**Introduction to Git and GitHub**

**Introduction**

This document contains training material for the Introduction to Git and GitHub training session run by the MoJ Analytical Platform tools training group. There are also PowerPoint slides used for part of the session, which you can find in the [course repo](https://github.com/moj-analytical-services/intro_to_github_training).

The session will explain what Git and GitHub are and discuss the key benefits of using them. We will give step-by-step instructions for how to use Git in RStudio and the command line, and how to use GitHub to store your work and collaborate with others.

The session is aimed at people who are new to Git and GitHub and will focus on the basic workflow for using them. We will mention a few more advanced features that it is helpful to be aware of but will not go into the detail of using them.

Note that we will demonstrate how to use Git and GitHub in RStudio, but most of what we will cover applies to others coding environments as well. Only the demonstration of the RStudio graphical interface in section 3 is specific to RStudio.

**Set-up steps**

Before the session, make sure you are set up on the Analytical Platform - see the [Quickstart guide](https://user-guidance.services.alpha.mojanalytics.xyz/get-started.html) in the user guidance. Also, make sure you have followed the steps to [set up RStudio](https://user-guidance.analytical-platform.service.justice.gov.uk/get-started.html#6-set-up-rstudio).

In RStudio, clone the repository for this course by selecting File > New project. In the New Project Wizard, select Version Control, then Git, and fill in the following:

* Repository URL: git@github.com:moj-analytical-services/intro\_to\_github\_training.git
* Project directory name: intro\_to\_github\_training
* Create project as subdirectory of: ~ (to use your home directory)

**Session outline**

1. Introduce Git and GitHub and explain the benefits of using them.
2. Explain the concepts underpinning Git and GitHub.
3. Demonstrate the basic Git and GitHub workflows using both the RStudio graphical interface and the command line.
4. Mention some more advanced features that you are likely to come across.
5. Discuss good practice for effective collaboration.
6. Mention some security considerations when using GitHub.
7. Signpost additional resources.

At the end of section 3 there is an exercise giving the opportunity to practice using the basic Git and GitHub workflow.

**Feedback**

Please leave feedback after you have attended a session or worked through the course material, using the [training review form](https://airtable.com/app54DtfpHprLGpKu/shr9u2OJB2pW8Y0Af).

# Background

## What are Git and GitHub?

Git is a version control system. Version control refers to the process of saving and retaining versions of a project’s files as changes are made. Git keeps track of the history of all changes made to a project’s files and handles the process of combining changes made by different contributors.

GitHub is an online platform that hosts coding projects. It uses Git software and provides features that help multiple people work on a project at the same time. The online storage area for a project on GitHub is a repository, or “repo” for short.

The GitHub website is public, but Data and Analysis has a private space called the “MoJ Analytical Services” organisation. Access to repositories within the organisation can be restricted to certain individuals, granted to all members of the organisation, or made publicly available. The Analytical Platform user guidance contains guidelines on which access setting to use.

## Benefits of using Git and GitHub

Benefits of Git include:

* **Version Control:** Having access to previous versions of your code ensures your work is secure and reproducible.
* **Branching:** Git allows you to create separate branches of a project so you can develop features without compromising the latest working version of the project. Git then handles the process of merging in the changes when you are ready. This is particularly powerful when multiple people are collaborating on a project.

Benefits of GitHub include:

* **Online access:** The central copy of your project is held online and can be viewed and accessed by collaborators (according to the repository visibility settings).
* **Collaboration:** Everyone with access to the repository can view all contributions. GitHub provides tools to discuss and review each other's work before it is merged into the main branch.
* **Quality assurance (QA) features:** GitHub has features for reviewing and discussing all proposed changes before they are accepted. The discussion page for each proposed change serves as a record of the QA carried out.
* **Project documentation:** README files are shown alongside the folders they sit it, providing a visible location for high-level project documentation.
* **GitHub Issues:** The GitHub Issues feature allows you to log and keep track of suggested changes to the repository, such as enhancements or bug fixes.

