**Introduction to Git/GitHub**

**Basic setup**

This document provides accompanying training material used in the Introduction to Git/GitHub Training session, conducted by the Data & Analysis R training group. Prior to joining the session, you should ensure you are set up on the Analytical Platform - see <https://user-guidance.services.alpha.mojanalytics.xyz/get-started.html>.

We expect participants to have completed the Introduction to using R on the Analytical Platform (or equivalent standard) and Introduction to R (or equivalent standard). Previous recordings and accompanying materials of sessions can be found here - <https://moj-analytical-services.github.io/ap-tools-training/>.

You should then work through the following:

1. Deploy (if necessary) and open RStudio: <https://user-guidance.services.alpha.mojanalytics.xyz/tools/control-panel.html#control-panel>
2. Connect RStudio to GitHub: <https://user-guidance.services.alpha.mojanalytics.xyz/github.html#setup-github-keys-to-access-it-from-r-studio-and-jupyter>
3. Clone the GitHub repository for this course (<https://github.com/moj-analytical-services/intro_to_github_training>) by following step 1 here: <https://user-guidance.services.alpha.mojanalytics.xyz/github.html#r-studio>
4. In the Console window in RStudio, enter this command to make sure you have the required packages installed: renv::restore()

**Contents**

This session will provide you with an understanding of what Git/GitHub is and explain the key benefits of using it alongside the Analytical Platform and RStudio. Step-by-step instructions will be provided about how to use GitHub in RStudio and how to collaborate effectively with others. Even if you are coding a one-off product, we highly recommend you use GitHub.

**Sessions aims**

1. What Git and GitHub are?
2. Understanding how to use GitHub
3. Ways to approach Git to enable effective use and collaboration
4. Security considerations when using GitHub
5. Further information and useful links

As this session is focused on those who are new to Git/Github, it will not cover more advanced commands and topics. We welcome comments and volunteers to develop a session in the future.

* 1. **What are Git and GitHub?**

First, Git and GitHub are two separate things.

Git is a version control system. Version control refers to the process of saving different files or ‘versions’ throughout the various stages of a project. It’s a bit like “Microsoft Office’s track changes on steroids”. This enables users to keep track of changes to code. All code written on the Analytical Platform should be stored in a Git repository on GitHub, including Python scripts and Jupyter notebooks. In this session, we will show how to use Git in RStudio.

GitHub makes it easier to collaborate using git. It’s a platform and website that provides a home for your coding projects allowing multiple people to work on a single project and see edits to the code in real-time. The code is held in a repository on the GitHub website – a repository is a file location where your project is stored. This means your code can be made publicly available in GitHub repositories on the GitHub website so others can interact with and contribute to the code. GitHub is therefore, an open-source platform which is free to use.

To gain access to the Analytical Platform, you would have needed to use GitHub to create a GitHub account. The Analytical Platform team use this authentication service of GitHub to enable users to have one account for the Analytical Platform, RStudio and GitHub. If you have not set up an Analytical Platform or GitHub account, then please refer to the guidance here [Get Started - Analytical Platform User Guide (justice.gov.uk)](https://user-guidance.analytical-platform.service.justice.gov.uk/get-started.html) and watch the Introduction to using R on the Analytical Platform training session found here <https://moj-analytical-services.github.io/ap-tools-training/>.

* 1. **Advantages of using Git and GitHub**

As your projects develop, new coding ideas thought up, or new improvements are created, you will need to keep track of what changes are being made, track future developments and be able to go back to previous versions if all else fails. Git and GitHub keep track of evolving projects in a sensible, precise way, with the ability to view, compare in detail and restore previous versions. It is also collaborative, allowing multiple people to work on a project at one time with a managed way of combining several people’s work. This makes it easier to resolve errors and fix other mistakes that might occur through development of code with the ability to restore previous versions.

**1.2.1 Advantages of Git**

As Git is different to GitHub, it is important to understand their differences and benefits. Git has many great advantages especially when it comes to performance. From easily creating new code that can be stored and saved, to managing and creating new versions of your project and quality checking your code so it has been optimised to perform better. Some of the advantages are as follows:

* **Security:** Git as the ability to keep your version or different versions secure, saved and easily accessible
* **Flexibility:** Git’s flexibility allows the user to have multiple different versions of your code where changes made in different versions do not affect each other.
* **Version Control:** Having different versions of your code will allow for greater quality checks and each different version can be checked individually.

