Experiment 04 - Advanced Git Commands

Roll No.	24
Name	Sreekesh Iyer
Class	D15-A
Subject	DevOps Lab
LO Mapped	LO1: To understand the fundamentals of DevOps engineering and be fully proficient with DevOps terminologies, concepts, benefits, and deployment options to meet your business requirements LO2: To obtain complete knowledge of the “version control system” to effectively track changes augmented with Git and GitHub

Aim: To implement Feature Branch workflow strategies in real-time scenarios

Introduction:

Branching in Git:

Branching is a feature available in most modern version control systems. Branching in other VCS can be an expensive operation in both time and disk space. In Git, branches are a part of your everyday development process. Git branches are effectively a pointer to a snapshot of your changes. When you want to add a new feature or fix a bug—no matter how big or how small—you spawn a new branch to encapsulate your changes. This makes it harder for unstable code to get merged into the main code base, and it gives you the chance to clean up your future's history before merging it into the main branch.
A branch represents an independent line of development. Branches serve as an abstraction for the edit/stage/commit process. The git branch command lets you create, list, rename, and delete branches.

git log:
The git log command is used to show the reachable recorded history of a project from the most recent commit snapshot backwards. By default it will only show the history of the branch you're currently on, but can be given different or even multiple heads or branches from which to traverse. It is also often used to show differences between two or more branches at the commit level.

git status:
Displays paths that have differences between the index file and the current HEAD commit, paths that have differences between the working tree and the index file, and paths in the working tree that are not tracked by Git

Merging in Git:

Merging is Git's way of putting a forked history back together again. The git merge command lets you take the independent lines of development created by the git branch and integrate them into a single branch. Git merge will combine multiple sequences of commits into one unified history. In the most frequent use cases, git merge is used to combine two branches.

`git branch`:
The git branch command is actually something of a branch management tool. It can list the branches you have, create a new branch, delete branches and rename branches.

`git branch <branch-name>`:
Create a new branch called ＜branch-name＞. This does not check out the new branch.

`git branch -d <branch>`:
Delete the specified branch. This is a “safe” operation in that Git prevents you from deleting the branch if it has unmerged changes.

`git switch <branch-name>` OR `git checkout <branch-name>`:
To switch to a branch. The name of a local or remote branch that you want to switch to. If you specify the name of an existing local branch, you will switch to this branch and make it the current "HEAD" branch.

`git switch –c <branch-name>` OR `git checkout -b <branch-name>`:
To create and switch to the new branch. The name of a new local branch you want to create. Using the "-c" flag, you can specify a name for a new branch that should be created. You can also specify a starting point (either another branch or a concrete revision); if you don't provide any specific starting point, the new branch will be based on the current HEAD branch.

`git merge <branch-to-be-merged>:`
The git merge tool is used to merge one or more branches into the branch you have checked out. It will then advance the current branch to the result of the merge.The git merge command was first introduced in Basic Branching. Though it is used in various places in the book, there are very few variations of the merge command — generally just git merge <branch> with the name of the single branch you want to merge in.

Workflow

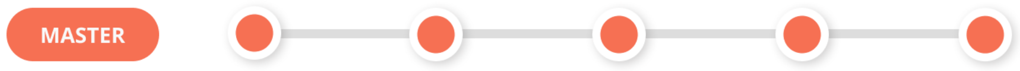
Git is the most commonly used version control system today. A Git workflow is a recipe or recommendation for how to use Git to accomplish work in a consistent and productive manner. Git workflows encourage developers and DevOps teams to leverage Git effectively and consistently.

When working with a team on a Git-managed project, it’s important to make sure the team is all in agreement on how the flow of changes will be applied. To ensure the team is on the same page, an agreed-upon Git workflow should be developed or selected.

Centralized Workflow

The most popular Git development workflow and the entry stage of every project.

The idea is simple: there is one central repository. Each developer clones the repo, works locally on the code, creates a commit with changes, and pushes it to the central repository for other developers to pull and use in their work. There is only one branch — the master branch. Developers commit directly into it and use it to deploy to the staging and production environment.



This workflow isn't usually recommended unless you're working on a side project and you're looking to get started quickly.

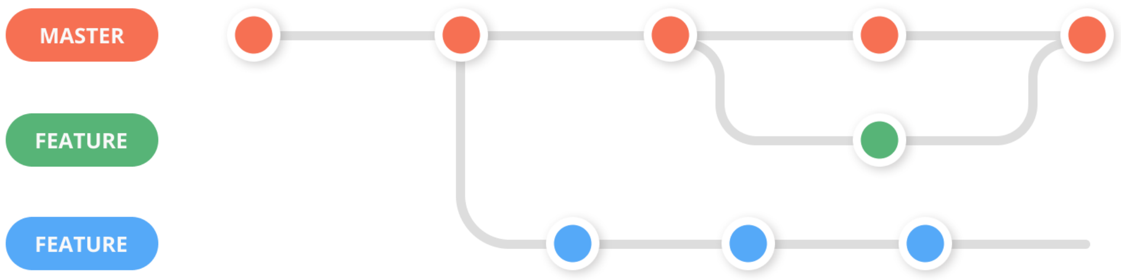
Since there is only one branch, there really is no process over here. This makes it effortless to get started with Git. However, some cons you need to keep in mind when using this workflow are:

1. Collaborating on code will lead to multiple conflicts.
2. Chances of shipping buggy software to production is higher.
3. Maintaining clean code is harder.

Feature Branch Workflow

The Git Feature Branch workflow becomes a must have when you have more than one developer working on the same codebase.

Imagine you have one developer who is working on a new feature. And another developer working on a second feature. Now, if both the developers work from the same branch and add commits to them, it would make the codebase a huge mess with plenty of conflicts.



To avoid this, the two developers can create two separate branches from the master branch and work on their features individually. When they're done with their feature, they can then merge their respective branch to the master branch, and deploy without having to wait for the second feature to be completed.

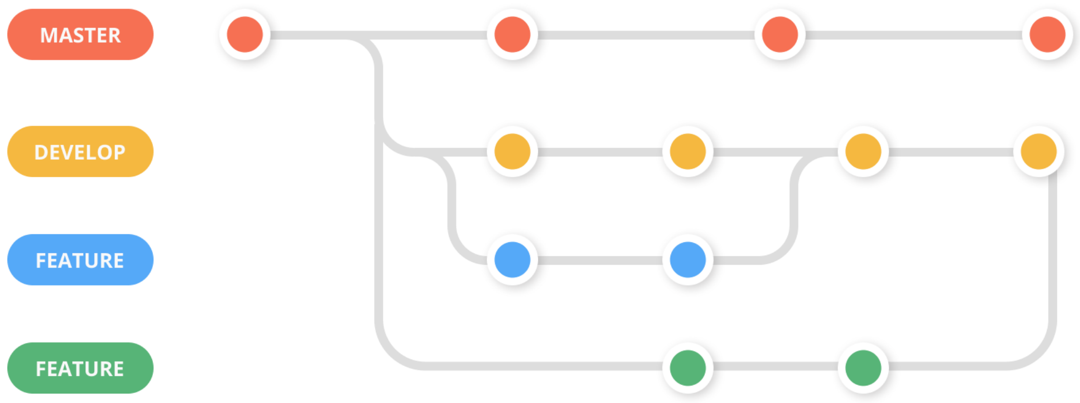
The Pros of using this workflow is, the git feature branch workflow allows you to collaborate on code without having to worry about code conflicts.

Gitflow Workflow

This workflow is one of the more popular workflows among developer teams. It's similar to the Git Feature Branch workflow with a develop branch that is added in parallel to the master branch.

In this workflow, the master branch always reflects a production-ready state. Whenever the team wants to deploy to production they deploy it from the master branch.

