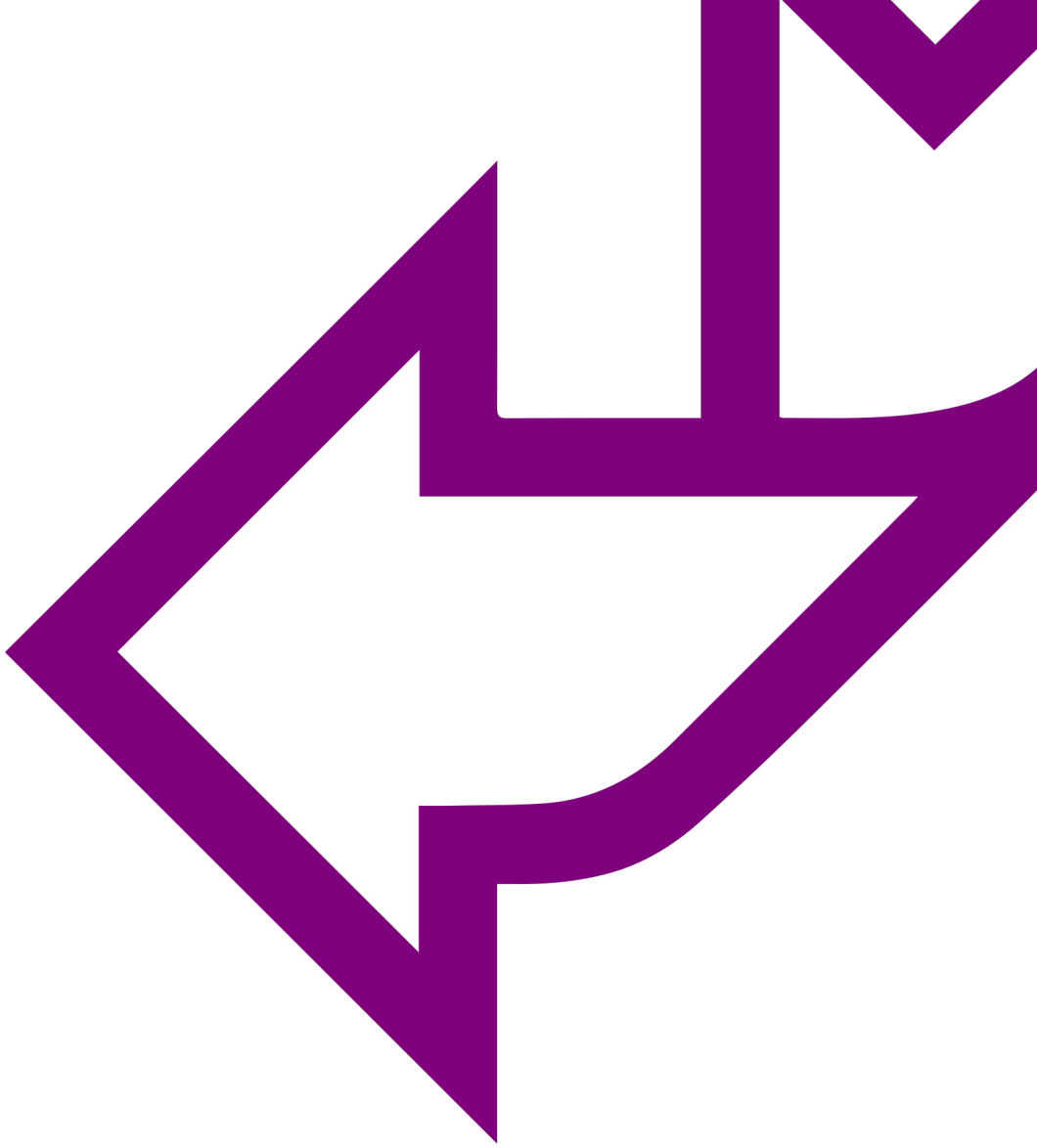
POWERING POTENTIAL



Git Workflow

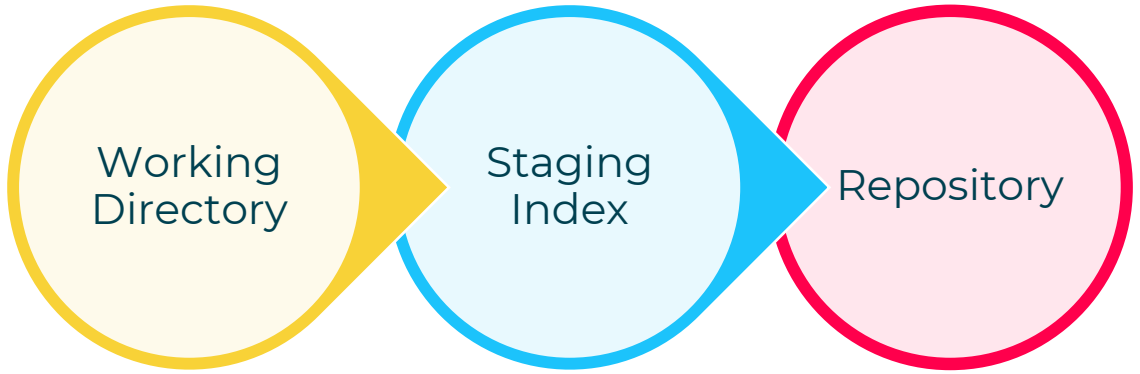
Module 5 - Version Control with GIT

Git's 3-Tree Architecture

1. **Working Directory:** this is where we make changes to our code. The files in this directory are untracked. We can ask Git to ignore some files/ folders in this directory with a **.gitignore** file.

2. **Staging Index:** this contains the changes that will be committed to the repository. We **add** files into this staging index.

3. **Repository:** changes that have been committed and are given a SHA-1 hash value. The history of these changes are stored in the repository and we can revert back to these versions.



GIT WORKFLOW

The general workflow of using Git is init/clone, make changes, add, commit, push

1. Create a new local repo with **git init** or clone an existing remote repo with **git clone**
2. Create/edit files for a feature in the directory. Make code changes as necessary.
3. Ask git to track the files by adding them to the staging index using **git add**
4. Use **git status** to see which files are untracked in the Working Directory and which are Staged but have been modified
5. Add the required files that form part of the change set to the Staging Index with **git add**