**1.2.2 Advantages of GitHub**

GitHub is your user-friendly interface of Git that brings everything to the forefront, in an easy to understand and comprehendible way. This allows someone to look at the project and understand why the changes were made earlier on. You can note changes in each version to help team members stay up to date on what’s completed and what needs to be accomplished. More benefits include:

* **Accessibility:** You can access the code files from another computer and so can other developers.
* **Documentation:** ReadME.md file one of the first things other users see within your repository/project. The README.md file is used for documentation purposes – providing other users with an overview of the project so they run the code in RStudio.
* **Collaboration:** GitHub provides you with tools which allow you or others to review each other's work, combining your code with others or making changes/updates to your own. It can do all this while keeping a track of the version history which can be changed back to old versions if required.
* **Project management:** Coordinate and track your project in one place. This is like a Trello Board but allows you to include issues you have found in the code.
* **GitHub issues:** This is a central place to maintain a ‘to do’ list for a project, and to discuss them with your team. ‘Issues’ can be bug fixes (such as ‘fix divide by zero errors in output tables’), or features (e.g. ‘add a percentage change column to output table’), or anything else you want.
* **Version control management:** By using issues and projects you can keep track of who is working on what. If you use issues, you automatically preserve a record of why changes were made to code. So you can see when a line of code was last changed, and which issue it related to, and who wrote it.

Git and GitHub have a multitude of benefits that help individuals manage their projects, and if used correctly and efficiently GitHub will give you additional security in your projects allowing you to track, manage and maintain the standard of your work. With this additional quality assurance measure you can be certain that your code and methods are exactly what you expect!

**1.3 Ways to use Git in RStudio**

To take advantage of GitHub, we need to know how to use Git to interact between RStudio on the Analytical Platform and GitHub.

There are two ways to use Git in RStudio:

* **The terminal** – this is the command line interface. This allows users to write commands that associate directly with the terminal window. Users must be aware of the commands to perform any task they want and understand the responses from the terminal.
* **The graphical user interface (GUI)** – this lets users use graphics for interacting with Git such as buttons, windows etc. It’s easier to use but some people find it hard to know what action Git is performing.

Some people prefer the terminal, some prefer the GUI and some people use a mixture of the two. The more you continue to use Git and GitHub, the more comfortable you’ll get using the commands and/or interface. The key is to start small and maintain your momentum. It will eventually get easier as you build small projects and host them on GitHub using Git.

Later in this session, we will provide a step-by-step guide on how to use the terminal and the graphical user interface (GUI).

* 1. **Understanding how to use GitHub**

We will walk through the basic steps on how to use Git and GitHub for you to take away and practice with your own projects. We will do this in four main sections:

1. Starting a repository and creating a branch
2. Editing files in your repository and committing your changes
3. Pushing your changes to Github
4. Pulling your changes to RStudio

Each section will first explain theoretically the action we are performing using visual aids and then demonstrate how to use this in the GUI and the terminal – providing you with both options. Please only use the terminal or the GUI once in each step if following along in your R studio.

**2.1 Starting a repository and creating a branch**

* + 1. **Create/clone a repository**

A repository is a place to store your code and record changes to your project.

On the homepage of the MoJ Analytical Services organisation on GitHub (<https://github.com/moj-analytical-services>) you can see all the repositories related to work in MoJ Data and Analysis that you have access to. A repository (known as a ‘repo’) is an area on GitHub where you can store code and documentation for a particular project. Each project in RStudio should have a separate repo associated with it.

Start by creating a new repository for your project on GitHub (see the [Analytical Platform guidance](https://user-guidance.analytical-platform.service.justice.gov.uk/github/create-project.html#create-a-new-project-in-github)) or find the repository you want to work on if it exists already. You then need to create a local copy of the repository that you can work on and change (see the [Analytical Platform guidance](https://user-guidance.analytical-platform.service.justice.gov.uk/github/rstudio-git.html#work-with-git-in-rstudio) and [Intro to us R on the AP](https://moj-analytical-services.github.io/ap-tools-training/)). Creating a local copy of a repository is called cloning the repository.

In this session, we will use the repository created for this training - <https://github.com/moj-analytical-services/intro_to_github_training>. You should have cloned this repository into RStudio before this session and have it ready in your projects. If not, then please follow the steps <https://user-guidance.services.alpha.mojanalytics.xyz/github.html#r-studio>. It is not essential to follow along during this course, but it is useful to practice the steps whilst watching the presenters.

