



Department
for Education

DfE Statistics Development Team Workshops

Coding RAP using R

Contents

Introduction	3
What is a RAP?	3
What is R/R Studio?	3
Pre-workshop requirements	5
Technical requirements	5
Working in teams	7
Getting started ...	8
Creating a collaborative project	8
Setting up a version-controlled repository	8
Your initial script	9
Summary	9
Troubleshooting	11
renv	11
Datafiles commit-hooks/.gitignore	11
merge conflicts	11

Introduction

We've prepared this walkthrough guide for statistics publication teams as an introduction to the ways in which coding in R can be used for Reproducible Analytical Pipelines (RAPs), creating functions for typical tasks that teams may come across. 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 with access to experienced R users 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 R, RAP and coding in general.

What is a RAP?

RAP stands for Reproducible Analytical Pipeline. The full words still hide the true meaning behind buzzwords and jargon though. What it actually means is using automation to our advantage when analysing data, and this is as simple as writing code such as an R script that we can click a button to execute and do the job for us.

Using R (the coding language) really helps us to put the R in RAP ('reproducible'). Ask yourself, if someone else picked up your work, could they easily reproduce your exact outputs? And when the time comes around to update your analysis with new data, how easy is it for you to reproduce the analysis you need? In an ideal RAP, it would be as simple as plugging the new data in and clicking 'go', with no need to manually scroll through multiple scripts updating the year in every file name or the variable name that's changed from using `_` to `-`.

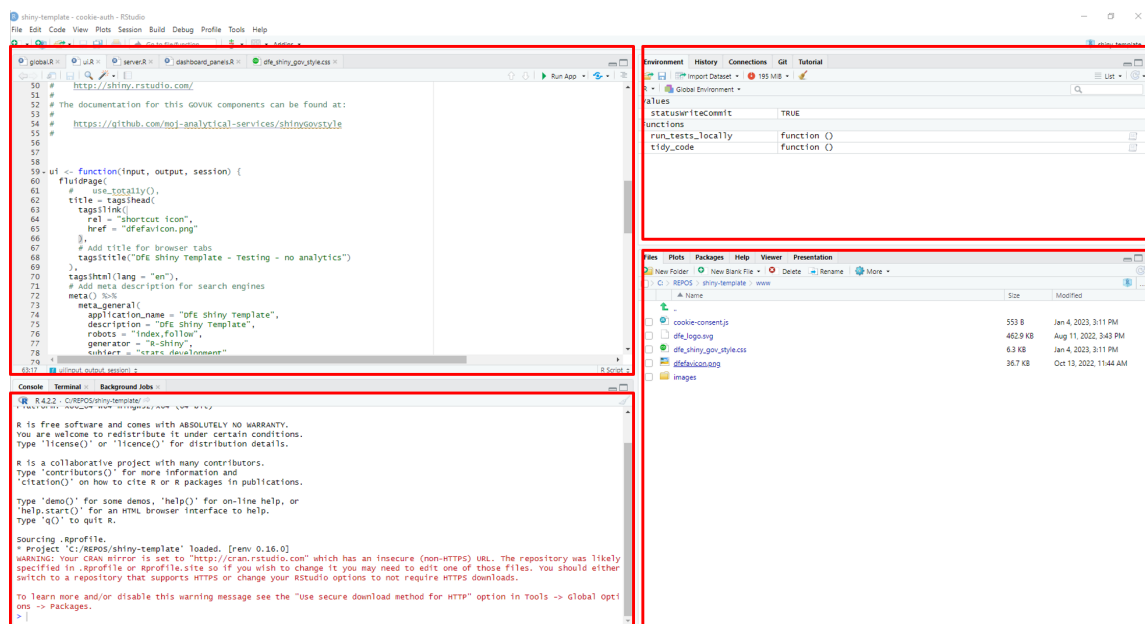
What is R/R Studio?