6. When satisfied with your progress, commit the files to the Repository with **git commit** passing in a commit message with **–m “message text”**

AN EXAMPLE WORKFLOW (1)

```
Devs-MBP:Example-Repo devgonsai$ ls
README.md
Devs-MBP:Example-Repo devgonsai$ git status
On branch main
Your branch is up to date with 'origin/main'.

nothing to commit, working tree clean
Devs-MBP:Example-Repo devgonsai$ git status
On branch main
Your branch is up to date with 'origin/main'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   README.md
```

We've cloned a repository down that has a README.md file within it. Running an **`ls`** command shows this. We run **`git status`** to see the state of the working area – no changes yet!

README.md

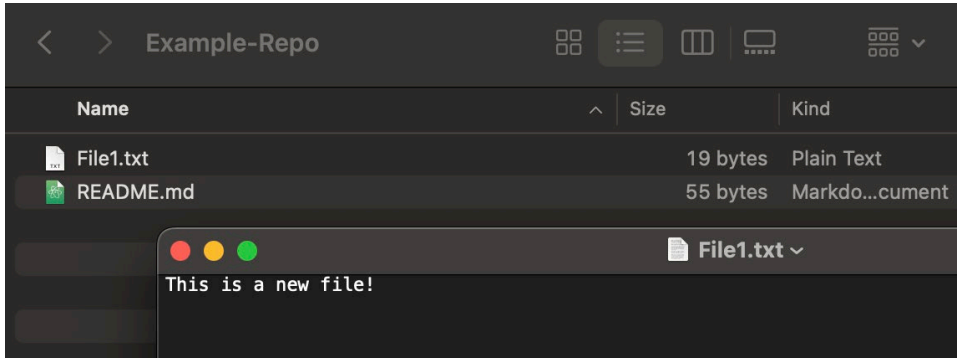
```
1  # Example-Repo
2
3  This is an edit of a file that exists!
```

We make some changes to a file (the README in this example) and save it.

```
Devs-MBP:Example-Repo devgonsai$ ls
README.md
Devs-MBP:Example-Repo devgonsai$ git status
On branch main
Your branch is up to date with 'origin/main'.

nothing to commit, working tree clean
Devs-MBP:Example-Repo devgonsai$ git status
On branch main
Your branch is up to date with 'origin/main'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   README.md
```

Running **`git status`** now shows that a file has been modified – reflective of what we've done in the file previously.