* + 1. **Create a branch**

Once you have created a repo or have cloned a repo into your RStudio, then you will want to add or make changes to your R project. You can make changes to your ‘main’ branch on the repository, or you can create a new branch to make changes to the project separately. Branches allow multiple people or multiple components to be developed at the same time. We will explain ways of working with branches in section 3.

GUI

1. Create a branch: click the “Git” tab on the right of R-Studio, then the small purple shapes on the right of the screen, add a branch name, and click create. (Note that if the “Sync branch with remote” checkbox is ticked the branch will also be created on the remote GitHub repo, removing the need to do that manually at a later stage.)
2. A pop-up will appear with a message that you have switched to a new branch, and the branch name will update in the Git pane. You can just close the pop-up.

Terminal

* Use the git status command before you create a new branch. git status displays the current state of your files and the staging area: what branch you’re on and which files have been modified or staged since the last commit.
* To create a new branch:

git checkout –b my\_branch

* To switch to another existing branch, for example to switch to main:

git checkout main

* 1. **Editing files in your repository and committing your changes**
     1. **Edit some files, for example write some code in an R script**

Next, make some changes to a file in your repository. For example, you could create an R script and write some code in it. In this training session, please write something in the R script and then push the changes so you can see it on GitHub. Normally, you could push any change to GitHub – comments, code, documents etc.

If using the GUI, any files that have been edited and saved will appear in the git tab on the right-hand side of the screen. You can view the changes by pressing the Diff button and selecting the file name. Lines highlighted in red have been removed; lines highlighted in green have been added. Lines written in grey have not been changed. The left margin shows the line numbers of the code that has changed.

In the terminal, the git diff command can be used to show changes. Typing the command without any options shows the uncommitted changes since the last commit.

* + 1. **Stage and commit a group of changes**

Once you are happy with the changes to your files, you can save a snapshot containing the state of all your files by creating a “commit”. A commit can contain changes to one file or multiple files. The ability to create commits, each containing a different state of your repository, and to switch between the commits, is what makes Git a version control system.

Before you can commit changes, you must add them to the "staging area", which tells Git to include the changes in your next commit.

The steps for committing changes are:

* + - 1. Save the changes to your files
      2. Add the modified files to the staging area.
      3. Commit the staged files.

Every commit needs an accompanying commit message in which you should briefly describe the changes made to the files. The commit message will appear alongside the commit in the repository history, helping you and others to see what changes were made.

Terminal

Use git add to add files to the staging area.

* To stage a single file:

git add my\_file.txt

* To stage all modified files:

git add .

Use git commit to commit the staged files.

* To commit the staged files and open a text editor in which to type the commit message:

git commit

* To commit your changes and type an inline commit message (instead of opening the text editor):

git commit -m "My commit message"

GUI

Click either Diff or Commit to open the Review changes pop-up.

1. To stage a file, click the check box by the file name. Stage every file you would like to include in the commit.
2. Add a commit message in the commit message box.
3. Click the Commit button.

When using the GUI changes to files can be seen in the status column in the pop up from Diff/Commit. Here are some of the most common.

* ? – An untracked file – a file that has not yet been staged or committed.
* A – A file that has been added to the staging area.
* M – A file that has been modified.
* D – A file that has been deleted. Staging and committing this file will delete it in the committed snapshot.
* R – A file that has been renamed.
  1. **Pushing your changes to GitHub**
     1. **Push changes**

Once you have committed your changes locally, you need to upload your commits to the remote GitHub repo. This is called pushing your commits. You push commits from a local branch to a remote copy of the branch.

Terminal

Use git push to push your branch to the remote repo.

* To push to a branch that already exists on the remote repo:

git push

* If your branch does not yet exist on the remote GitHub repo, the first time you push you will need to create a remote version:

git push –u origin my\_branch

The option -u is a shortcut for --set-upstream, which creates a version of the branch on the remote repo. Note this will not be necessary if you created the branch using the RStudio GUI and the “Sync branch with remote” checkbox was ticked.

GUI

1. After committing, you will see a pop-up listing any changes, additions, deletions and the commit message.
2. Back on the main RStudio screen, you can see how commits ahead of the remote GitHub repo you are.
3. To push your changes to GitHub, click the up arrow.
4. You will see another pop-up confirming your branch has been updated on GitHub.

