**Git / GitHub Cheat sheet**

1. **SSH Keys**

After setting up a GitHub account and downloading the Git client (inc Git Bash) you will need to set up the SSH keys between the GitHub account and your client PC to prove who you are. If you don’t do this then nothing will work! You will only need to do this once (per connected PC).

<https://docs.github.com/en/authentication/connecting-to-github-with-ssh/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent>

1. Generate a local SSH key.

Load the Git Bash client window. At the $ prompt type (case sensitive):

1. **ssh-keygen –t rsa –b 4096 –C “**[**email@example.com**](mailto:email@example.com)**”**

***-t rsa*** *is the encryption type.*

***-b 4096*** *is the encryption type.*

***-C “***[***email@example.com***](mailto:email@example.com)***”*** *is the email address you used when creating the GitHUb account.*

1. Enter file in which to save the key: **keyfilename**

***Keyfilename*** *can be anything you like just make it identifiable e.g.* ***GHHomePCKey***

*This key file will be created here:* ***C:\Users\YourUserName\.ssh\***

1. Enter Passphrase (empty for no passphrase**): yourpassphrase**

*You don’t have to have a passphrase however I suggest that you do for security. Make it strong (e.g.* ***monument-binoculars-cheesecake****) and write it down in your Bitwarden.*

*Beware: As you enter, you will not see the characters appear on screen, nor will your cursor move.*

*Confirm you passphrase.*

1. Your Key will now be generated. Navigate to the **..\.ssh\** folder in windows explorer.

*You will see 2 (key) files created:*

**keyfilename**

**keyfilename.pub** \*

*\*Note that windows may think this is an* ***Office Publisher*** *file, it isn’t!*

1. Add the public key to GitHub.
2. Open the **keyfilename.pub** file and **copy** its contents.

*Open With… Notepad.exe*

*You will see lots of text that starts with something like:*

***ssh-rsa AAADB3MzaC1yc2EBAAsD***

*And ends with:*

***==*** [***email@example.com***](mailto:email@example.com)

1. In **GitHub** navigate to your Profile Settings and select ‘**SSH and GPG Keys**’ from the menu on the left.
2. Click ‘**New SSH key**’.

Paste the copied contents from above into the ‘**Key**’ field.

Add a title to the ‘**Title**’ field. This can be anything but something like ***GHHomePCKey.***

1. Click ‘**Add SSH key**’

*This will now add the public key to GitHub.*

1. Register the local key with your local PC

Load the Git Bash client window. At the $ prompt type (case sensitive):

1. **eval “$(ssh-agent –s)”**

*You will see the following appear:*

* *Agent pid 59566*

1. **ssh-add C:\Users\YourUserName\.ssh\keyfilename**

Enter the **passphrase** you created earlier.

*This registers your local key with Git on your PC.*

1. **Git Commands**
2. **git status**

Provides the status on the current local repository. It will show any files (in your project) that have been modified but not added to the *commit pre-list*.

1. **git add <filename> or git add .**

Add either a single filename or ALL files from the not added list to the ‘**staged’** list.

1. **git reset / git reset filename**

Will **‘un-stage’** any added file(s) prior to commit but does not undo the changes within the files (see git restore).

1. **git reset HEAD~1**

Used after an accidental commit to revert back past the last commit by 1. **Will place the files at the last commit back into an un-staged and un-committed state.**

1. **git reset <commit hash>**

Allows you to reset back to a specific commit using that commit’s unique has code. The hash code can be located by invoking the command **git log** (see below) to list all of the commits made.

1. **git reset --hard <commit hash>**

Using the **--hard** switch (notice that a double ‘-‘ is used) on the git reset command, will also remove ALL changes made up to that commit and set this commit as the **HEAD** of the branch (i.e. as far as git is concerned, this was the last commit ever made on this branch.

**I suggest being ultra-careful with this.**

1. **git commit –m “**<mandatory brief comment>**”** **–m “**optional long comment**”**

Commits the added files on the local repository.