AN EXAMPLE WORKFLOW (2)



We've also added a new file into the directory called File1.txt

```
Devs-MBP:Example-Repo devgonsai$ git status
On branch main
Your branch is up to date with 'origin/main'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   README.md

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        File1.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

Running **`git status`** shows that a file has been modified and another has been created (i.e. untracked)

```
Devs-MBP:Example-Repo devgonsai$ git add .
Devs-MBP:Example-Repo devgonsai$ git status
On branch main
Your branch is up to date with 'origin/main'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   File1.txt
        modified:   README.md
```

We can now stage these files with the **`git add .`** (the period meaning all files)

These files are ready to push onto our GitHub repository.

AN EXAMPLE WORKFLOW (3)

We run the **`git commit`** command with **`-m`** and a message in double quotes. This creates a new revision of the VCS with all files added previously

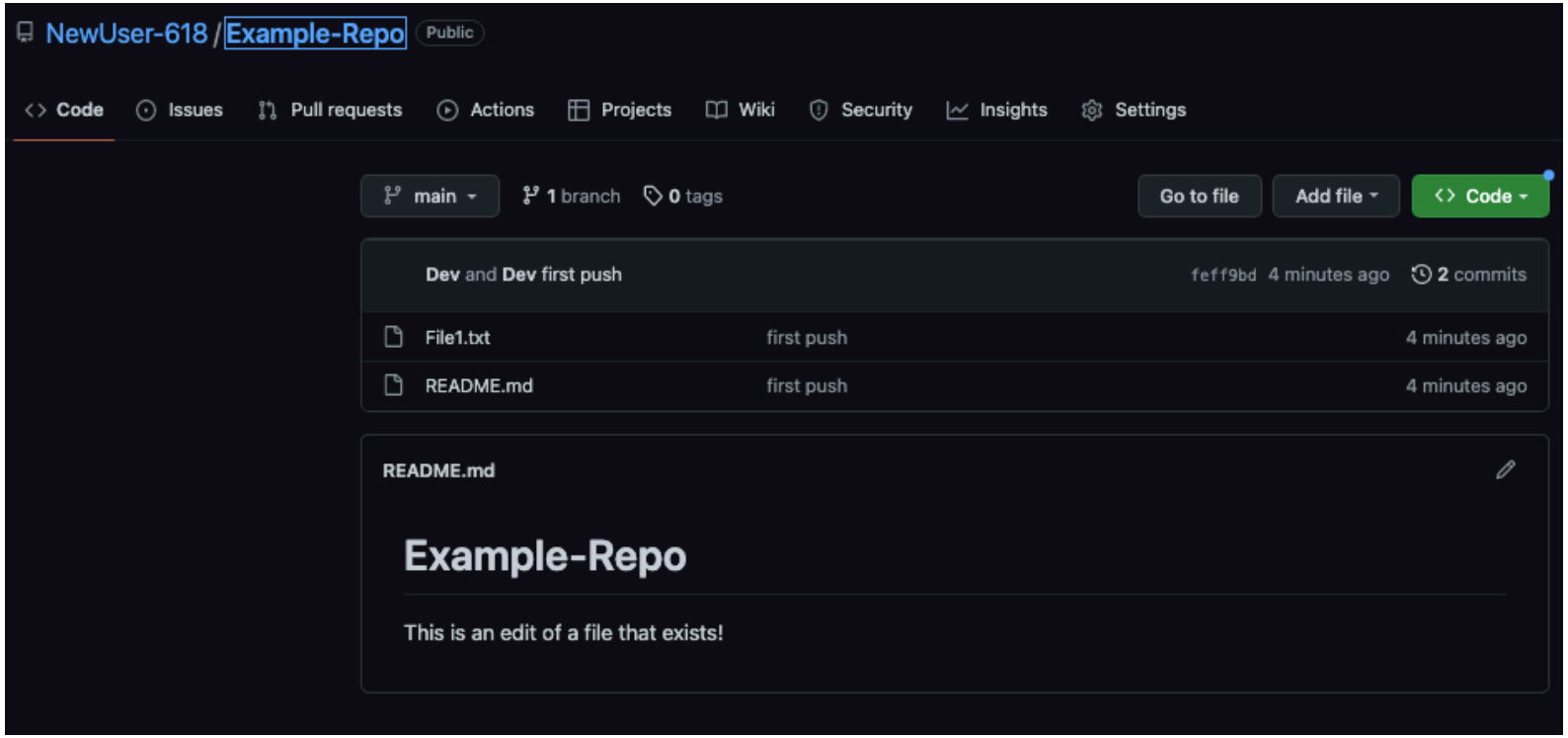
```
Devs-MBP:Example-Repo devgonsai$ git add .
Devs-MBP:Example-Repo devgonsai$ git status
On branch main
Your branch is up to date with 'origin/main'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   File1.txt
        modified:   README.md

