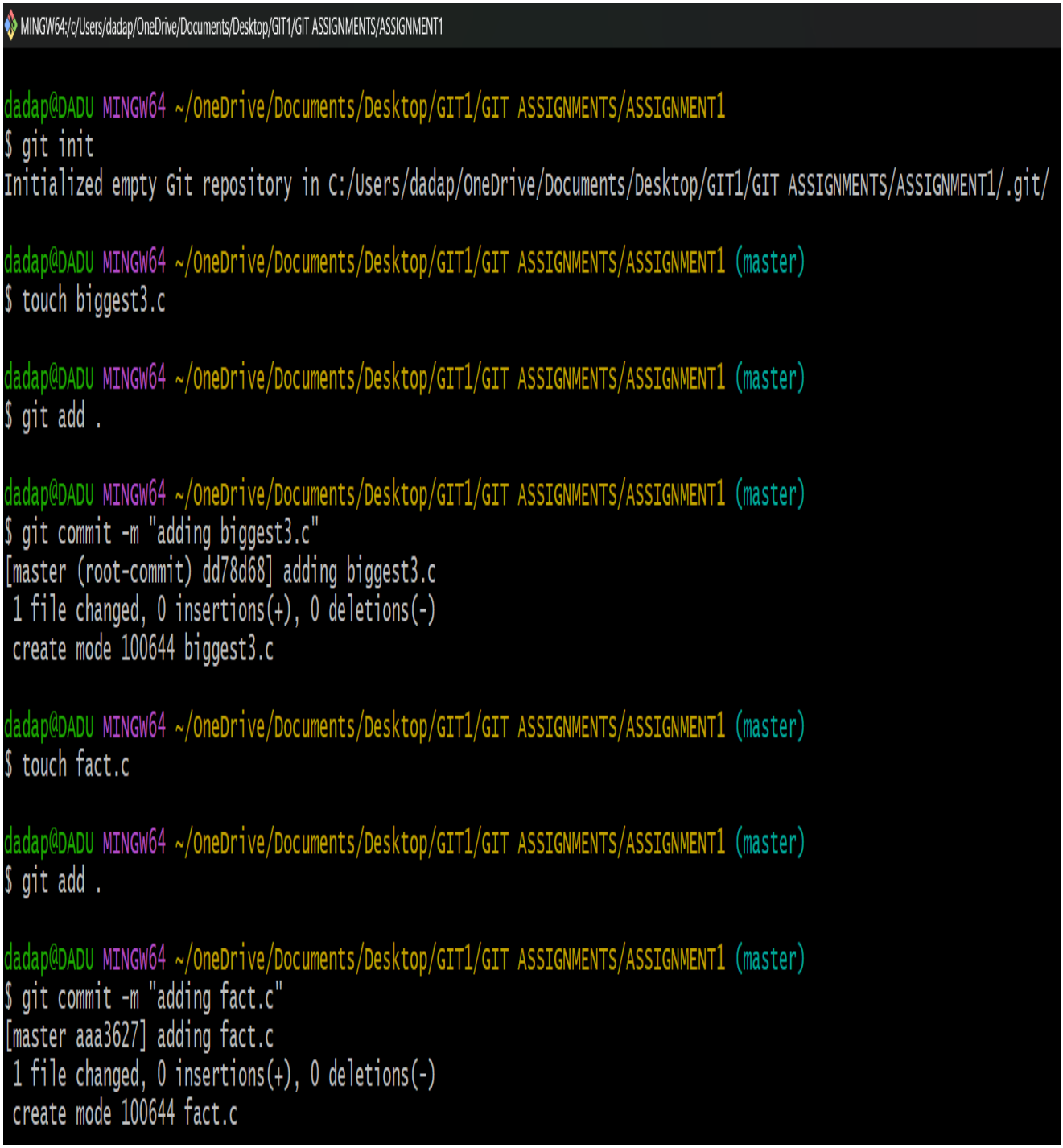
Name: Dadapeer

Git Assignment(All git commands & Assignments)