The develop branch reflects the state with the latest development changes for the next release. Developers create branches from the develop branch and work on new features. Once the feature is ready, it is tested, merged with the develop branch, tested with the develop branch's code in case there was a prior merge, and then merged with master.



The advantage of this workflow is, it allows teams to consistently merge new features, test them in staging, and deploy to production. While maintaining code is easier, it can get a little tiresome for some teams since it can feel like going through a tedious process.

Forking Workflow

The Forking Workflow is fundamentally different from other popular Git workflows. Instead of using a single server-side repository to act as the "central" codebase, it gives every developer their own server-side repository. This means that each contributor has not one, but two Git repositories: a private local one and a public server-side one.

The flow usually looks like this:

1. The developer forks the open-source software's official repository. A copy of this repository is created in their account.
2. The developer then clones the repository from their account to their local system.
3. A remote path for the official repository is added to the repository that is cloned to the local system.
4. The developer creates a new feature branch is created in their local system, makes changes, and commits them.
5. These changes along with the branch are pushed to the developer's copy of the repository on their account.
6. A pull request from the branch is opened to the official repository.
7. The official repository's manager checks the changes and approves the changes to get merged into the official repository.

The Forking Workflow is most often seen in public open source projects.

Feature Branch Workflow

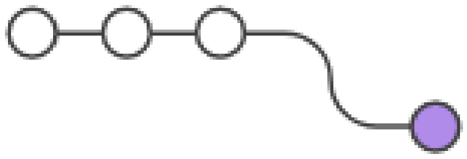
The following is an example of the type of scenario in which a feature branching workflow is used. The scenario is that of a team doing code review around on a new feature pull request. This is one example of the many purposes this model can be used for.

Mary begins a new feature: Before she starts developing a feature, Mary needs an isolated branch to work on. She can request a new branch with the following command:

```
git checkout -b marys-feature main
```

This checks out a branch called marys-feature based on main, and the -b flag tells Git to create the branch if it doesn't already exist. On this branch, Mary edits, stages, and commits changes in the usual fashion, building up her feature with as many commits as necessary:

```
git status
git add <some-file>
git commit
```



Mary goes to lunch: Mary adds a few commits to her feature over the course of the morning. Before she leaves for lunch, it’s a good idea to push her feature branch up to the central repository. This serves as a convenient backup, but if Mary was collaborating with other developers, this would also give them access to her initial commits.

```
git push -u origin marys-feature
```

This command pushes marys-feature to the central repository (origin), and the -u flag adds it as a remote tracking branch. After setting up the tracking branch, Mary can call git push without any parameters to push her feature.

Mary finishes her feature: When Mary gets back from lunch, she completes her feature. Before merging it into main, she needs to file a pull request letting the rest of the team know she's done. But first, she should make sure the central repository has her most recent commits:

```
git push
```

Then, she files the pull request in her Git GUI asking to merge marys-feature into main, and team members will be notified automatically. The great thing about pull requests is that they show comments right next to their related commits, so it's easy to ask questions about specific changesets.

Bill receives the pull request: Bill gets the pull request and takes a look at marys-feature. He decides he wants to make a few changes before integrating it into the official project, and he and Mary have some back-and-forth via the pull request.

Mary makes the changes: To make the changes, Mary uses the exact same process as she did to create the first iteration of her feature. She edits, stages, commits, and pushes updates to the central repository. All her activity shows up in the pull request, and Bill can still make comments along the way.

If he wanted, Bill could pull marys-feature into his local repository and work on it on his own. Any commits he added would also show up in the pull request.

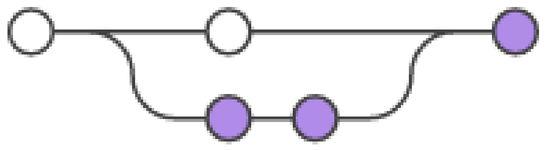
Mary publishes her feature: Once Bill is ready to accept the pull request, someone needs to merge the feature into the stable project (this can be done by either Bill or Mary):

```
git checkout main
git pull
git pull origin marys-feature
git push
```

This process often results in a merge commit. Some developers like this because it's like a symbolic joining of the feature with the rest of the code base. But, if you're partial to a linear history, it's possible to rebase the feature onto the tip of main before executing the merge, resulting in a fast-forward merge.

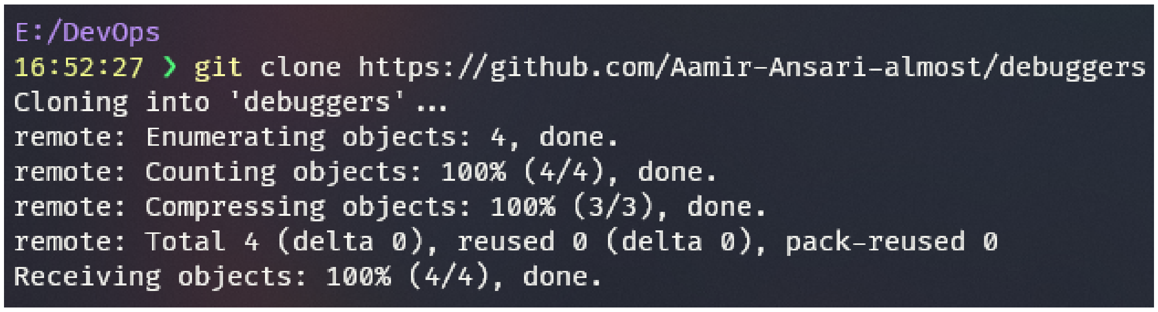
Some GUI's will automate the pull request acceptance process by running all of these commands just by clicking an "Accept" button. If yours doesn't, it should at least be able to automatically close the pull request when the feature branch gets merged into main.

Meanwhile, John is doing the exact same thing. While Mary and Bill are working on Mary's-feature and discussing it in her pull request, John is doing the exact same thing with his own feature branch. By isolating features into separate branches, everybody can work independently, yet it's still trivial to share changes with other developers when necessary.



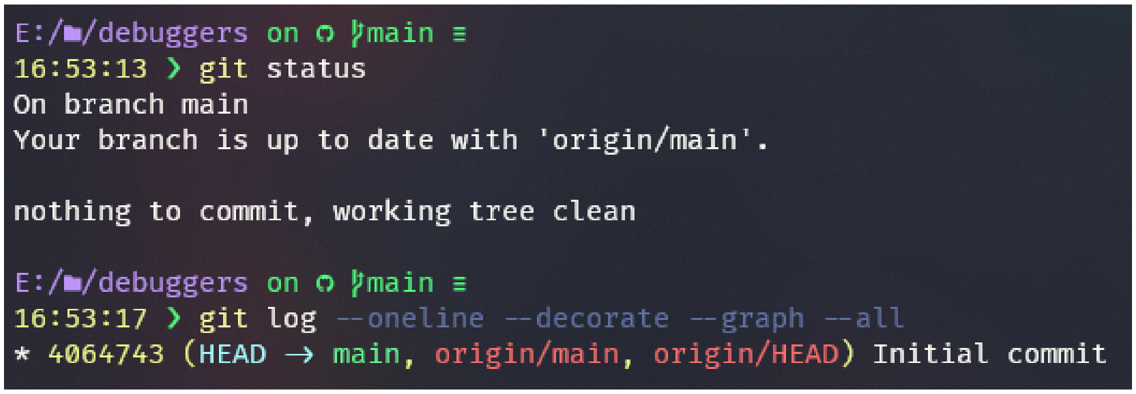
Demonstration of Feature Branch Workflow

1. Clone the repository from github and make sure that you are a collaborator or owner of the repository to make the changes
   a. *git clone <URL of repository>*