Devs-MBP:Example-Repo devgonsai$ git commit -m "first push"
[main feff9bd] first push
 2 files changed, 4 insertions(+), 1 deletion(-)
 create mode 100644 File1.txt
```

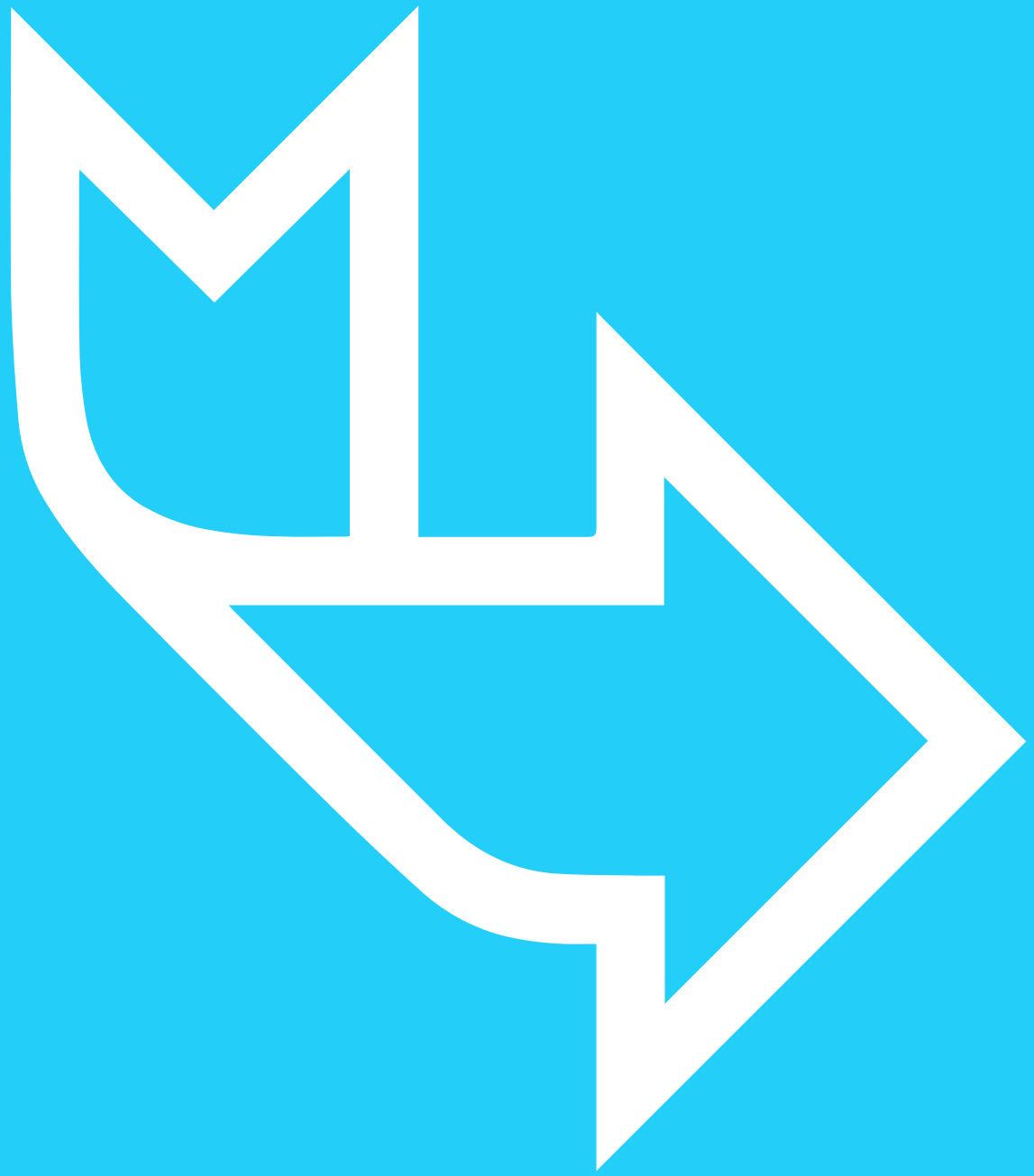
We can then push our changes to our remote repository with the command **`git push`**. We add origin main as this is the branch we're pushing to - more on this later.

