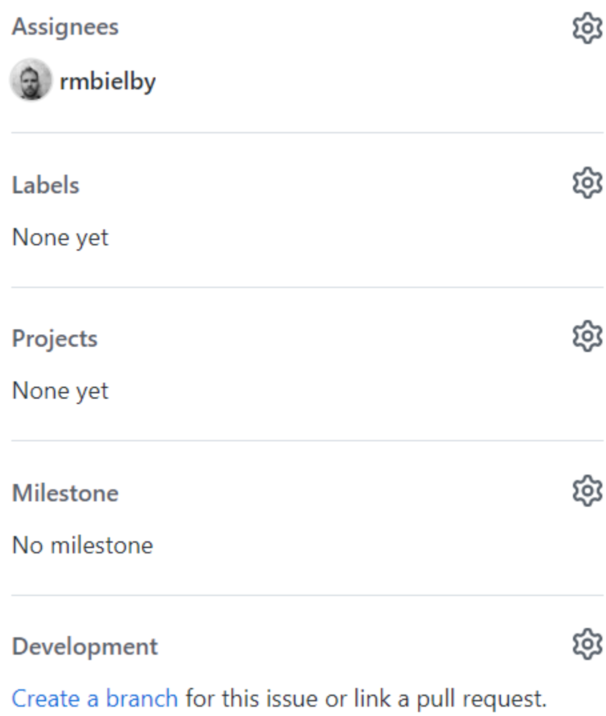
After you've done so, you'll see you now have the option in the right hand side panel to create a new branch linked to this issue. Select that and choose to check out the branch locally as the next option. Once you've confirmed you'll get some code to use to checkout the branch locally, which you can copy and use in the terminal in RStudio. It should look something like this:

```
git fetch origin
```

```
git checkout 1-update-inputs-for-new-data
```

Run that in the `BASH` terminal in RStudio and you'll have your new branch ready to go.

Figure 11: Once you've created an Issue on GitHub, you can create a new branch from the GitHub webpage and then fetch it to work on locally on your laptop.



So we've got a few small changes just to update here given the changes to the input data. Whoever's performing Task 1a, should be updating the data frame containing the new input data to be called `dfProgressHE` to reflect the different data that's being read in. The main changes that are needed are in the *server.R* and *R/dashboard_panels.R*.

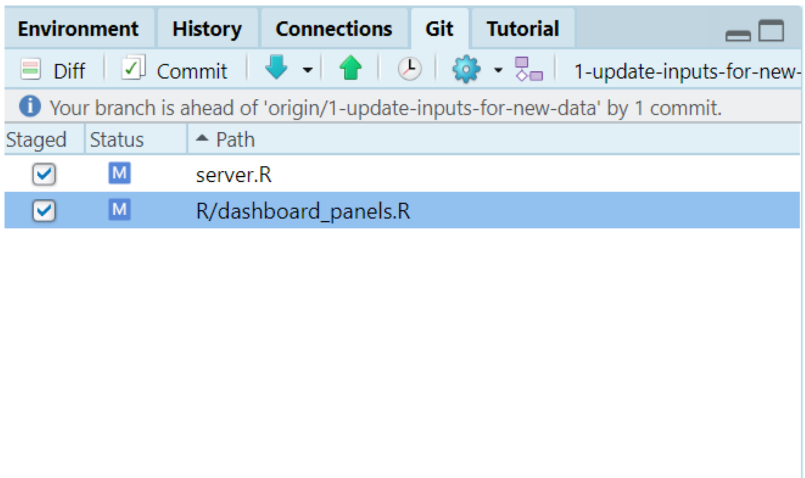
Add the following lines of code to `R/plots.R` in your local copy of the repository:

```
plot_ACrange <- function(df){
  dfplot <- df %>% filter(time_period>=201920,
                          !X94AstarC %in% c('z',':','x'),
                          grepl('Studies',subject)) %>%
    select(time_period,characteristic_gender,entries,subject,X94AstarC) %>%
    mutate(entries=as.numeric(entries)/1.e3,
           X94AstarC=as.numeric(X94AstarC))
  df1 <- dfplot %>% filter(time_period==202021,characteristic_gender=='Total')
```

```
  plot(df1$entries,df1$X94AstarC)
}
```

And that should be it for this task. All that's left is to commit and push your changes. If you've got a preferred way already to perform commits, then go for it. If not then let's use the RStudio git panel.

Figure 12: Staging files in the RStudio git panel.



Firstly click on the git tab in the top right of RStudio to show the git panel (see the screenshot below). Next click the tickboxes next to the files with changes (i.e. these should be server.R and R/dashboards_panels.R) to **stage** (aka **add**) the files. Now click **commit**, add a commit message in the relavent text box and then hit **commit** in the bottom right corner of the window.