## Git in RStudio

There are two ways to use Git in RStudio:

* **The graphical user interface (GUI)** – RStudio contains a Git pane with buttons for interacting with Git and GitHub. The GUI provides all commonly needed actions, but not the full range of Git functionality.
* **Terminal commands** – using the command line interface. This is the general way to interact with Git and isn’t specific to RStudio. For example, the terminal works the same way in the other Analytical Platform coding environments, JupyterLab and Visual Studio Code (whereas each has a different GUI). The command line is the only way to run the full range of Git commands.

In this session, we will provide a step-by-step guide showing how to use both the RStudio graphical user interface and terminal commands for the most common tasks.

# Overview of Git and GitHub concepts

## Git: commits and branches

Once you have set up a project to use Git, you will see a .git folder in your working directory. Git is now monitoring all the changes you make to the directory. You can create some files, and whenever you want, you can tell Git to take a snapshot of their current state. A snapshot is called a commit. We can represent a commit as a blob.

1

You can then continue working - modifying files. At any point, you can take another snapshot. The project now looks like version 2, but version 1 is still saved, and you can look at it or switch back to it if you want. (Note that a real commit in Git don’t have an integer as an identifier, but rather a hexadecimal string called a hash.)

1

2

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.

Git calls the series of commits a branch. By default, Git labels your initial branch "main". (Note that if you are using an older repo, the default branch might be called “master”.)

main

1

2

A branch is just a pointer to a commit. Each time you add a commit, the pointer automatically moves along to the new commit.

main

1

2

3

Typically, you don’t want to develop by adding commits directly to the main branch. If you want to develop a new feature, or change something in the project, you should create a new branch. Then you can develop the new feature by making commits to the new branch. Once the feature is working and has been quality assured, you will merge it back into the main branch. Following this workflow ensures that the main branch is always a fully functioning and stable version of the project.

A new branch starts as just a new pointer to the current commit.

main

new\_feature

If you add more commits, they will now be added to the new\_feature branch.

main

new\_feature

While you are working on your feature branch, you (or a colleague) might make changes to the main branch, so the branches will diverge - each branch contains changes that the other does not have.

main

new\_feature

Eventually, once your feature is complete, you can tell Git to merge it back into the main branch. Git handles this merge process so that the main branch then contains all the changes from both branches. If there are conflicting changes, Git will require you to resolve them manually.

new\_feature

main

Branches are a powerful feature of Git. An important benefit is they allow multiple people to work on separate features at the same time, each working on a separate branch. Git then handles the process of merging all the changes together. Once a branch has been merged, all the commits will be contained in the destination branch, so the merged branch can be deleted.

main

## GitHub: collaborating using branches and pull requests

The course PowerPoint presentation contains a series of animated slides illustrating how colleagues can use GitHub to work simultaneously on a project and synchronise their work. If you are working through the course on your own, see the presenter notes below the slides for a full description.

**A diagram of a network

Description automatically generated**

## File states: staging and committing

This section is also covered fully in the course PowerPoint slides.

**A black background with orange rectangles and white text

Description automatically generated**

# Walk-through of the basic Git workflow

We will now show how to implement the basic steps of using Git and GitHub. We will divide them into four steps.

1. Clone a repository.
2. Create a new branch, edit some files and make some commits.
3. Push the branch to GitHub and merge it into main.
4. Pulling the updated main branch to update your local version.

We will first demonstrate the steps using the RStudio graphical interface and then demonstrate them using Git terminal commands. Following the demonstration, we will give some time for an exercise asking you to follow through the steps on your own to contribute some changes to the practice folder in the course repo.

In the training sessions we will do a live demonstration of the steps. They are also shown in the course PowerPoint file with screenshots.

## Clone a repository

If you want to start a new project, create a repository by following the steps in the Analytical Platform user guidance. If you want to contribute to an existing repository, find it on the [MoJ Analytical Services organisation](https://github.com/moj-analytical-services) on GitHub. Then download a copy of the repo to your Analytical Platform directory. The user guidance gives the steps. Downloading a copy of the repository that you can work on is called cloning the repository.