```
Devs-MBP:Example-Repo devgonsai$ git push origin main
Enumerating objects: 6, done.
Counting objects: 100% (6/6), done.
Delta compression using up to 8 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (4/4), 349 bytes | 349.00 KiB/s, done.
Total 4 (delta 0), reused 0 (delta 0)
To github.com:NewUser-618/Example-Repo.git
   250c981..feff9bd  main -> main
```

AN EXAMPLE WORKFLOW (4)

Back on GitHub, our changes have been made!!





Exercise 2: Adding files

- Create files
- Stage the files
- Push the changes to your remote repository
- Add a new file, commit, push again

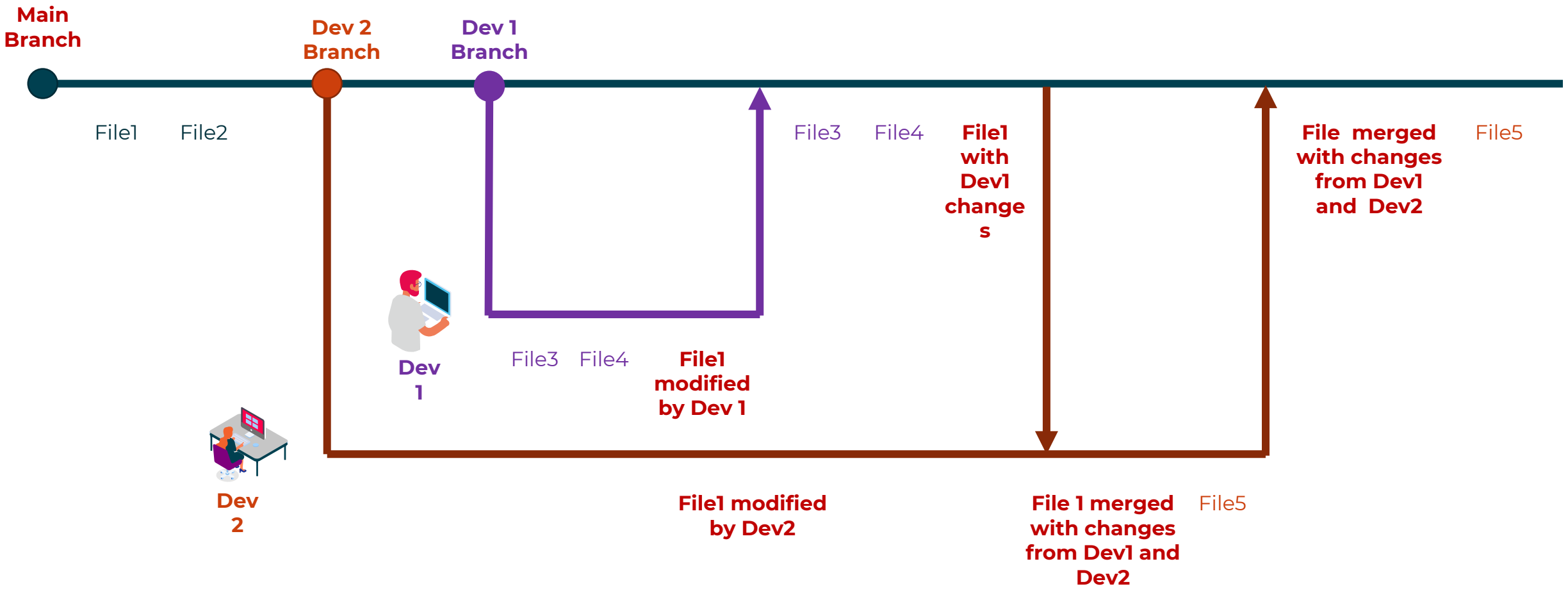
Basic Workflow

This basic git workflow can be improved.

We are currently making code changes on the default branch (**main**).

This is not ideal. Instead we should use a technique called **branching** that allows each new feature or bug fix to be worked on in isolation.

Example: Branching and Merging



Branches allow members of the development team to work on different parts of a project without impacting the main branch. When the work is complete, a branch can be merged with the main branch.

WHY USE BRANCHES?

Branching allows each developer to work on new features in isolation by creating a version of the source code that only they are working on.

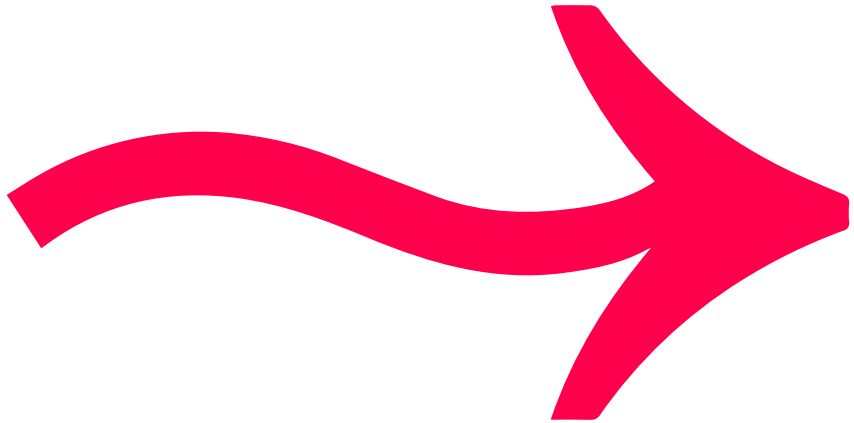
This new version is called a **feature branch**. It shares a common history with the main branch at the point the branch was created.

Once changes have been made and the feature implemented, the feature branch code is **merged** back into the **main** branch.