Assignment 1:

try to create master branch in master branch create a file biggest3.c in master branch create a file fact.c and give tag name under master you create one more brach --> branch 1 in branch 1 create a file with code of reverse.c again create a branch 2 in master branch in branch 2 create a file fibbonacci.called merge branch 1 and branch 2 in master branch



```
MINGW64:/c/Users/dadap/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT1
$ touch biggest3.c

dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT1 (master)
$ git add .

dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT1 (master)
$ git commit -m "adding biggest3.c"
[master (root-commit) dd78d68] adding biggest3.c
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 biggest3.c

dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT1 (master)
$ touch fact.c

dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT1 (master)
$ git add .

dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT1 (master)
$ git commit -m "adding fact.c"
[master aaa3627] adding fact.c
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 fact.c

dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT1 (master)
$ git log
commit aaa362755b2853b1d55529fd29f4dc8e5295b07f (HEAD -> master)
Author: Dadu <dadapeerdpr@gmail.com>
Date:   Wed Jul 31 14:56:00 2024 +0530

    adding fact.c

commit dd78d680bf3384862cd371034d8d02b0da607de9
Author: Dadu <dadapeerdpr@gmail.com>
Date:   Wed Jul 31 14:55:21 2024 +0530

    adding biggest3.c

dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT1 (master)
$
```

```
dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT1 (master)
$ git tag v1.0.0

dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT1 (master)
$ git log
commit aaa362755b2853b1d55529fd29f4dc8e5295b07f (HEAD -> master, tag: v1.0.0)
Author: Dadu <dadapeerdpr@gmail.com>
Date:   Wed Jul 31 14:56:00 2024 +0530

    adding fact.c

commit dd78d680bf3384862cd371034d8d02b0da607de9
Author: Dadu <dadapeerdpr@gmail.com>
Date:   Wed Jul 31 14:55:21 2024 +0530

    adding biggest3.c

dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT1 (master)
$ git status
On branch master
nothing to commit, working tree clean

dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT1 (master)
$
```

```
dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT1 (master)
$ git status
On branch master
nothing to commit, working tree clean

dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT1 (master)
$ git branch branch1

dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT1 (master)
$ git checkout branch1
Switched to branch 'branch1'

dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT1 (branch1)
$ vi reverse.c

dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT1 (branch1)
$ git add .
warning: in the working copy of 'reverse.c', LF will be replaced by CRLF the next time Git touches it

dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT1 (branch1)
$ git commit -m "adding reverse.c"
[branch1 07baf30] adding reverse.c
 1 file changed, 1 insertion(+)
 create mode 100644 reverse.c

dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT1 (branch1)
$
```

```
MINGW64:/c/Users/dadap/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT1

dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT1 (master)
$ git checkout branch2
Switched to branch 'branch2'

dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT1 (branch2)
$ touch fibonacci.called

dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT1 (branch2)
$ git add.
git: 'add.' is not a git command. See 'git --help'.

The most similar command is
        add

dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT1 (branch2)
$ git add .

dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT1 (branch2)
$ git commit -m "adding fibonacci.called"
[branch2 21b5e8c] adding fibonacci.called
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 fibonacci.called

dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT1 (branch2)
$ ls
biggest3.c  fact.c  fibonacci.called

dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT1 (branch2)
$
```

```
MINGW64:/c/Users/dadap/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT1

dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT1 (branch2)
$ git checkout master
Switched to branch 'master'

dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT1 (master)
$ git merge branch1
Merge made by the 'ort' strategy.
 reverse.c | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 reverse.c

dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT1 (master)
$ git merge branch2
Merge made by the 'ort' strategy.
```

```
MINGW64:/c/Users/dadap/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT1
$ git log
commit fb30546d550fd1c3e9a5cffbf3588d6e54687f2c (HEAD -> master)
Merge: 1aba35a 21b5e8c
Author: Dadu <dadapeerdpr@gmail.com>
Date:   Wed Jul 31 15:23:20 2024 +0530

    Merge branch 'branch2'

commit 1aba35a11a20397b015e32208935d579e0edf56e
Merge: 0d87257 07baf30
Author: Dadu <dadapeerdpr@gmail.com>
Date:   Wed Jul 31 15:22:48 2024 +0530

    Merge branch 'branch1'

commit 21b5e8c8e9e797915756ac483b0411c8ad1314c6 (branch2)
Author: Dadu <dadapeerdpr@gmail.com>
Date:   Wed Jul 31 15:10:16 2024 +0530

    adding fibonacci.called

commit 0d87257591f41bb22d7dd7c78749682f755d1419
Author: Dadu <dadapeerdpr@gmail.com>
Date:   Wed Jul 31 15:07:21 2024 +0530

    adding fibonacci.called

commit 07baf3094ae39a8023a577f163df8d0445a2d5d5 (branch1)
Author: Dadu <dadapeerdpr@gmail.com>
Date:   Wed Jul 31 15:04:31 2024 +0530

    adding reverse.c

commit aaa362755b2853b1d55529fd29f4dc8e5295b07f (tag: v1.0.0)
Author: Dadu <dadapeerdpr@gmail.com>
Date:   Wed Jul 31 14:56:00 2024 +0530

    adding fact.c
```