For the demonstration and exercise we will use the repo for this training session: <https://github.com/moj-analytical-services/intro_to_github_training>.

## Create a branch, edit some files, make some commits

Once you have cloned a repo, you are ready to contribute to the project. You will probably be on a branch called “main” by default, although the repo might be set up with a different default branch.

**Create a branch**

To contribute some work, first make a new branch. Some tips for naming a branch are:

1. Use a descriptive name that describes the purpose of the branch
2. Be concise
3. Use – or \_ as separators between words

**Edit some files**

Next, make some changes to one or more files in your repository. You can create, modify or delete files – any changes you make to the files in the project folder will be captured by Git. For example, you could create a new R script and write some code in it. Make sure you have saved any changes within files.

**Stage and commit the changes**

Once you are happy with the changes you have made, you are ready to make a commit. A commit can contain changes to any number of files. Ideally, a commit should be a self-contained, working unit of change to your code.

Add the files you want to commit to the “staging area”. The files in the staging area will be those included in the next commit. You might want to include all the changes you have made or just some of them.

Then commit the changes. Every commit needs a commit message in which you should briefly describe the changes made. The commit message will appear alongside the commit in the repository history, helping you and others to see what changes were made, so it is helpful to be descriptive but concise.

## Push the branch to GitHub, open a pull request, and merge

**Push the branch to GitHub**

Once you have committed your changes locally, you can 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.

**Open a pull request**

When you have finished making all the changes you want to include in your branch, you will want to merge your branch into the main branch of the repository. There are two ways to merge a branch, either locally in the terminal or on GitHub via a “pull request” (PR). If you are merging into the main branch, you will usually want to do that as a pull request.

A pull requests is the GitHub feature that handles the merge process on the remote repo. A pull request allows collaborators on a project to review and discuss the proposed changes before the branch is merged, using the pull request discussion page. The repo can be configured so that every pull request must be reviewed and approved before it can be merged.

Aside from quality assurance, pull requests also help with project management. The PR description at the top of the discussion thread can be used to explain the proposed changes and link to project planning software or an issues log.

If you recently pushed changes to GitHub and navigate to the online repo, GitHub notices the changes to the branch and shows a banner with the option to open a pull request. Otherwise, to open a pull request:

* + 1. Go to the "Pull request" tab.
    2. Click "New pull request".
    3. Use the drop-down selectors to select the base branch (e.g. main) and comparison branch (e.g. your new feature branch).
    4. Click "Create pull request".

You can continue to add commits to your branch while the pull request is open. The pull request page will update to include the new commits.

**Merge the branch**

To merge a pull request, select "Merge pull request" at the bottom of the discussion thread.

Once you have merged the pull request, GitHub will give you the option to delete the branch. This is safe to do because all the commits from the branch have now been incorporated into the main branch. The pull request page, with the discussion thread and review comments, will remain on GitHub – it is just the branch label that will be deleted. Deleting merged branches keeps the repo tidy, making it easier to find the branches that are currently being worked on.

## Pull the updated branch from GitHub

**Pull changes into a local branch**

Once you have merged the pull request into the main branch on GitHub, you will want to update your local copy of the main branch to match. In your local repo, make sure you are on the branch you want to update, in our example the main branch. Then use the pull command to update the local branch. Once the local branch is updated you are ready to repeat the workflow and create a new branch for your next contribution.

**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 the RStudio interface or the git log command.

## Exercise

Now we will practice the workflow we have just been through. You can use either the RStudio graphical interface or terminal commands for the exercise.

Slides have been uploaded to the GitHub repo showing you the steps, which you can refer to during the exercise.