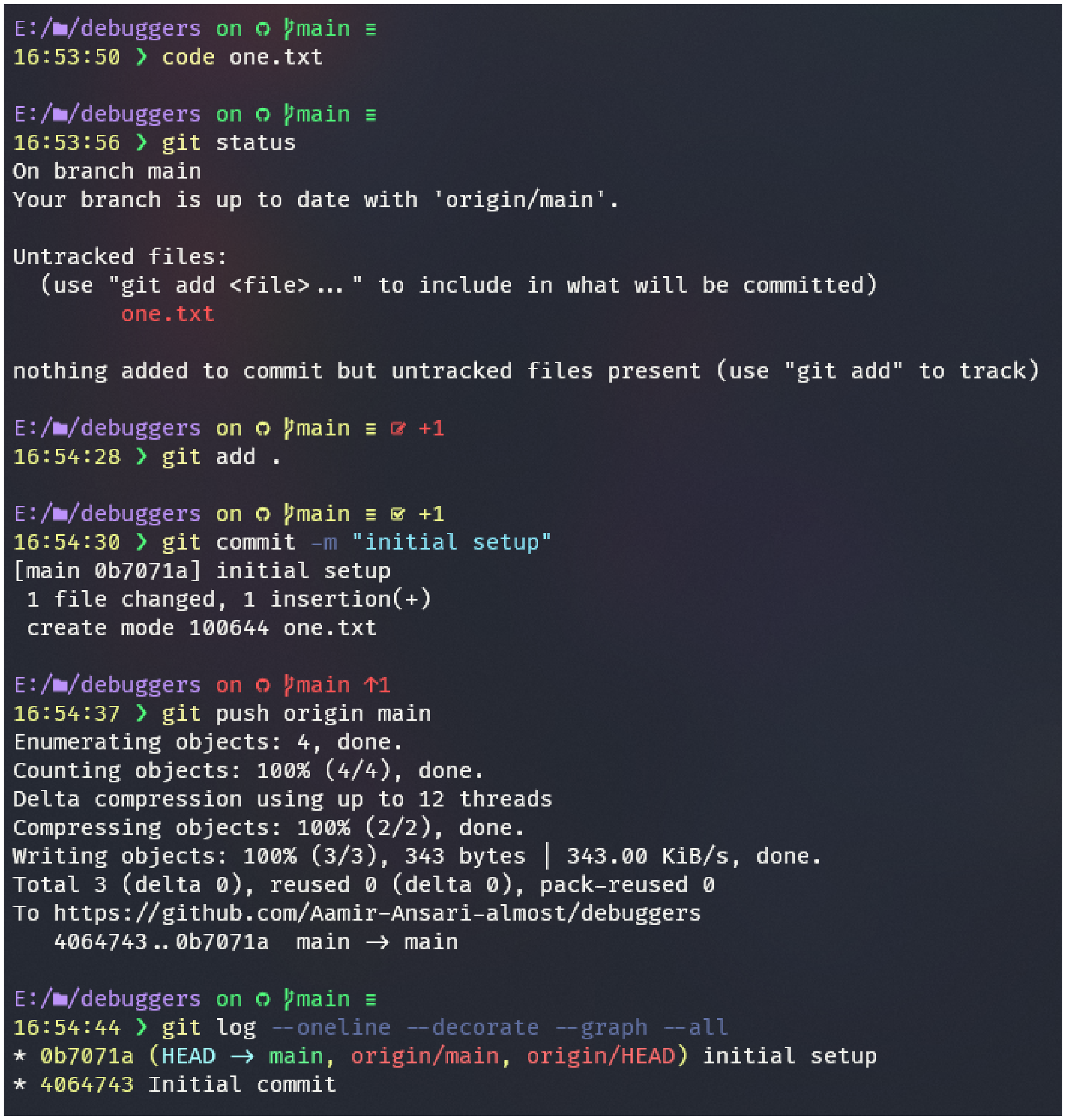
```
E:/DevOps
16:52:27 ❯ git clone https://github.com/Aamir-Ansari-almost/debuggers
Cloning into 'debuggers'...
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 4 (delta 0), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (4/4), done.
```

2. We will be logging after every commit to view the changes. To view the status of repository and logs, use the following command
    a. *git status*
    b. *git log --oneline --decorate --graph --all*



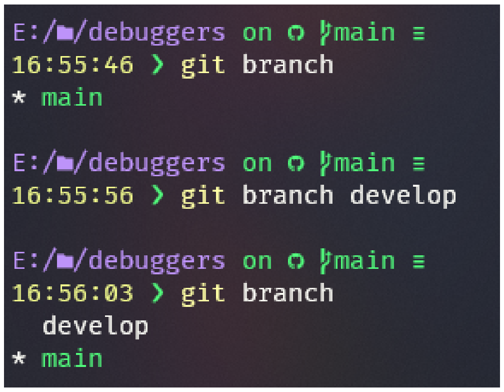
```
E:/■/debuggers on  main ≡
16:53:13 ❯ git status
On branch main
Your branch is up to date with 'origin/main'.

nothing to commit, working tree clean

E:/■/debuggers on  main ≡
16:53:17 ❯ git log --oneline --decorate --graph --all
* 4064743 (HEAD → main, origin/main, origin/HEAD) Initial commit
```

3. We will push something to our main branch (which is usually initial setup or stable version of application)



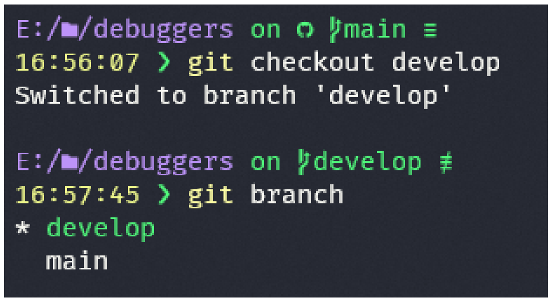
```
E:/■/debuggers on ʘ ᚹmain ≡
16:53:50 ❯ code one.txt

E:/■/debuggers on ʘ ᚹmain ≡
16:53:56 ❯ git status
On branch main
Your branch is up to date with 'origin/main'.

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

nothing added to commit but untracked files present (use "git add" to track)

E:/■/debuggers on ʘ ᚹmain ≡ ☑ +1
16:54:28 ❯ git add .

E:/■/debuggers on ʘ ᚹmain ≡ ☑ +1
16:54:30 ❯ git commit -m "initial setup"
[main 0b7071a] initial setup
 1 file changed, 1 insertion(+)
 create mode 100644 one.txt

E:/■/debuggers on ʘ ᚹmain ↑1
16:54:37 ❯ git push origin main
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 12 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 343 bytes | 343.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/Aamir-Ansari-almost/debuggers
   4064743..0b7071a  main -> main

E:/■/debuggers on ʘ ᚹmain ≡
16:54:44 ❯ git log --oneline --decorate --graph --all
* 0b7071a (HEAD -> main, origin/main, origin/HEAD) initial setup
* 4064743 Initial commit
```

4. For a proper work flow all the development should take place on another branch (conventionally `develop` branch) and the main branch should point towards a stable version of the application. So we create a branch `develop` branch with the following command
    a. *git branch develop*