**2.3.2 Merging and pull requests**

When you’ve finished making all the changes you want to include in your branch, you will probably want to merge your branch into the main branch of the repository, which will incorporate your commits into that branch. There are two ways to merge a branch, either locally using the git merge command, or on GitHub via a “pull request”.

A pull request is a GitHub feature that handles the merge process on the remote repo. It allows collaborators to discuss and review the proposed changes before you merge them. This is particularly useful for both quality assurance and project management. The discussion thread can be used to explain the proposed changes and link to a project plan or issues log. A branch can be protected so that pull requests can only be merged once reviewed and approved by a colleague.

Note that once you have opened a pull request, you can continue to commit to the branch. When you push additional commits, the pull request page will update to show the latest changes.

To merge a branch locally using the terminal:

* + 1. Switch to the destination branch with git checkout, for example if you want to merge a branch into the main branch:

git checkout main

* + 1. Ensure the destination branch is up to date with the remote repo with:

git pull

* + 1. Use git merge to merge in the branch containing changes:

git merge my\_branch

To merge on GitHub via a pull request:

If you recently pushed a change to GitHub and navigate to the online repo, GitHub notices the changes to the branch and asks whether you want to open a pull request. Otherwise, to open a pull request on GitHub:

* + 1. Go to the "Pull request" tab.
    2. Click "New pull request".
    3. Use the drop-downs to select the base branch and comparison branch.
    4. Click "Create pull request"
    5. **Confirm the pull request**

To merge a pull request, select "Merge pull request" at the bottom of the conversation thread. If the repo has been set up to require an approving review before you can merge, the option will not be available until someone has approved.

* 1. **Pull changes into a local branch**

If there have been changes made to a branch on the remote repo, you will need to update your local branch to incorporate the changes. For example, if a pull request has been merged into the main branch on GitHub, you will probably want to update your local version of the main branch. This is done using the git pull command.

Terminal

Use the git pull command to merge commits added to the remote branch into your local branch.

For example, if the remote main branch has been updated:

* + 1. If you are not currently on the main branch, switch to it with:

git checkout main

* + 1. To download the remote version and update your local branch:

git pull

GUI

1. After the merge has been accepted on GitHub, use the down arrow (next to the push arrow) to pull in the changes on GitHub.
2. A pop up will appear showing the changes.

In the GUI example screen shots, StewBranch2 was merged with Main branch. The local copy of Main Branch in R-Studio was updated by pulling the changes to Main Branch from GitHub.

* 1. **View the commit history**

Once a commit has been made, it is retained in the project’s history, which means it can be viewed and returned to at any later point. It is often useful to see the history of the most recent commits on a branch, which you can do using RStudio or the git log command.

GUI

In the Rstudio Git pane, click on “History” to see the commit history for the current branch.

Terminal

The git log command shows the commit history for the current branch, starting with the most recent. The --oneline option is useful to condense the output. You can limit the number of commits shown with the – option.

For example, to see the most recent five commits on the current branch:

git log --oneline -5

* + 1. **Ways to approach Git to enable effective use and collaboration**

There is no limit to the number of people who can potentially be working on a repo at the same time. However, there are technical limits to Git’s ability to integrate people’s work effectively, and established processes and practices for dealing with these limits.

**3.1 Merges and merge conflicts**

What happens when a pull request is created, and git tries to merge depends on the nature of the differences (diffs) between the branches. As much as it can, Git will do try to work out what to do with the diffs and merge branches automatically, but it cannot always do this. Sometimes there can be conflicts which you need to manually resolve.

You don’t need to understand exactly what Git is doing under the hood in these scenarios, but if you know what Git can and can’t automatically handle you can:

* figure out the best way to work with your team
* be equipped to deal with conflicts when they do arise (and they will arise)

Fast forward merge

This is when the descendant branch is being merged back into the ancestor and there have only been commits in the descendent since divergence. This is easy - effectively it is just an update.

e.g. Anna branches off from main – her branch is called a ‘feature’ branch - does some commits, submits a pull request to merge her branch into main, which her colleague Bruce reviews.

Three-way merge

This is when the descendant and ancestor branches being merged have both had commits made in them since divergence.

e.g., if Bruce also branched off to do some commits at the same time as Anna did, and then wants to merge back in after Anna’s work has been merged back in with her fast-forward merge.