Assignment 2:

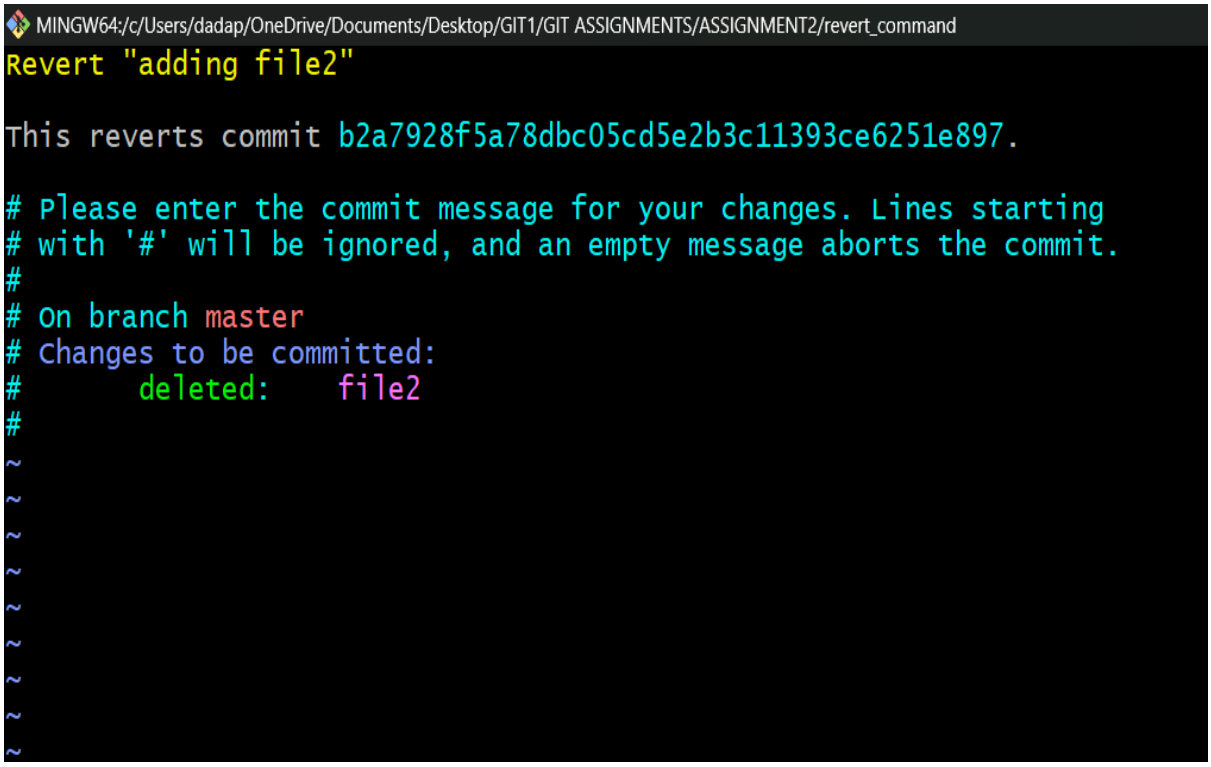
git revert - we can go back to the workspace after commiting,but commit history will be stored