Git will automatically detect the differences between the **feature** branch and **main** branch and alert the developer if there are any *conflicts*.

Conflicts arise if changes are made to the same lines of code within the feature branch and the main branch after the feature branch has been created.

Creating Branches



You can see all the current branches in your repository

```
git branch
```

To create a new branch, the command is:

```
# git branch [NEW_BRANCH_NAME]
git branch develop
```

This will create a new branch from whichever branch you are currently on (so this new branch will branch from **main** if you run the command whilst on that branch).

If you want to work on a new branch straight away, you can create a branch and checkout to it at the same time:

```
# git checkout -b [NEW_BRANCH_NAME]
git checkout -b develop
```

Deleting Branches

When you have finished working on your branch, and your code has been merged to main, it is good practice to delete the branch:

```
# git branch -d [BRANCH_NAME]
git branch -d feature-123
```

You will also need to do this on your Git Service, (such as GitHub). This is usually done when closing a **Pull Request**.

If you are not closing a Pull Request and find yourself needing to delete a branch, you can use the following command to delete a branch on your remote repository (e.g., on GitHub):

```
# git push --delete origin [BRANCH_NAME]
git push --delete origin feature-123
```

Exercise 3: Branching

- Create a new branch
- Create a file on the new branch
- Add, commit, push
- Inspect GitHub


END OF SECTION