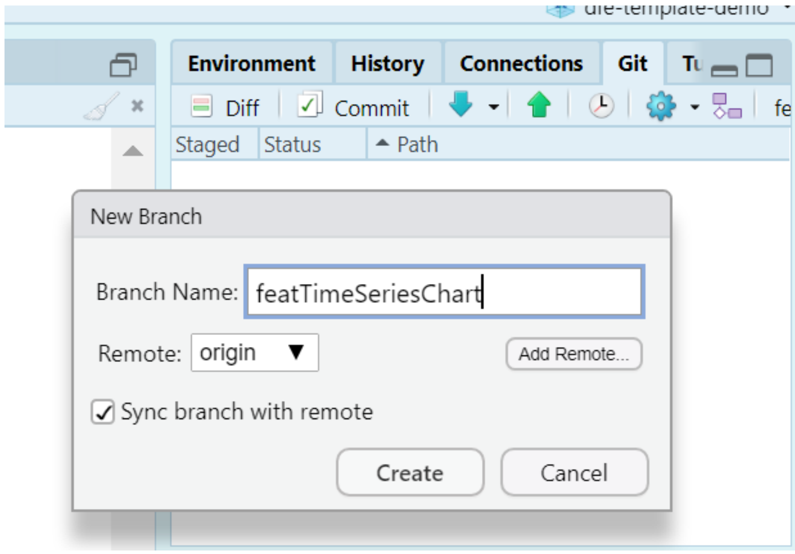
Assuming that all went through without any issues, you can now press the green up arrow in the git panel to **push** your changes to the remote repository on GitHub.

Let the others know that you're done!

Creating the tidy data plot There's already an example chart in the template, so let's re-wire that as the basis for our demo chart here. Let's start by creating a new branch in RStudio. With your repo open (and pulled with the latest changes) in RStudio, click the new branch button in the git panel toolbar. This will open a dialogue box as shown here in which you can enter a new branch name. Enter a name (e.g. *featTimeSeriesChart*), make sure syncing to the remote is selected and press *Create*.

Now in this new branch, you can make updates that won't effect any other branches until you choose to merge this branch with another one.

Figure 13: The new branch dialogue box in RStudio is opened using the purple new branch icon in the git toolbar.



To keep the code for the dashboard organised, we've split it into separate scripts and functions in different files. The main code for defining the line chart in the template is contained in `R/plotting.R`. Open that script now and change the lines:

```
createAvgRevTimeSeries <- function(dfRevenueBalance,inputArea){
  ggplot(dfRevenueBalance, aes(x=year,y=average_revenue_balance,color=area_name)) +
```

to:

```
createTimeSeries <- function(df,inputArea){
ggplot(df, aes(x=year,y=cohort,color=institution_group)) +
```

Then we need to update the code in *server.R* to reflect these changes to the plot, so update:

```
  ggplotly(createAvgRevTimeSeries(reactiveRevBal(), input$selectArea)) %>%
```

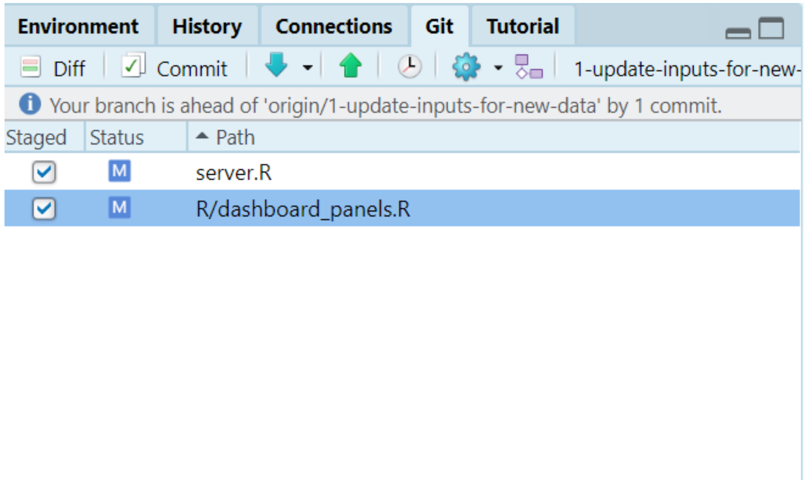
to

```
  ggplotly(createTimeSeries(reactiveRevBal(), input$selectArea)) %>%
```

This tells the plot to now take the variable `cohort` as the value to plot on the y-axis and `institution_group` as a filter on which to colour the output.

And that should be it for this task. All that's left is to commit and push your changes. If you've got a preferred way already to perform commits, then go for it. If not then let's use the RStudio git panel.

Figure 14: Staging files in the RStudio git panel.