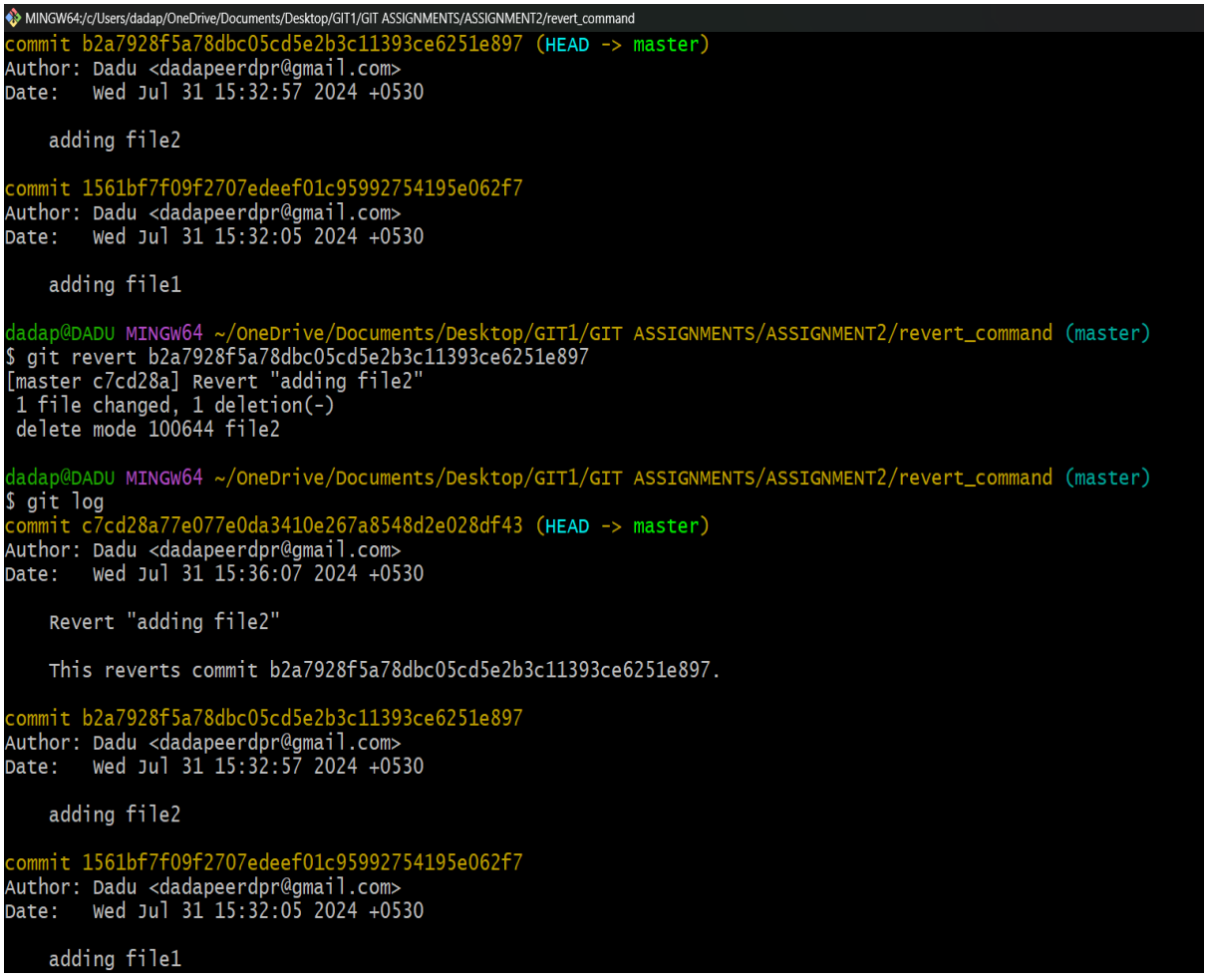
git reset - we can go back to the workspace after commiting,but commit history is removed