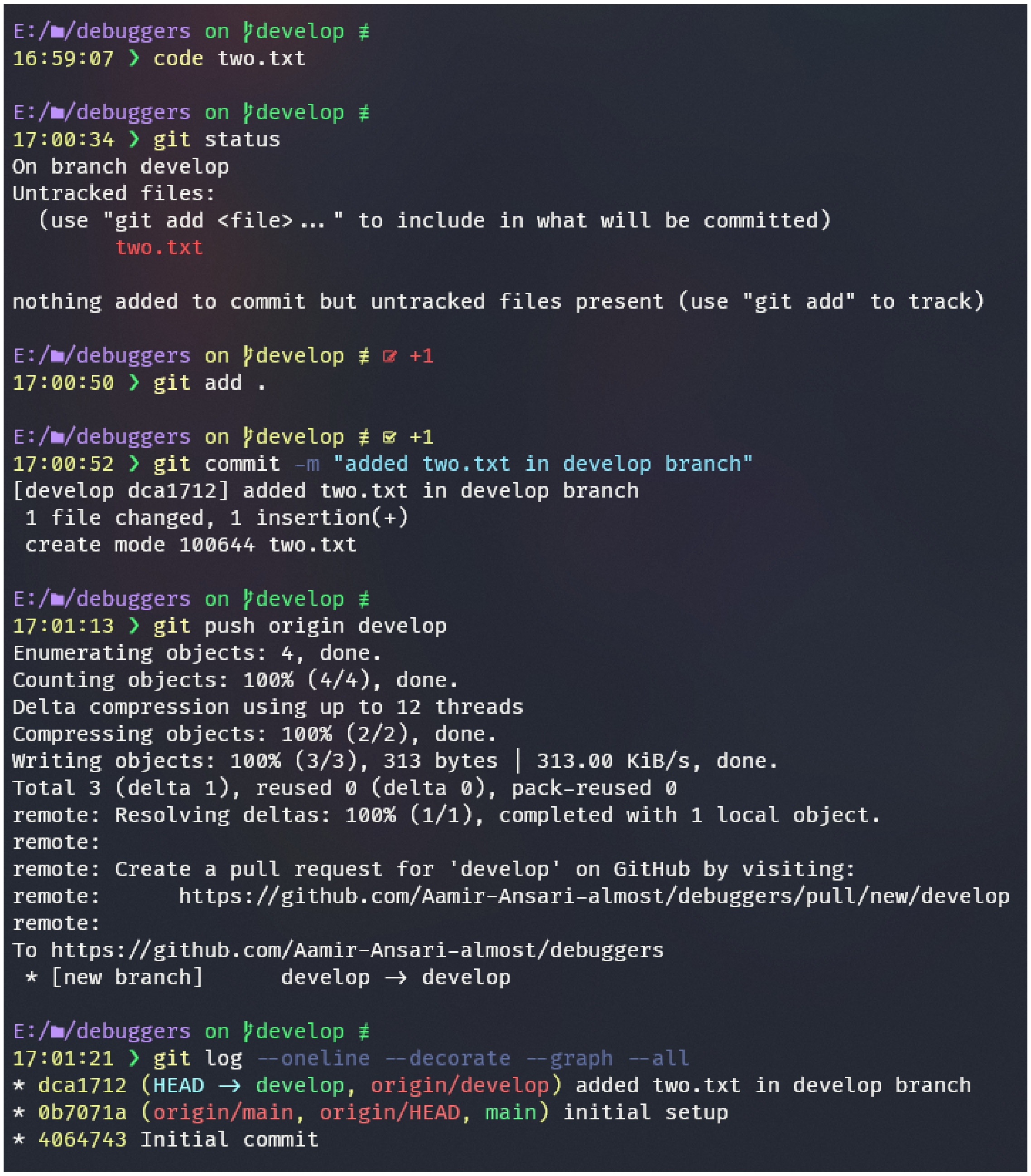
```
E:/■/debuggers on  main ≡
16:55:46 ❯ git branch
* main

E:/■/debuggers on  main ≡
16:55:56 ❯ git branch develop

E:/■/debuggers on  main ≡
16:56:03 ❯ git branch
  develop
* main
```

5. To shift pointer to develop branch, use the following command
    a. *git checkout develop*



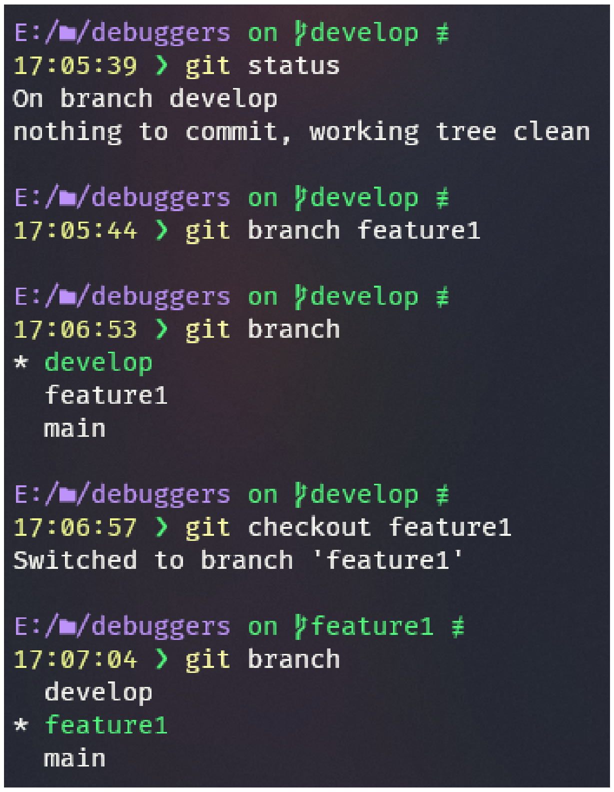
```
E:/■/debuggers on  main ≡
16:56:07 ❯ git checkout develop
Switched to branch 'develop'

E:/■/debuggers on develop ≢
16:57:45 ❯ git branch
* develop
  main
```

6. We can start our development on the develop branch, make two.txt and push it to remote repository



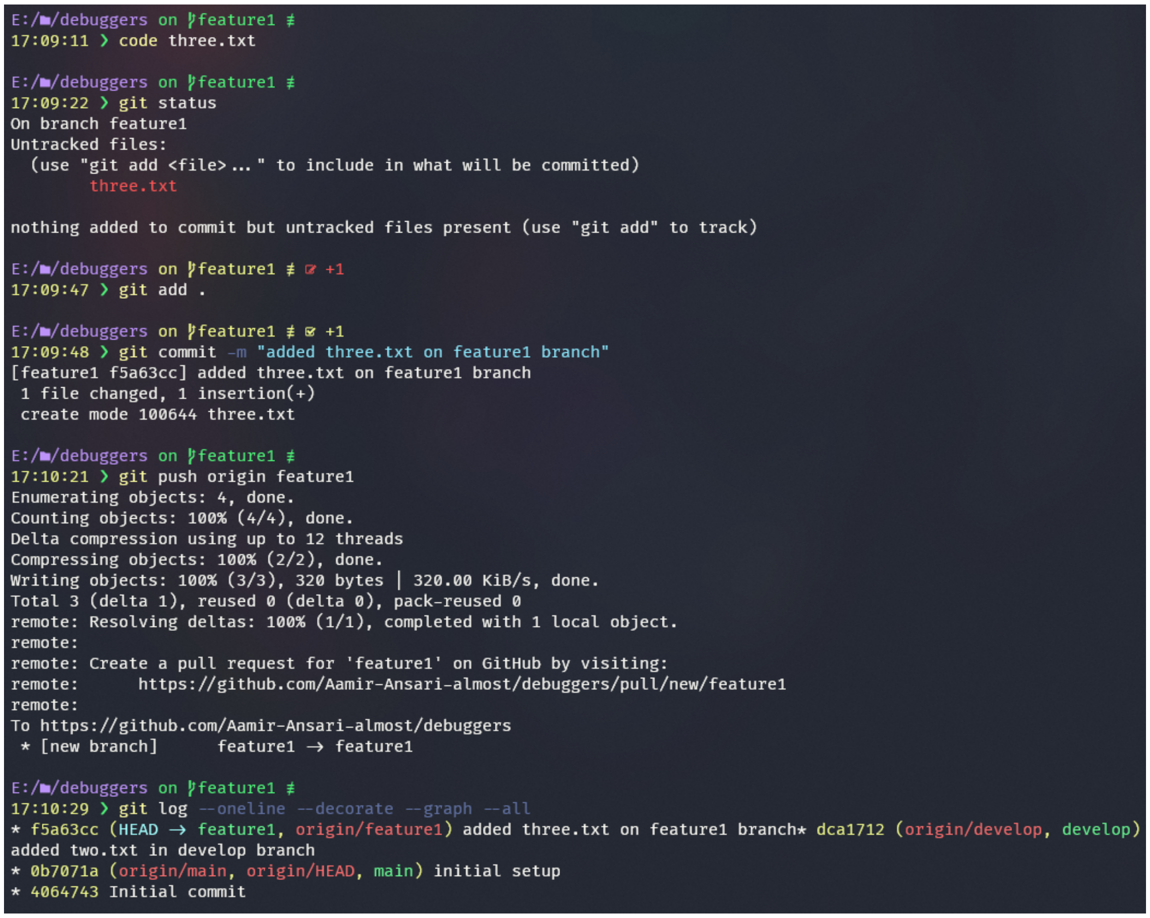
```
E:/■/debuggers on develop ≢
16:59:07 ❯ code two.txt

E:/■/debuggers on develop ≢
17:00:34 ❯ git status
On branch develop
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        two.txt

nothing added to commit but untracked files present (use "git add" to track)

E:/■/debuggers on develop ≢ +1
17:00:50 ❯ git add .

E:/■/debuggers on develop ≢ +1
17:00:52 ❯ git commit -m "added two.txt in develop branch"
[develop dca1712] added two.txt in develop branch
 1 file changed, 1 insertion(+)
 create mode 100644 two.txt

E:/■/debuggers on develop ≢
17:01:13 ❯ git push origin develop
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 12 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 313 bytes | 313.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
remote:
remote: Create a pull request for 'develop' on GitHub by visiting:
remote:      https://github.com/Aamir-Ansari-almost/debuggers/pull/new/develop
remote:
To https://github.com/Aamir-Ansari-almost/debuggers
 * [new branch]      develop -> develop

E:/■/debuggers on develop ≢
17:01:21 ❯ git log --oneline --decorate --graph --all
* dca1712 (HEAD -> develop, origin/develop) added two.txt in develop branch
* 0b7071a (origin/main, origin/HEAD, main) initial setup
* 4064743 Initial commit
```