Firstly click on the git tab in the top right of RStudio to show the git panel (see the screenshot below). Next click the tickboxes next to the files with changes (i.e. these should be server.R and R/dashboards_panels.R) to **stage** (aka **add**) the files. Now click **commit**, add a commit message in the relavent text box and then hit **commit** in the bottom right corner of the window.

Assuming that all went through without any issues, you can now press the green up arrow in the git panel to **push** your changes to the remote repository on GitHub.

Let the others know that you're done!

Wrap up

Provided you've all followed the instructions above, then you should each have used a different method to create a new branch. Try and take a few minutes here to discuss and compare how you each made a new branch.

Each comes with it's own benefits, limitations and challenges. How do you find each compare?

Some basic merging

Once a task is completed on a branch, it will likely need merging into another branch, for example the *main* branch. Once task 1a and 1b are complete, we should merge them into the branch for 1c so that all the changes are collated together. To do so, make sure you're in the *featTimeSeriesChart* branch (or equivalent depending on what you called it in task

1c) and run the following command in the BASH terminal (either in RStudio or in the standalone terminal):

```
git merge featReadData
```

That merges in the branch from task 1a. If you get any messages about merge conflicts, then you'll need to take some extra steps that we can help you through. If it merges in without any problems, then run the following to merge in the changes from task 1b:

```
git merge 1-update-inputs-for-new-data
```

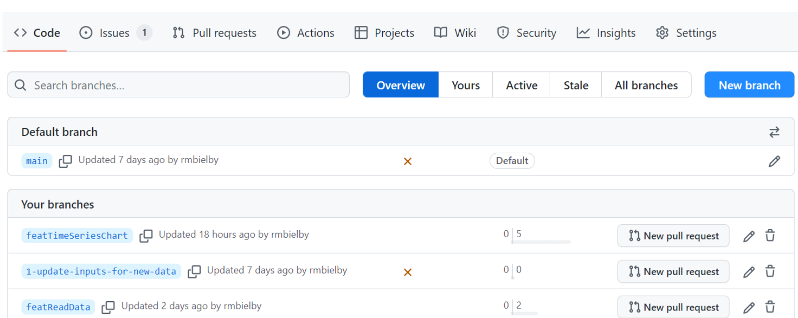
Provided you don't get any issues with the above merges, then you should now have all the updates from the three branches collected together in the final branch.

Merging with pull requests

Whilst the basic merges above work fine for pulling in some simple changes, using `git merge` via the terminal lacks any collaborative functionality like discussing and reviewing changes. This is where pull requests come in to play. Pull requests are a part of both GitHub and Dev Ops and provide similar functionality between those two platforms. Here we're using GitHub, but a lot of this will be transferrable to using Azure Dev Ops.

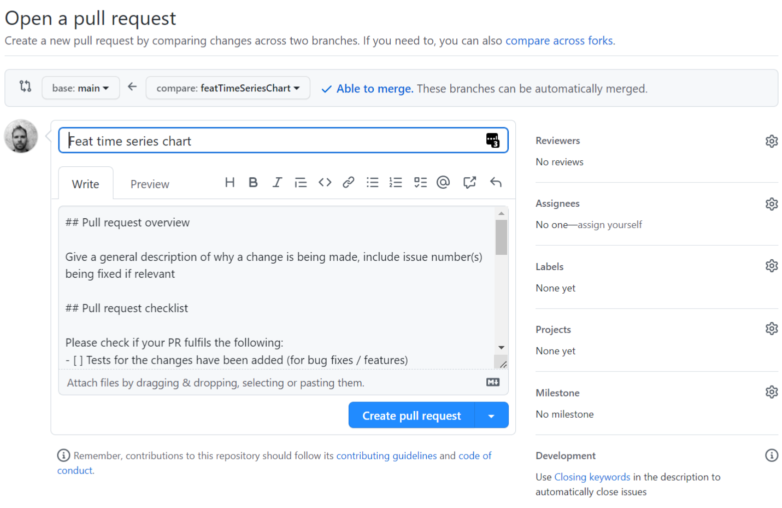
What we'd like to do now is to merge all the changes made so far into the *main* branch in the repository. To do so, whoever did task 1c should open up the repository on GitHub in their web browser and go to the branches panel under the code tab.

Figure 15: The branches overview pane in GitHub.



Click **New pull request** on the branch for task 1c, which now contains all the updates. That will open up the pull request page, where you can give some details for collaborators to understand what's in this update to the code. The pull request will default to performing a merge from the branch you've created it on into the *main* branch.

Figure 16: Opening a Pull Request on GitHub.



There are a few useful things to do here:

1. Give the PR a meaningful title.
2. Give it a meaningful description so collaborators know what change in the project you're aiming for. You can also link to issues here, for example by saying "This PR closes #1" (where #1 is the issue reference).
3. Assign reviewers - add the other members of your group here!