git reset - commited changes are reset,but commit history is removed

git restore -- to unstage a file

git revert: file2 added and deleted using git revert

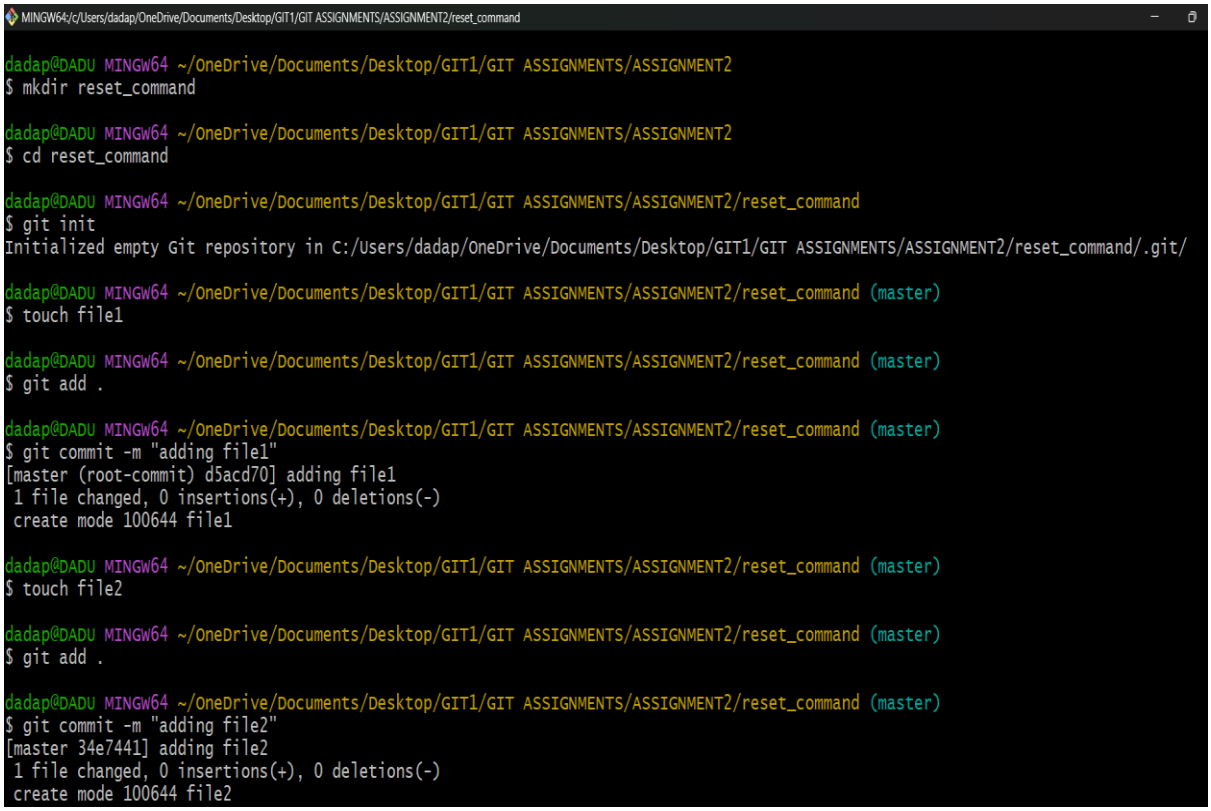

MINGW64:/c/Users/dadap/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT2/revert_command
Revert "adding file2"
This reverts commit b2a7928f5a78dbc05cd5e2b3c11393ce6251e897.
Please enter the commit message for your changes. Lines starting
with '#' will be ignored, and an empty message aborts the commit.
#
On branch master
Changes to be committed:
#	deleted: file2
#