7. Now if we have to add some feature on top of the following code, we make feature branches, on top of `develop` branch. To make a feature branch, use the following code.
    a. *git branch <branch name>*



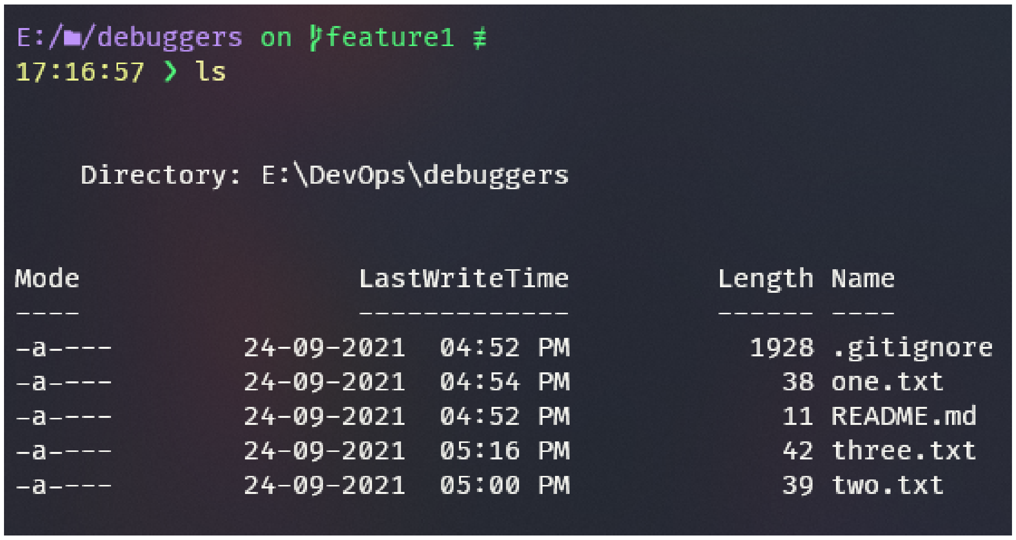
```
E:/■/debuggers on develop ≢
17:05:39 ❯ git status
On branch develop
nothing to commit, working tree clean

E:/■/debuggers on develop ≢
17:05:44 ❯ git branch feature1

E:/■/debuggers on develop ≢
17:06:53 ❯ git branch
* develop
  feature1
  main

E:/■/debuggers on develop ≢
17:06:57 ❯ git checkout feature1
Switched to branch 'feature1'

E:/■/debuggers on feature1 ≢
17:07:04 ❯ git branch
  develop
* feature1
  main
```

8. Now that we are on feature1 branch we can freely add features to our application, without worrying about other developers who are working on other versions. We add some content from `feature1` branch and push it onto remote repository



```
E:/■/debuggers on ⎇feature1 ≢
17:09:11 ❯ code three.txt

E:/■/debuggers on ⎇feature1 ≢
17:09:22 ❯ git status
On branch feature1
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        three.txt

nothing added to commit but untracked files present (use "git add" to track)

E:/■/debuggers on ⎇feature1 ≢ ✎ +1
17:09:47 ❯ git add .

E:/■/debuggers on ⎇feature1 ≢ ✎ +1
17:09:48 ❯ git commit -m "added three.txt on feature1 branch"
[feature1 f5a63cc] added three.txt on feature1 branch
 1 file changed, 1 insertion(+)
 create mode 100644 three.txt

E:/■/debuggers on ⎇feature1 ≢
17:10:21 ❯ git push origin feature1
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 12 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 320 bytes | 320.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
remote:
remote: Create a pull request for 'feature1' on GitHub by visiting:
remote:      https://github.com/Aamir-Ansari-almost/debuggers/pull/new/feature1
remote:
To https://github.com/Aamir-Ansari-almost/debuggers
 * [new branch]      feature1 -> feature1

E:/■/debuggers on ⎇feature1 ≢
17:10:29 ❯ git log --oneline --decorate --graph --all
* f5a63cc (HEAD -> feature1, origin/feature1) added three.txt on feature1 branch* dca1712 (origin/develop, develop)
added two.txt in develop branch
* 0b7071a (origin/main, origin/HEAD, main) initial setup
* 4064743 Initial commit
```