Once you've filled in those elements, you can click **Create pull request**. Note that this won't merge the branch into main, this just creates a review area where you can access useful info on the merge and get feedback from collaborators.

Reviewing a pull request

There are some basic steps to go through if you're asked to review a pull request. The key elements are:

1. Review any automated checks or QA scripts;
2. Clone the repository, switch to the relevant branch and run the code;
3. Look through and comment on the changes to the repository using the **Files changes** panel in the PR.

In this case, we don't have any automated checks set up properly, so we'll focus on points 2 and 3.

First of all try switching to the *featTimeSeriesChart* branch if you're not already in it and run the Shiny app - this can be done by opening the global.R script in RStudio and clicking Run App in the top right hand corner of the viewer pane. Assuming the dashboard runs, then try cycling through the different panels of the dashboard looking for any problems, errors or just things that could be improved.

There should be plenty of issues to find as we've kept the actual dashboard coding brief to focus on using git. As you find them, enter them in to the GitHub pull request, either under **Review changes** on the Files changed panel or as comments in the **Conversation** panel.

In reality, once you've got those reviews collated, you'd go through and make changes to the code accordingly. This provides the checks and balances and a structure for code QA necessary when developing reproducible analyticle pipelines or data dashboards.

Assuming we've dealt with the outcomes of those reviews appropriately, the next step is to complete the Pull request by clicking the **Merge pull request** button. This then completes the merge in to *main* in this case. Whilst the basic mechanics of what's happening with the branches is the same here as with just running `git merge`, the Pull request provides that extra layer of administrative structure to perform proper QA of the code and the resulting product.

Merge conflicts

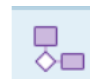
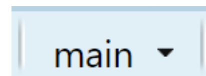
When `git` performs a merge, it uses the histories of different changes that were made to files to help understand what changes in any given file supersede other across a pair of branches. In some cases, for example of two branches contain contradictory changes to the same part of a file, then `git` won't automatically merge the two versions of that file and will instead call for the user to give it some steer on how it should procede. This is called a **merge conflict**.

Summary

We've looked through a lot of the basics in this section, covering adding/staging, committing, pushing/pulling between remote and local repos, merging and pull requests. These are all the main concepts you need to use git.

We've also tried to cover doing all this through a mixture of RStudio, git BASH and GitHub (and as we've said Azure Dev Ops offers similar functionality to GitHub). Most common processes can be done multiple ways and there's not necessarily a single right method to follow, just whichever makes most sense in your situation.

Just a quick final note on why it's useful to be familiar with git `BASH`. Whilst most of the basic git functionality can be accessed via the RStudio panel or GitHub/Dev Ops, there are some things that are best achieved through BASH. In particular, if you have a file in your repo that you need to remove entirely, this pretty much requires someone to use commands via git BASH.

Process	git BASH	RStudio git panel
Create branch	`git checkout -b branch_name`	
Switch branch	`git checkout branch_name`	
Merge branch	`git merge branch_name`	N/A - use GitHub/Dev Ops

Troubleshooting

renv

If `renv::restore()` causes issues, then one of your team should try `renv::init()` and select option 2 to restart renv. Then do a add/commit/push cycle and get the other team members to do a pull and then try running `renv::restore()` again on their local clones of the repo.

Datafiles commit-hooks/`.gitignore`

To help teams keep on top of avoiding any accidental publishing of unpublished data, we've added in some code around commits that checks through any data files in the repo and checks them against a logfile and the .gitignore file. Any files listed in .gitignore will not be included in commits and therefore won't be sent to the remote repo as part of any push.

merge conflicts

Merge commits happen when two branches have conflicting changes that have been made concurrently. `git` can usually figure out how to prioritise changes based on the commit history, but if changes have happened at the same time to the same bit of code across different branches, then it will need to get your input on how to prioritise the changes.

The easiest way to go through how to deal with merge conflicts is by discussing with an example, so ask us in the workshop if and when you hit a merge conflict.

Briefly though, when there's a merge conflict, git will add some text to the file containing the conflict along the following lines:

```
<<<<<<<<<<< branch_1
code
on
branch
1
===========
conflicting code on branch 2
>>>>>>>>>>> branch_2
```

Effectively as the user, you need to decide which bit of code is the right bit to keep and then delete anything you don't want to keep as well as the tag-lines that git has added in. So for example, you should be left with something along the lines of:

```
code
on
branch
1
```

Once you've cleared up all merge conflicts in the branch that you're working on, then perform another add/commit cycle and thay should clear out the conflict from the branch that you're working on and you'll be able to continue with the intended merge/PR.

About this publication:

enquiries	www.education.gov.uk/contactus
download	www.gov.uk/government/publications

Follow us on Twitter: @educationgovuk

Like us on Facebook: facebook.com/educationgovuk