1. **git commit –am “message”**

Adds and commits files at the same time, so long as they are not new but updates.

1. **git restore <filename>**

Restores the changes made to the file in the local repository since the last commit was made.

1. **git push origin main**

Pushes the files to the GitHub server in the repository and branch (in this case main).

1. **git push –u origin main**

Invokes the push (as point j above), however the **–u** adds the ‘**upstream**’ qualifier, setting origin main **as the default branch to push to**.

*From this point forward simply typing git push will push to origin main automatically.*

1. **git push --set-upstream origin branchname**

This version of git push is used when pushing to a new branch (or a branch that is not main). A new upstream (server) branch is created and the changes are pushed to that instead of main. In future, from then on, simply using git push will push to the new branch.

1. **git remote – v**

Checks the remote status of the current active local repository (location).

Returns:

origin https://github.com/username/repositoryname.git (fetch)

origin https://github.com/username/repositoryname.git (push)

1. **git branch**

Will return the name of the current branch you are in: e.g.

**\* main**

**\* hotfix**

1. **git branch –d branchname**

Deletes the branch branchname.

1. **git branch –m master main**

Rename local branch to match that of server (if git has set local to be master). Branch name master is deprecated, using master locally and main on the server is only going to cause you headaches so fix this and move on.

1. **git checkout branchname**

Switches between branches.

1. **git checkout - b branchname**

Creates and switches to a new branch.

1. **git diff branchname**

*Compares two versions of the code and shows which lines/files have changed. you need to be in one branch (say ‘main’) to compare to the other (‘say ‘database-crash-hotfix’) for it to work – invoking the command ‘git diff database-crash-hotfix’ whilst currently in the database-crash-hotfix branch will yield TSROFA.*

1. **git diff --name-only HEAD~10 HEAD~5**

**(or git diff --name-only SHA1 SHA2)**

Compares 2 commits and lists just the files that changed between the commits.

1. **git merge branchname**

Will merge the branch into your current branch. e.g.:

**(main) git merge database-crash-hotfix**

*The branch will be merged into main.*

1. **git log**

Lists a log of your commits (in revere chronological order (latest at the top of the list first)). A sample of a log entry looks like this:

commit **418f56ea1d1652ce7109d07dc71a2327e4a43d2d** (origin/main, origin/HEAD, main)

Author: user1 <60434121+user1@users.noreply.github.com>

Date: Fri Oct 22 08:44:15 2021 +0100

hello everybody 🡨 This is the commit message

**Important:** Notice the commit message here, this is a good example as to why you must use a clear, logical message to understand which commit was which!

The long line of letters and numbers (418f56ea1d1652ce7109d07dc71a2327e4a43d2d) is the unique **commit hash** code. This allows you to **reset** back to a certain commit using this code. e.g.

**git reset 0bb7a552574255245169c0690c63506678cae226**

1. **Create a repository locally**

You are going to want to create new local repositories for projects.

1. **Navigate to the project folder in Git Bash or Command Prompt or VS Terminal.**
2. Issue command **git init**.

*You will receive a message that’s says ‘****Initialized empty Git repository****…’*

1. In GitHub create a **New Repository.**

*Enter a Repository Name (that matches your new project locally).*

*Click* ***Create Repository***

1. Copy the HTTPS url from the new GitHub repository.

*It will look something like:*

https://github.com/username/newrepositoryname.git

1. Back in your terminal window (from point a above), issue command:

**git remote add origin** https://github.com/username/newrepositoryname.git

*This adds the link between your local and remote repositories.*

1. Verify the remote connection has been established.

Issue command: **git remote –v**

*You will see the following returned if all is ok*:

origin https://github.com/username/newrepositoryname.git (fetch)

origin https://github.com/username/newrepositoryname.git (push)

1. **Clone a GitHub repository**

If the repository already exists on the GitHub server but does not exist on your local PC then you will need to clone it (assuming you need version control access to the files in the repository). This creates the connection between the remote repository and the clone on your PC.