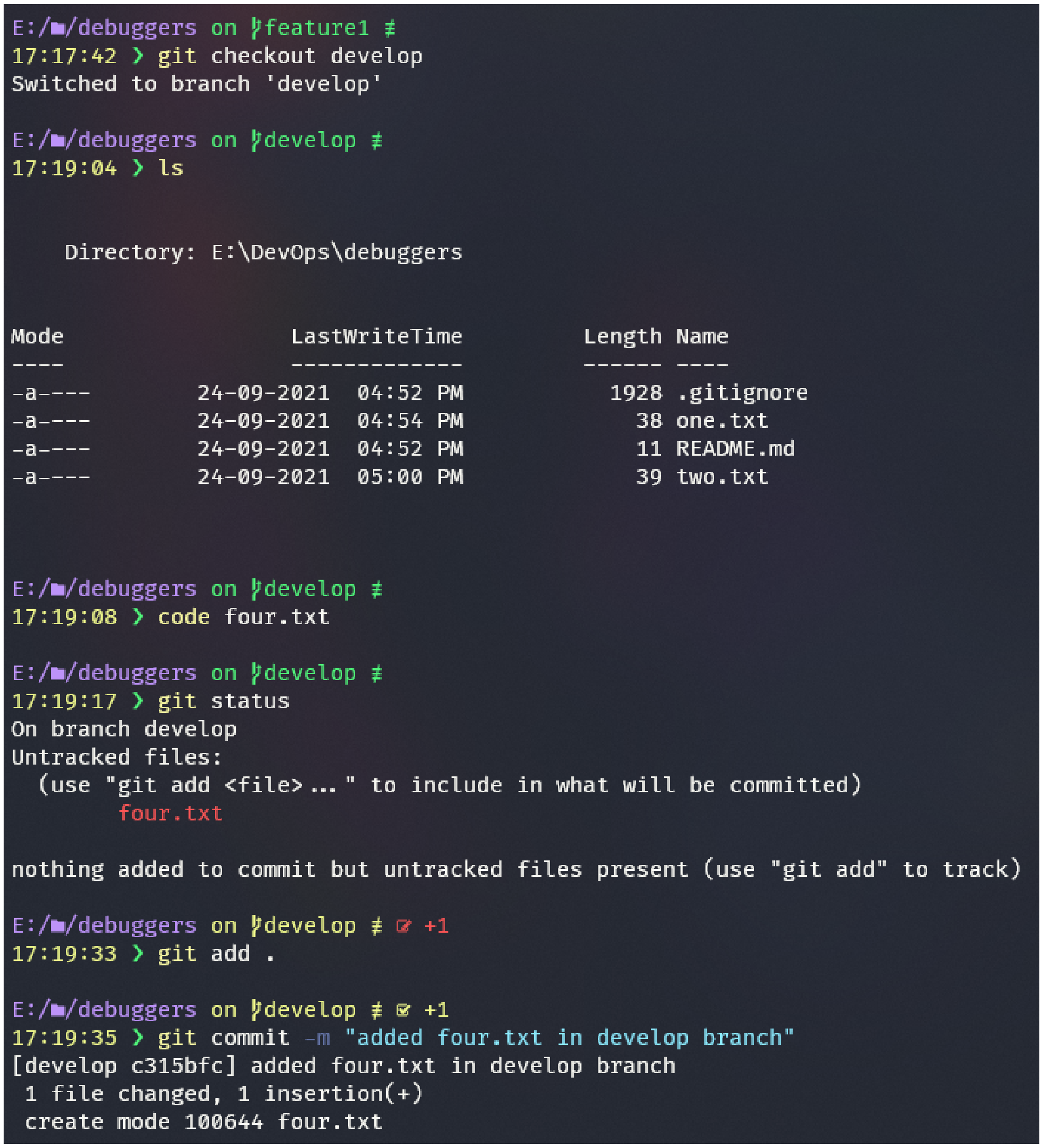
9. Contents of feature1 branch are



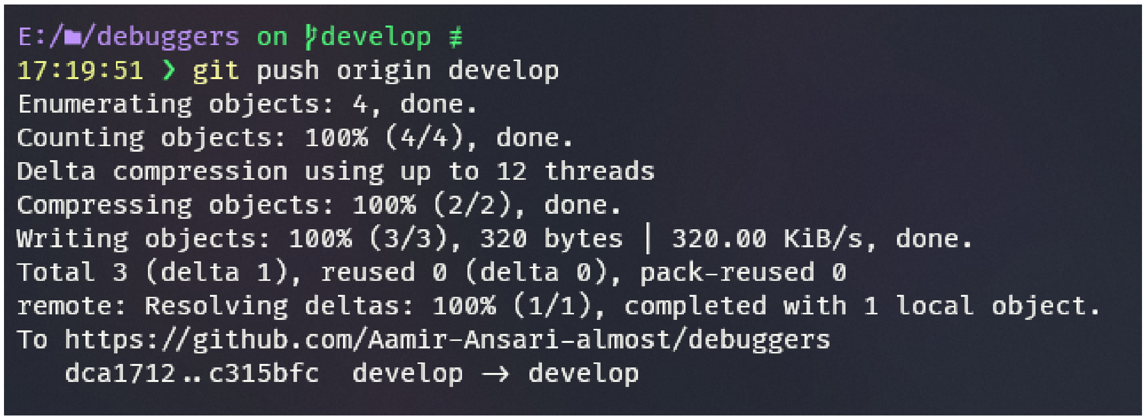
```
E:/■/debuggers on ⎇feature1 ≢
17:16:57 ❯ ls


    Directory: E:\DevOps\debuggers


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----        24-09-2021   04:52 PM          1928 .gitignore
-a----        24-09-2021   04:54 PM            38 one.txt
-a----        24-09-2021   04:52 PM            11 README.md
-a----        24-09-2021   05:16 PM            42 three.txt
-a----        24-09-2021   05:00 PM            39 two.txt
```

10. Let’s say, we wanted to add some files which are important to the project, so we want to add them to the develop branch. To do this, we first checkout to `develop` branch and

make sure that there will not be any conflicts by viewing the file, then push our newly added file

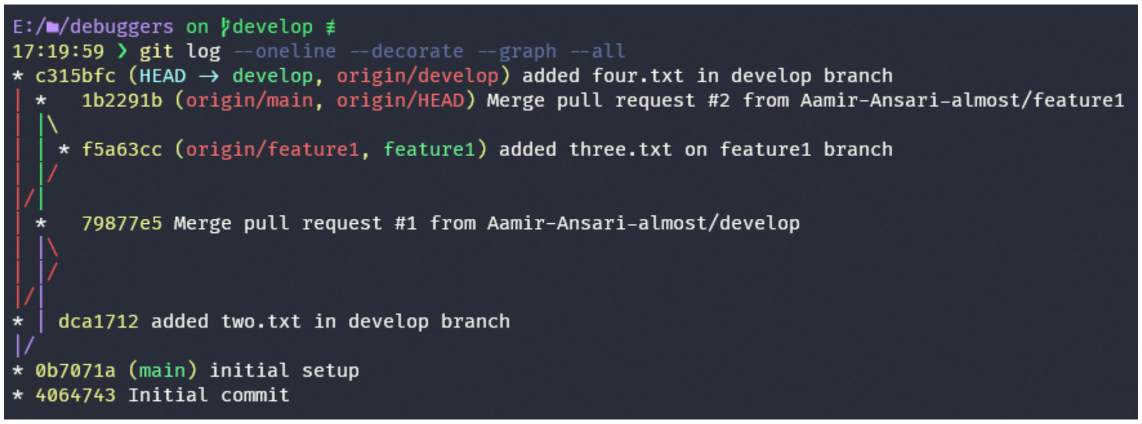


```
E:/■/debuggers on ᚹfeature1 ≢
17:17:42 ❯ git checkout develop
Switched to branch 'develop'

E:/■/debuggers on ᚹdevelop ≢
17:19:04 ❯ ls


    Directory: E:\DevOps\debuggers


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----        24-09-2021  04:52 PM           1928 .gitignore
-a----        24-09-2021  04:54 PM             38 one.txt
-a----        24-09-2021  04:52 PM             11 README.md
-a----        24-09-2021  05:00 PM             39 two.txt



E:/■/debuggers on ᚹdevelop ≢
17:19:08 ❯ code four.txt

E:/■/debuggers on ᚹdevelop ≢
17:19:17 ❯ git status
On branch develop
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        four.txt

nothing added to commit but untracked files present (use "git add" to track)

E:/■/debuggers on ᚹdevelop ≢ ☑ +1
17:19:33 ❯ git add .

E:/■/debuggers on ᚹdevelop ≢ ☑ +1
17:19:35 ❯ git commit -m "added four.txt in develop branch"
[develop c315bfc] added four.txt in develop branch
 1 file changed, 1 insertion(+)
 create mode 100644 four.txt
```