1. **Clone the course repository and create a new branch**
   * 1. If you have not done so already, clone the course repo by following the steps in the set-up section at the start of this document.
     2. Create a branch with any name you choose.
2. **Create and edit a file and commit the changes**
   * 1. Create a new folder within ‘practice\_folder’ and then create a new R script within the folder. Write something in the R script. Press ‘save’ to save the script.
     2. Add the changes to the ‘staging area’ and then commit them. Add a commit message that briefly describes the changes you have made.
3. **Push the changes to Github**
   * 1. Now that you have committed your changes to the local version of your new branch, push the updated branch to the remote Github repo.
4. **Open a pull request on GitHub and merge**
   * 1. Navigate to the GitHub repo and open a pull request to merge your branch into main.
     2. Merge the branch using the option at the bottom of the pull request page.
5. **Pull the updated version of the main branch into your local version of the repo**
   * 1. In RStudio, switch to the main branch. Look in the file pane and note that files you made in your new branch are not shown. You might get a dialogue box asking you if you want to close open files that are not in the main branch.
     2. Pull changes from GitHub and notice that the file pane updates to include files from the new branch.
     3. Check the commit history to see that your commits are now part of the main branch.

# Advanced topics

## Resetting changes

Git has options for making changes to work you have already staged or committed. It is straightforward to unstage changes that you have not yet committed. It is also straightforward to make changes to your most recent commit – provided you haven’t pushed it to GitHub. It is easiest to not edit an earlier commit than that, or a commit you have already pushed to GitHub.

To unstage a file that you have staged but not yet committed, untick the checkbox by the filename in the RStudio Git pane.

If you want to edit your most recent commit, you can amend it using the RStudio interface. Make the changes you want, stage them, and click on “commit”, but before typing a commit message, tick the “Amend previous commit” checkbox.

The terminal commands for resetting changes include:

* git reset, which can be used to reset the staging area (but can also be used to undo commits)
* git commit --amend allows you to amend the previous commit by adding new staged changes.
* git revert, to make a commit that reverses a previous commit

The details of using those commands are beyond the scope of this course. There are links to some resources in section 7.

## Merge conflicts

Git can normally handle the process of merging changes made in different branches automatically. However, if two branches have made changes to the same lines of code, Git does not know which change to accept, and will inform the user that there is a merge conflict that must be resolved manually before the merge can complete.

Git will add the changes from both branches to the file containing the conflict, surrounded by conflict markers as shown below.

A screenshot of a computer code

Description automatically generated

The code between <<<<<<< and ======= is the code in the current branch. (“HEAD” is the label Git uses to refer to the current commit.) The code between ======= and >>>>>>> is the code in the branch that is being merged in. To continue with the merge, edit the code to contain the final version you want. That could be the version from either of the two branches or some new combination. Then delete the conflict markers. To complete the merge, stage the file and commit.

See the links in the additional resources section below for more details of resolving conflicts using RStudio and the Git command line.

## Stashing changes

git stash is a command that temporarily stores your work without making a commit, allowing you to re-apply it later. An example where this can be useful is if you need to temporarily switch branches but aren’t ready to commit yet.

# Effective collaboration

We will list some standard good practice when collaborating with others on a project. These are useful even when you are working on your own but are especially important when working with others.

**Make small and self-contained commits:** Commit regularly. Ideally each commit should describe a discrete change or group of associated changes.

**Clear commit messages:** Commit messages should summarise the changes made so that it is easy to read through the commit history and identify the commit that introduced a particular change.

**Small branches and regular merges:** Make sure each branch has a clearly defined purpose and don’t let the scope expand to include anything extra as you work on it. This will make it easier to review pull requests and reduce the risk of merge conflicts. If a branch is getting too big, consider whether you can split the changes into a few smaller branches.

**Pull request descriptions:** Use the pull request description to describe the purpose of the pull request and summarise all the changes introduced. You can also use this space to suggest to colleagues how to review and test the changes.

**Branch protection rules:** Project administrators can add branch protection rules to control how changes can be made to a branch. For a team project, it is a good idea to protect your main branch by requiring pull requests before merging and by requiring approvals. That will ensure no changes are made to the main branch that haven’t been reviewed and approved by a colleague.