Git is good at this and will do it automatically – its facility at this is one of the key reasons people use it.

However, for git to do its job, diffs must be limited to:

* file additions
* changes to different files
* changes in non-adjacent lines in the same file

Other diffs (deletions in one of the merging branches, changes on the same or adjacent lines) cannot be automatically merged and trigger a merge conflict (see below)

It is called a three-way merge because in addition to the two merging branches, Git needs to use their common ancestor (their shared state at the point of divergence) as the third branch - *this tells Git what the baseline state was so that it can adopt all changes from that state.*

Merge conflicts

When the descendant and ancestor branch have both had commits made in them since divergence and there are diffs that git cannot resolve automatically, we have a merge conflict. Problem diffs that trigger a merge conflict are:

* File changes to the same or adjacent lines
* File deletion in one branch

When this happens, git gives up and it’s over to you to fix it.

**3.2 Effective collaboration**

Getting used to working together in this way, adding to and drawing from a shared synchronised resource (the ‘origin’ on GitHub), can be difficult. There is a number of different practices that your project can adopt in order to make the best use of Github.

**Gitflow:** There are different conventions for workflows (branching patterns) that teams may adopt when using git. Analysts within MoJ are generally recommended to use gitflow *(not to be confused with the similarly named Github flow).* The key feature of gitflow is the use of an additional branch, usually called dev, as an insulator between main and feature development branches Section 5 has more details on Gitflow.

**Work on separate project features:** As a general rule, collaboration on a project works best if people try to work not only on separate files, but on separate project features, so as to minimise prospect of changes to overlapping code. Combining this approach with gitflow is a pretty solid formula.

**Project management:** Teams can manage work allocation and PR review assignment using project management tools such as Jira or Gihub’s project board, or even just using Github’s issues features. Agree your project’s broad workflow/strategy. Communicate with one another (e.g. using Teams/slack).

**Small and regular commits:** Ideally with each commit describing a discrete change, group of associated changes, involving few files. *It should be possible with RStudio integration to stage individual lines as well as chunks, although some users on the AP report problems with this.*

**Small branches and regular merges:** This will reduce the risk of merge conflicts. To make this work effectively, communicate regularly with your team to ensure that PRs can be swiftly reviewed *(note that this is not itself a tactic associated with gitflow, which is designed to be robust to long-lived branches)*

**Unit tests:** Formal comprehensive systematic testing of code across the repo before committing (and review) will reduce the chances that your changes will break part of the project.

**Protect your branches:** e.g. Make dev the default, require review of PRs before merge, prevent non-admin from merging dev -> main

**Delete merged branches:** Your commit history will remain if it is part of the tree of work. Note that closed PRs for abandoned branches will remain and retain commit history

**Document your repo:** If you write a README file, github will show this in rich markdown at the front of the repo page. There is also a wiki feature.

**3.3 ‘undo’ commands**

There is a number of different Git commands that can be used to effectively undo your work, after you’ve committed. Let’s start with changing saved objects in the working directory.

**git restore**

**git restore** is used to restore a specific file to a previous state.

You must specify a filepath, e.g.

**git restore path/to/file**

By default it will restore the file to the state that it was in at the last commit. In RStudio with git integration you can also do this with the revert button (not to be confused with git revert). But you also have the option of specifying a different branch, if you so wish, e.g.

**git restore --source=branchname path/to/file**

**git revert**

git revert is a useful command that works at the commit level. It creates a new commit that reverts a previous one.

So if you run

**git revert HEAD**

I will create a new commit that reverts the last one. You will get prompted to enter a method. There is a default so you can just press caret -X

You can actually revert to a state that undoes any previous commit by providing the commit hash. You can find these a number of ways. You can find them with **git reflog** (show), or you can find them on github. You simply append it to your git reset command, e.g.

**git revert 7b668c1**

This creates a commit that undoes the diffs in that commit.

The great thing about **git revert** is that the commit history is preserved, which is generally considered good for collaboration.