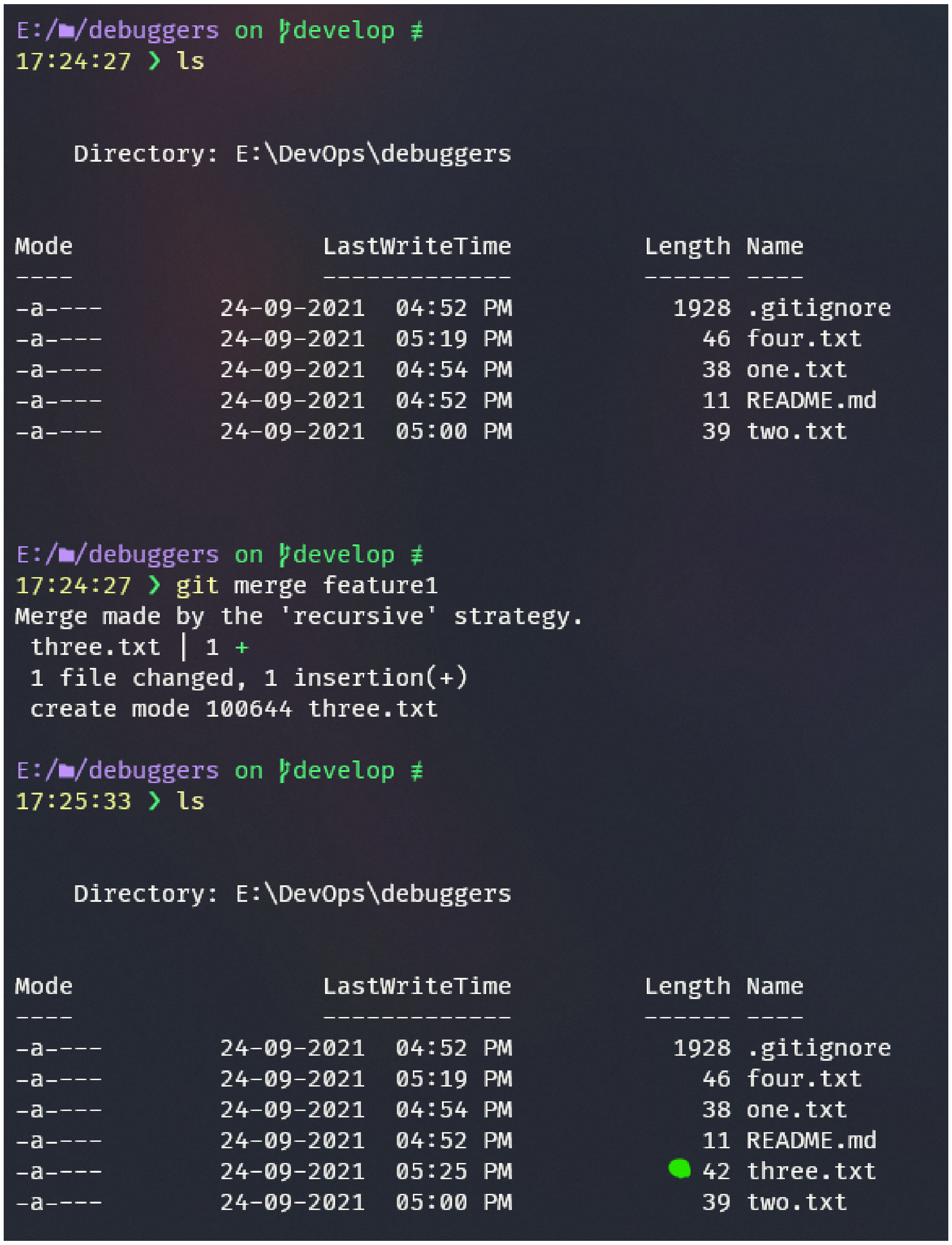
```
E:/■/debuggers on ᚠdevelop ≢
17:19:51 ❯ git push origin develop
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 12 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 320 bytes | 320.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To https://github.com/Aamir-Ansari-almost/debuggers
   dca1712..c315bfc  develop → develop
```

11. From the logs, we can see that main is pointing to the first commit, while `feature1` and `develop` branch are on their different commits



```
E:/■/debuggers on ᚠdevelop ≢
17:19:59 ❯ git log --oneline --decorate --graph --all
* c315bfc (HEAD → develop, origin/develop) added four.txt in develop branch
| *   1b2291b (origin/main, origin/HEAD) Merge pull request #2 from Aamir-Ansari-almost/feature1
| |\
| | * f5a63cc (origin/feature1, feature1) added three.txt on feature1 branch
| |/
|/|
| *   79877e5 Merge pull request #1 from Aamir-Ansari-almost/develop
| |\
| |/
|/|
* | dca1712 added two.txt in develop branch
|/
* 0b7071a (main) initial setup
* 4064743 Initial commit
```

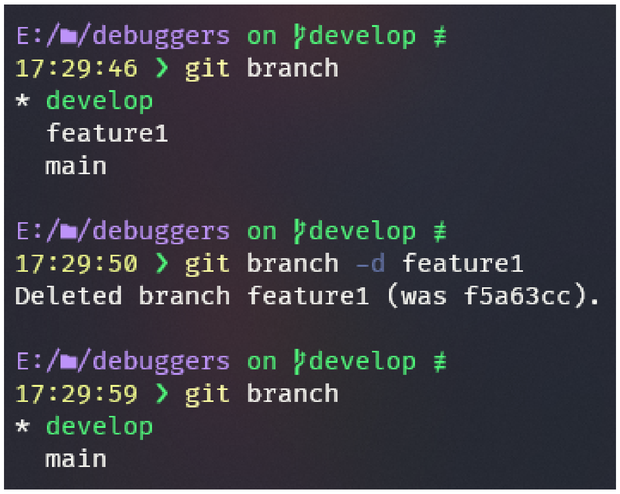
12. Our work related to feature on is done, so now we can merge `feature1` with `develop`
    a. *git merge <branch name>*



```
E:/■/debuggers on ᚹdevelop ≢
17:24:27 ❯ ls


    Directory: E:\DevOps\debuggers


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----        24-09-2021  04:52 PM           1928 .gitignore
-a----        24-09-2021  05:19 PM             46 four.txt
-a----        24-09-2021  04:54 PM             38 one.txt
-a----        24-09-2021  04:52 PM             11 README.md
-a----        24-09-2021  05:00 PM             39 two.txt



E:/■/debuggers on ᚹdevelop ≢
17:24:27 ❯ git merge feature1
Merge made by the 'recursive' strategy.
 three.txt | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 three.txt

E:/■/debuggers on ᚹdevelop ≢
17:25:33 ❯ ls


    Directory: E:\DevOps\debuggers


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----        24-09-2021  04:52 PM           1928 .gitignore
-a----        24-09-2021  05:19 PM             46 four.txt
-a----        24-09-2021  04:54 PM             38 one.txt
-a----        24-09-2021  04:52 PM             11 README.md
-a----        24-09-2021  05:25 PM           ● 42 three.txt
-a----        24-09-2021  05:00 PM             39 two.txt
```

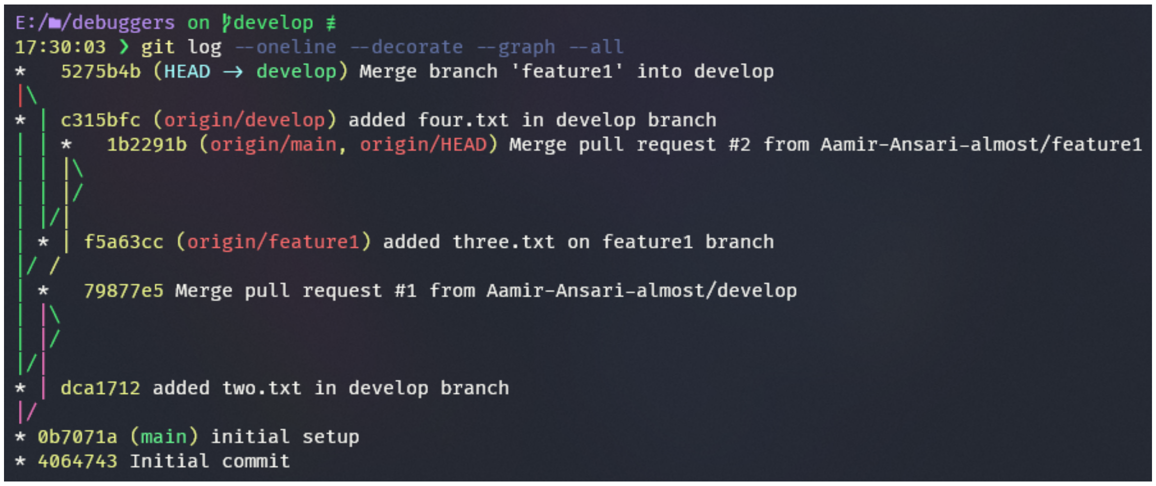
We can see that three.txt is there in develop branch after merging
And since we worked on different files, there was no merge conflicts