R is a coding language. There are many different languages of code, some others include SQL, python, JavaScript and many more. They all have benefits, and the difference is often the syntax used (literally like learning new languages!). R is open-source, meaning it is free, anyone can use it and anyone can contribute to developing new 'packages'.

- A *package* is a set of functions someone else has written and tied together in a nice neat bow, ready for you to use! You simply install the package, and then you have all of the functions available.
- A *function* is a chunk of code that has been grouped together, given a name, and often has 'place holders' you can change, such that you can use that name to run that code, and apply it to different data. (for example, `mean(x)` is a function that calculates the arithmetic mean of `x`. You replace `x` with any numeric data.)

R studio is a programme that enables us to code in the R coding language, while also providing a visual and interactive interface. It has useful areas and windows that enable you to see what tables you have loaded, charts you have created, Git user interface, file explorer, help windows and many more! Normally you will see the screen split into 3 or 4 windows:

Figure 1: R studio panels



- Top left - Source pane. Open and view your code scripts, tables, data sets, functions.
- top right - Environment/History/Connections/Git (if you have it). The environment shows you what data, functions, objects etc you have.
- Bottom left - Console. Shows what you have run, and you can type and run commands directly into the console.

- Bottom right - File explorer/plots/packages/Viewer.

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.

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).

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.

Figure 2: Enter the path to your git executable in the git path option box

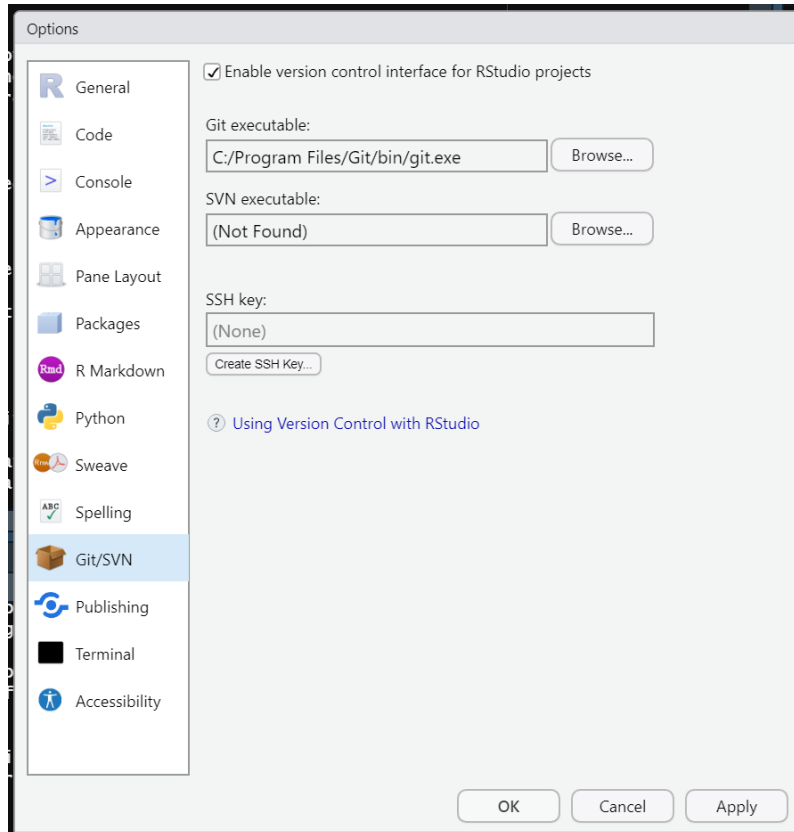
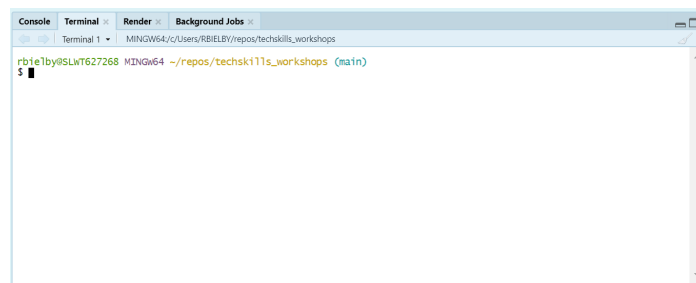


Figure 3: The 'git BASH' terminal in R-Studio



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.