If you’re wanting to undo some work someone else has done that you’ve pulled, you should be using

**git revert**

It is also arguably conducive to transparency to use git revert within your own code development to keep a record of what you’ve done

**git reset**

Finally, we make you aware of another option: **git reset.** This allows you to move back to a previous commit (e.g. git reset HEAD~1 will take you back one). Although it can make your commit history cleaner to look through, it is considered more risky as commits that have been reset over will ultimately be ‘orphaned’, which ultimately causes discarded work to become completely irretrievable. It’s therefore less conducive to transparency and accountability and recovering work that you might have changed your mind about discarding. In particular, it should not generally be used to undo others’ work. However, it can be a useful tool in some contexts, such as when working locally.

* + 1. **Security considerations when using GitHub**

GitHub is a website with public areas accessible to all internet users. Within GitHub we have a private area called moj-analytical-services. Despite this, it is important to think about the sensitivity of the information you push to Github. You should not add data to GitHub that isn’t already publicly available - this would be a breach. See here for more guidance on security and what to do if you accidentally pushed sensitive data <https://user-guidance.analytical-platform.service.justice.gov.uk/github/security-in-github.html#security-in-github>.

Amazon S3 buckets should be used to store data. It is a web-based cloud stage platform and all data and final analytical outputs in Data and Analysis is stored in an Amazon S3 bucket. To learn more about Amazon s3 buckets, please refer to section 5 of the Introduction Using R on the AP training session. Previous recordings and accompanying materials of sessions can be found here - <https://moj-analytical-services.github.io/ap-tools-training/>.

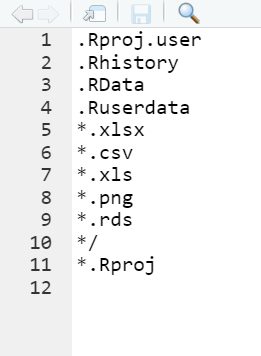
**4.1 Git ignore files**

You may also not want to push some files/information/images that are required for your project onto GitHub due to sensitive information. This could also include files such as the renv folder, which is used for package management.

To help ensure we don’t push something by accident to GitHub, we use Git ignore files. You should use the Git ignore file in your repository to tell Git which files and directories to ignore when you make a commit – allowing you to control which file types are shared on the GitHub website. Once the Git ignore file is on Github, then others can pull this from the project into RStudio to ensure they also follow the rules of pushing information to GitHub.

Git ignore files have a list of recommended files to ignore – you can edit and save this to ensure it is relevant to your project. For example, if I saved some data for a report in a .xlsx file in the same directory as my R project, GitHub would attempt to upload this to the GitHub website with any code amendments. This could potentially lead to a breach as sensitive data has been shared online.

To avoid this, we can use a .gitignore file – as seen below. We can add file types with a .gitignore file using simple syntax.



This .gitignore file allows .Rproj.user, .Rhistory, .RData, .Ruserdata to be uploaded to GitHub when the project is pushed. It doesn’t allow .xlsx, .csv, .cls, .png, .rds, .Rproj . An \* in front of a file type will tell GitHub to ignore all file types of that kind when syncing with the online repository.

Users should also avoid putting outputs of R markdown reports on GitHub as this could contain official sensitive information and results of analysis. Adding \*.docx to the .gitignore file would stop any documents being added to GitHub.

We would also recommend adding the core file to the .gitignore file. This is generated when using RStudio crashes when trying to work with large datasets. These should not be committed to gitignore and should be deleted when they appear in the files section of RStudio. More information can be found here [Security in GitHub - Analytical Platform User Guide (justice.gov.uk)](https://user-guidance.analytical-platform.service.justice.gov.uk/github/security-in-github.html#protecting-information-in-github).

**5. Further information on Git and GitHub**

**5.1. Advanced topics**

We have added some information and links below on some other aspects of GitHub that users could explore that haven’t been covered in this session. These are more advanced features of GitHub and are not required to be able to use Git and GitHub as a beginner.

* Git releases/versions - useful for package development and those working on production/publication code - [About releases - GitHub Docs](https://docs.github.com/en/repositories/releasing-projects-on-github/about-releases).
* Git stash – when you want to store changes to code when they are not ready to be committed and revert to a previous commit.
* Git detached head - [Git Detached Head: What Is It & How to Recover (cloudbees.com)](https://www.cloudbees.com/blog/git-detached-head)
* Packages on Github - Github allows you to host your software packages privately or publicly and use them within your projects.
  + If you’re interested in creating packages, then listen to the session on Developing R packages and RAP ways of working - <https://moj-analytical-services.github.io/ap-tools-training/ITG.html#ITG>.
  + There also some packages created by analysts with MoJ that are hosted within moj-analytical-services on GitHub – for example, mojrap which includes functions that have been created in various Reproducible Analytical Pipelines <https://github.com/moj-analytical-services/mojrap>
  + [Install packages from GitHub - Analytical Platform User Guide (justice.gov.uk)](https://user-guidance.analytical-platform.service.justice.gov.uk/github/install-packages.html#install-packages-from-github)

**Additional information on Gitflow**

Gitflow is one practice that we recommend you give serious consideration to adopting to any project that involves multiple contributors.