13. Now that we have merged feature1 branch, we do not need it anymore and we can delete it, use the following command
    a. *git branch -d <branch name>*



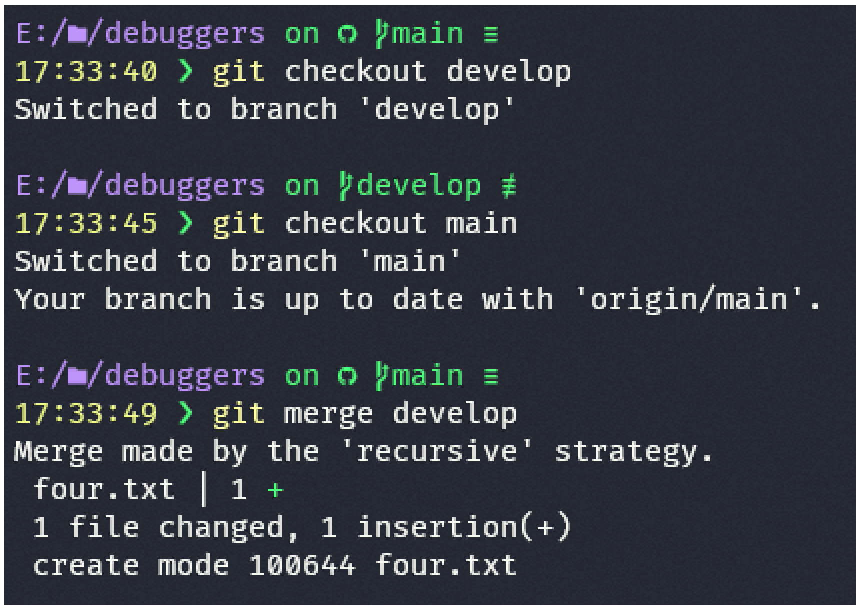
```
E:/▰/debuggers on ⎇develop ≢
17:29:46 ❯ git branch
* develop
  feature1
  main

E:/▰/debuggers on ⎇develop ≢
17:29:50 ❯ git branch -d feature1
Deleted branch feature1 (was f5a63cc).

E:/▰/debuggers on ⎇develop ≢
17:29:59 ❯ git branch
* develop
  main
```

14. To visualise this merging and deleting of branches, we see the logs



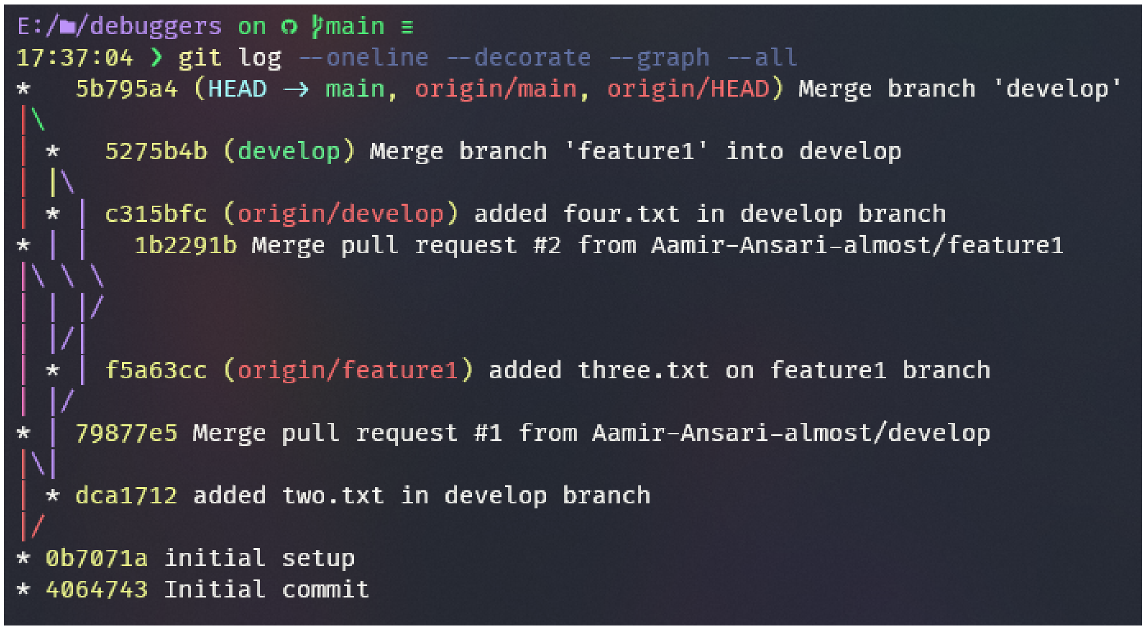
```
E:/▰/debuggers on ⎇develop ≢
17:30:03 ❯ git log --oneline --decorate --graph --all
*   5275b4b (HEAD -> develop) Merge branch 'feature1' into develop
|\
* | c315bfc (origin/develop) added four.txt in develop branch
| | *   1b2291b (origin/main, origin/HEAD) Merge pull request #2 from Aamir-Ansari-almost/feature1
| | |\
| | |/
| |/|
| * | f5a63cc (origin/feature1) added three.txt on feature1 branch
|/ /
| *   79877e5 Merge pull request #1 from Aamir-Ansari-almost/develop
| |\
| |/
|/|
* | dca1712 added two.txt in develop branch
|/
* 0b7071a (main) initial setup
* 4064743 Initial commit
```

15. When we verify that our develop branch is stable, and can be build for production, we merge it with the main branch



```
E:/▬/debuggers on ○ main ≡
17:33:40 ❯ git checkout develop
Switched to branch 'develop'

E:/▬/debuggers on develop ≢
17:33:45 ❯ git checkout main
Switched to branch 'main'
Your branch is up to date with 'origin/main'.

E:/▬/debuggers on ○ main ≡
17:33:49 ❯ git merge develop
Merge made by the 'recursive' strategy.
 four.txt | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 four.txt
```

16. Now our main and develop branch are in sync



```
E:/▬/debuggers on ○ main ≡
17:37:04 ❯ git log --oneline --decorate --graph --all
*   5b795a4 (HEAD → main, origin/main, origin/HEAD) Merge branch 'develop'
|\
| *   5275b4b (develop) Merge branch 'feature1' into develop
| |\
| * | c315bfc (origin/develop) added four.txt in develop branch
* | |   1b2291b Merge pull request #2 from Aamir-Ansari-almost/feature1
|\ \ \
| | |/
| |/|
| * | f5a63cc (origin/feature1) added three.txt on feature1 branch
| |/
* | 79877e5 Merge pull request #1 from Aamir-Ansari-almost/develop
|\|
| * dca1712 added two.txt in develop branch
|/
* 0b7071a initial setup
* 4064743 Initial commit
```

This was the demonstration of basic workflow on git

Conclusion

Thus, we have learnt how branching and merging works in Git along with the commands which enables us to use these features. We also learnt various types of workflows in Github along with various scenarios where they can be used.