**Delete merged branches:** Once a branch has been merged, it is safe to delete it since all the commits will be part of the target branch. The closed pull request page remains, so there is a record of what was merged, and there is an option there to restore the branch if needed. Deleting merged branches makes active branches easier to find since they aren’t obscured by a long list of old branches.

**Document your repository:** When you create a new repository, you will be given the option of initialising it with a README file. The [Analytical Platform User Guidance](https://user-guidance.analytical-platform.service.justice.gov.uk/github/create-project.html#readme) recommends selecting that option. (Alternatively, you can add a README.md file to the top level of the repository later.) The README file should minimally tell readers the purpose of the repository and how to get started using it.

# Security considerations when using GitHub

## Repository visibility

GitHub is a public website visible to anyone on the internet. When creating a repository, you have the option to make it public, internal (to MoJ Analytical Platform users) or private (restricted to named users). For guidance on which to use, see the Analytical Platform User Guidance on [repository visibility](https://user-guidance.analytical-platform.service.justice.gov.uk/github/create-project.html#repository-visibility).

## Storing data

The Analytical Platform Guidance page [Security in GitHub](https://user-guidance.analytical-platform.service.justice.gov.uk/github/security-in-github.html#security-in-github) says that GitHub should primarily be used to store code and not data. The [Amazon S3](https://user-guidance.analytical-platform.service.justice.gov.uk/data/amazon-s3/#amazon-s3) cloud storage platform should be used for storing data.

The Analytical Platform is set up so that by default, data files will not be tracked by Git or uploaded to GitHub. If you accidentally publish sensitive data to GitHub, see the Analytical Platform guidance on [accidentally publishing data to GitHub](https://user-guidance.analytical-platform.service.justice.gov.uk/github/security-in-github.html#accidentally-publishing-data-to-github).

## Git ignore files

A .gitignore file lists files that git should ignore. This can include configuration files and data files. The Analytical Platform provides [.ignore templates](https://user-guidance.analytical-platform.service.justice.gov.uk/github/create-project.html#repository-visibility:~:text=can%20use%20it.-,.gitignore,-A%20.gitignore%20file) when you create a new repository.

# Additional resources

MoJ resources:

* Git and GitHub help in the MoJ Analytical Platform User Guidance:

<https://user-guidance.analytical-platform.service.justice.gov.uk/github/index.html#git-and-github>

* Training material and recordings of courses, including this course, from the MoJ Analytical Platform tools training group:

<https://moj-analytical-services.github.io/ap-tools-training/ITG.html#ITG>

Cheat sheets and other resources:

* Cheat sheet “Using Git and GitHub with RStudio”:

<https://rstudio.github.io/cheatsheets/git-github.pdf>

* Git cheat sheet from DataCamp:

<https://www.datacamp.com/cheat-sheet/git-cheat-sheet>

* Online book on using Git and GitHub with R, by Jenny Bryan:

<https://happygitwithr.com/>

* DataCamp GitHub Foundations track:

<https://app.datacamp.com/learn/skill-tracks/github-foundations>

(Everyone at MoJ can access DataCamp. Go to the [sign-up page](https://www.datacamp.com/users/sign_up?group_invite=true&group_name=Justice&group_type=enterprise&policy_version=&redirect=https%3A%2F%2Fwww.datacamp.com%2Fgroups%2Fshared_links%2Fb0711390ec1a1aa7201e99bf85a97e4931d776e8abe651c3fdc9f55eeb5495d0%3Fdc_referrer%3Dmain-group%26tos%3Dtrue) if you don’t yet have an account.)

Undoing changes:

* Undoing the most recent commit, from Jenny Bryan’s book:

<https://happygitwithr.com/reset>

* Detailed tutorial on undoing commits and changes, from Atlassian:

<https://www.atlassian.com/git/tutorials/undoing-changes>

Merge conflicts:

* Dealing with conflicts in RStudio, from Jenny Bryan’s book:

<https://happygitwithr.com/git-branches#dealing-with-conflicts>

* Atlassian tutorial on understanding and resolving merge conflicts:

<https://www.atlassian.com/git/tutorials/using-branches/merge-conflicts>