To help illustrate the challenges and benefits of git and Dev Ops and how to work collectively within the same space, all the groups will be working within the same repository. Each group will have their own working branch (already created) with the naming format `workshop_group_N`:

- `workshop_group_1`
- `workshop_group_2`
- ...

We'll need to prefix branches and tasks within the repository with `grN_` and group N respectively (again switching the N for your group number).

By the end, you should get a good idea of how to utilize R for RAP processes, as well as how to collaborate using git!

Getting started ...

Now we will begin the tasks and group work. First, everyone open up R studio.

Creating a collaborative project

When creating RAP processes, it's key that you use code that is reusable, transparent, and well documented, such that it is future-proofed for any future team members. It is also part of the department's baseline expectation that code scripts are version controlled. Therefore, first, we will show you how to create an R project that is linked to an online repository using Git. This means that all of the work is saved in an online, cloud-type environment that your team members can all access, which is great for QA, peer-review, and collaboration.

Using version controlled scripts also removes the need to copy the same code scripts into a new folder every single year. When all of your code is saved in a Git repo, there is no need to recreate the same scripts every year, you can simply update the single code script you need each year since the version-control software automatically stores the history and tracks any changes.

Setting up a version-controlled repository

You are now going to work through a series of tasks. We are starting at the very beginning - creating a version-controlled R project, which is linked to a repository. There are two places we create the repository, *Azure DevOps* or *GitHub*.

Azure DevOps is secure, and access is controlled within the department. This is where many teams save their RAP processes if they aren't suitable to be visible to the public.

GitHub is open to the public, and so you should only use GitHub for things that it's appropriate for the public to see. Transparency and reproducibility are important for RAP, and open-sourcing code to the public contributes to these. The code behind all of the department's public-facing dashboards is currently in the [dfe-analytical-services GitHub area](#).

Today we will use

First, open R studio and select File > New project. . .

Your initial script

Your first script should definitely include the library-calls for all of the packages you will need. You should also activate 'renv', this stands for R environment, and is a package that tracks all of the packages you use/need within a project as well as the version used, and stores the information in a `renv.lock` file. If someone new comes to the project, they will only need to run `renv::restore()` to immediately install all of the packages that the project requires. Once `renv` is activated, if you add more or delete packages, you simply need to run `renv::snapshot()` to save the state of your library to the `renv.lock` file.

Our first steps only require two packages, 'readr' and 'dplyr'. Install these by typing `install.packages('readr')` and `install.packages('dplyr')` in the console. Then, remember to take a `renv::snapshot()`!

Now we are going to create the first script of the project, in which we will call all of the packages, files and functions we need for the workshop!


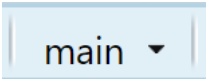
Open a new R script file by selecting *File > New File > R script*. Start by adding some useful comments to the first few lines that you think would be useful for someone

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	<code>git checkout -b branch_name</code>	
Switch branch	<code>git checkout branch_name</code>	
Merge branch	<code>git merge branch_name</code>	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 that 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.



Department for Education

© Crown copyright 2022

This publication (not including logos) is licensed under the terms of the Open Government Licence v3.0 except where otherwise stated. Where we have identified any third party copyright information you will need to obtain permission from the copyright holders concerned.

To view this licence:

visit www.nationalarchives.gov.uk/doc/open-government-licence/version/3

email psi@nationalarchives.gsi.gov.uk

write to Information Policy Team, The National Archives, Kew, London, TW9 4DU

About this publication:

enquiries www.education.gov.uk/contactus

download www.gov.uk/government/publications



Follow us on Twitter:
[@educationgovuk](https://twitter.com/educationgovuk)



Like us on Facebook: [face-
book.com/educationgovuk](https://facebook.com/educationgovuk)