These notes assume the repository already exists in GitHub.

1. **In GitHub, copy the repository URL.**

Inside the repository, click **Code** button then copy the url. Such as:

https://github.com/username/TestRepo.git

1. **Create a Readme.md file**

*Use this to test later that the clone has worked.*

1. **In your terminal window, navigate to the folder where the new local repository is going to be.**

e.g. C:\Repos\TestRepo\

1. Invoke command: **git clone https://github.com/username/TestRepo.git**

The following (or similar) will appear:

Cloning into 'TestRepo'...

remote: Enumerating objects: 3, done.

remote: Counting objects: 100% (3/3), done.

remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0

Receiving objects: 100% (3/3), done.

*This has now cloned the remote repo on your local pc. Navigate to the folder, and look for the* ***readme.me*** *file from above.*

*Test also with* ***git remote –v*** *command.*

1. **Branching**

Branching allows you to create code off of the main branch to enable you to edit code without affecting the main branch.

Main branch may contain our production code, therefore we should create a new branch called (say) feature branch that we can then create our new features in without breaking the production code branch.

In the terminal:

1. Invoke command: **git branch**.

*This returns the current branch you are in.*

1. Invoke command: **git checkout –b branchname**

This will create and swap to a new branch. Note that the branch name should be descriptive to the need for the branch. For example: **git checkout –b database-crash-hotfix**

1. Edit the files (on the new branch).

Editing and saving the files will now only do so on the new branch. If you swap between branches (with **git checkout branchname**) you will notice that the changes will not be in the other branch.

On the new branch **git add and git commit** as necessary.

1. Move to the main branch and compare file/line changes

Invoke commands:

**git checkout main**

**git diff branchname**

This will show you the differences between the main branch and the branch you were editing on.

1. Push changes to the new branch.

*At this stage, doing a* ***git push*** *will cause an error as we currently do not have an upstream (server side) branch to push to. Therefore we need to create one (it will always be the same name as the local branch name):*

**git checkout** to new branch.

**git status** – to check the files have been committed.

**git push --set-upstream origin branchname**

1. Create a Pull Request (PR).

*A pull request invites the branch we have been working on to be ‘pulled’ (merged) into the main branch. However, before the files/code are merged, the changes can be reviewed, commented on and changes requested by your peers. Changes can still be made on the branch, committed and pushed. Warning!!! It is imperative that any further changes are made on the same branch!*

*Once the PR request has been merged into the main branch, typically the new branch is deleted and a new branch is created going forward.*

On GitHub, you will now see your push with a button ‘**Compare & pull request**’. Pushing this takes us to a screen that allows you create a comment on the changes.

You will see which branch (the base branch) that will be merged into and also the branch to be merged. GitHub checks to see if the two branches can be merged without any issue and will present the notification ‘**Able to merge**’ if this is possible.

**Add a title** for the Pull request. **Insert a comment, which will be a list of the changes (and why)** that have been made.

**Create Pull Request.**

1. Review files and changes, Resolve issues.

Make any comments on the files. **ALL comments MUST be resolved** before a merge can take place.

1. Merge Pull Request.

**Click Merge.**

**Click Confirm merge.**

Message: Pull request successfully merged and closed – is displayed.

1. Pull the merged changes to the local repository.

At this stage, if you go to the main branch locally and check the changes, you will notice that they are not there yet. We need to pull down the merged changes to the local repository:

Ensure you are in the main branch – **git checkout main**.

Type **git pull origin main** (or **git pull** if the upstream default has been set already) at the terminal. The merged changes will have now been pulled to the local repository.

1. Delete the branch.

Once a branch has been merged into the base branch, you no longer use it and it should be deleted.

Ensure you are in the main branch – **git checkout main**.

Type **git branch** at the terminal to check the branch is there.

Type **git branch –d branchname** at the terminal. This will delete the branch.

Type **git branch** at the terminal to confirm deletion.

1. **Merge Conflicts & Undoing Adds and commits**
2. **Forking**