**The dev branch**

One feature that we recommend is using an additional branch as an insulator between the main branch and feature development branches, usually called dev.

We saw how git can handle changes to the same file without there being merge conflicts. But even if colleagues work on different files and avoid conflicts, it doesn’t mean that all merged code will function as they intended.

Following a fast-forward merge, the code in a merged branch (main) will work if the code in the merging branch (Anna) worked. But that isn’t necessarily the case if Bruce’s work then merges in. Anna and Bruce’s code changes may not work when combined. If so, their project has a problem that Git doesn’t know about. It would only emerge when they try to run the code in the new merged main branch.

Using a dev branch provides an easy way to mitigate. It sits between feature branches and the main branch. Like main, we don’t make changes directly in dev. Anna and Bruce branch from and merge back into dev, instead of main.

With that in place, it's easy to manage things. They test the code in dev. If dev works, then after they’ve done their work it can be fast-forward merged to main. But if the combination of Anna and Bruce’s changes has broken the code, they can make another branch to fix it, and then do a fast-forward merge to dev before the fast-forward merge of dev into main.

The dev branch is not a perfect solution to be relied upon its own – *e.g. what if there’s a need for another branch off while the fix is happening? That person may have to work with broken code or may reintroduce the same problem back into dev later*

*(For future reference, other possibilities for dealing with this specific problem include automated integration tests with github actions, and an alternative to merging called rebasing, but these are technically challenging and/or complex and beyond the scope of this class).*

* 1. **Useful links**

Analytical Platform Guidance and slack channels

* Use this link to access the AP control panel: <https://controlpanel.services.analytical-platform.service.justice.gov.uk/tools/>.
* Lots more helpful guidance can be found in the MoJ Analytical Platform Guidance: <https://user-guidance.services.alpha.mojanalytics.xyz/>
* Specific Git section on AP guidance and how to set up slack channels: [Git and GitHub - Analytical Platform User Guide (justice.gov.uk)](https://user-guidance.analytical-platform.service.justice.gov.uk/github/index.html)
* Instructions for getting your GitHub account connected to your AP account: <https://user-guidance.services.alpha.mojanalytics.xyz/github/set-up-github.html#set-up-github>
* Gitflow in the MoJ RAP (Reproducible Analytical Pipeline) manual: <https://moj-analytical-services.github.io/rap-manual/git-flow.html>

R training

* Analytical platform and related tools training: <https://moj-analytical-services.github.io/ap-tools-training/index.html>
* Previous recordings and accompanying materials of sessions can be found here [3 Internal Training Group materials | Analytical Platform and related tools training (moj-analytical-services.github.io)](https://moj-analytical-services.github.io/ap-tools-training/ITG.html#ITG)
* The repo for this course <https://github.com/moj-analytical-services/intro_to_github_training>
* The GitHub repo for Introduction using R on the AP: <https://github.com/moj-analytical-services/intro_using_r_on_ap>
* The GitHub repo for the Introduction to R course: <https://github.com/moj-analytical-services/IntroRTraining>

Cheat sheets/further links

* <https://github.com/moj-analytical-services/intro_to_github_training/blob/main/github_cheatsheet_RStudio.pdf>
* [git-cheat-sheet-education (github.com)](https://education.github.com/git-cheat-sheet-education.pdf)
* [Let’s Git started | Happy Git and GitHub for the useR (happygitwithr.com)](https://happygitwithr.com/index.html)
* [Hello World - GitHub Docs](https://docs.github.com/en/get-started/quickstart/hello-world)
* [Git from the inside out (maryrosecook.com)](https://maryrosecook.com/blog/post/git-from-the-inside-out)
* [Git from the inside out - YouTube](https://www.youtube.com/watch?v=fCtZWGhQBvo)