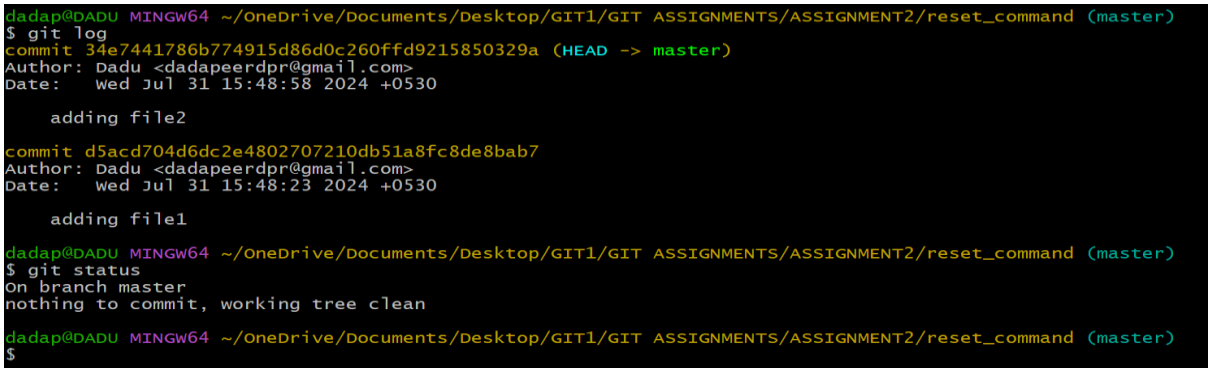

MINGW64:/c/Users/dadap/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT2/revert_command
commit b2a7928f5a78dbc05cd5e2b3c11393ce6251e897 (HEAD -> master)
Author: Dadu <dadapeerdpr@gmail.com>
Date: Wed Jul 31 15:32:57 2024 +0530
adding file2
commit 1561bf7f09f2707edeef01c95992754195e062f7
Author: Dadu <dadapeerdpr@gmail.com>
Date: Wed Jul 31 15:32:05 2024 +0530
adding file1
dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT2/revert_command (master)
$ git revert b2a7928f5a78dbc05cd5e2b3c11393ce6251e897
[master c7cd28a] Revert "adding file2"
1 file changed, 1 deletion(-)
delete mode 100644 file2
dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT2/revert_command (master)
$ git log
commit c7cd28a77e077e0da3410e267a8548d2e028df43 (HEAD -> master)
Author: Dadu <dadapeerdpr@gmail.com>
Date: Wed Jul 31 15:36:07 2024 +0530
Revert "adding file2"
This reverts commit b2a7928f5a78dbc05cd5e2b3c11393ce6251e897.
commit b2a7928f5a78dbc05cd5e2b3c11393ce6251e897
Author: Dadu <dadapeerdpr@gmail.com>
Date: Wed Jul 31 15:32:57 2024 +0530
adding file2
commit 1561bf7f09f2707edeef01c95992754195e062f7
Author: Dadu <dadapeerdpr@gmail.com>
Date: Wed Jul 31 15:32:05 2024 +0530
adding file1

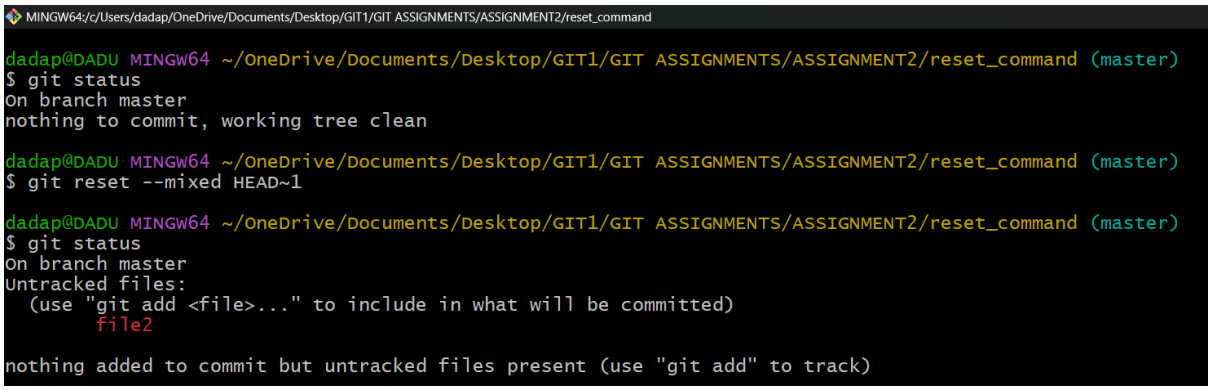
Git reset:

1. --mixed Mode

The --mixed option moves the HEAD to the specified commit and resets the index, but not the working directory. This means that the changes from the commit will appear as unstaged changes.

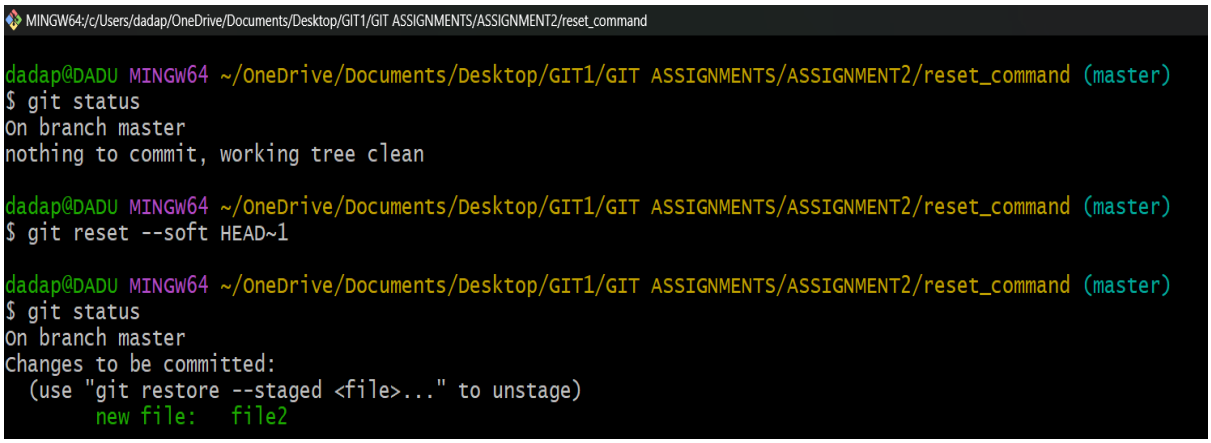






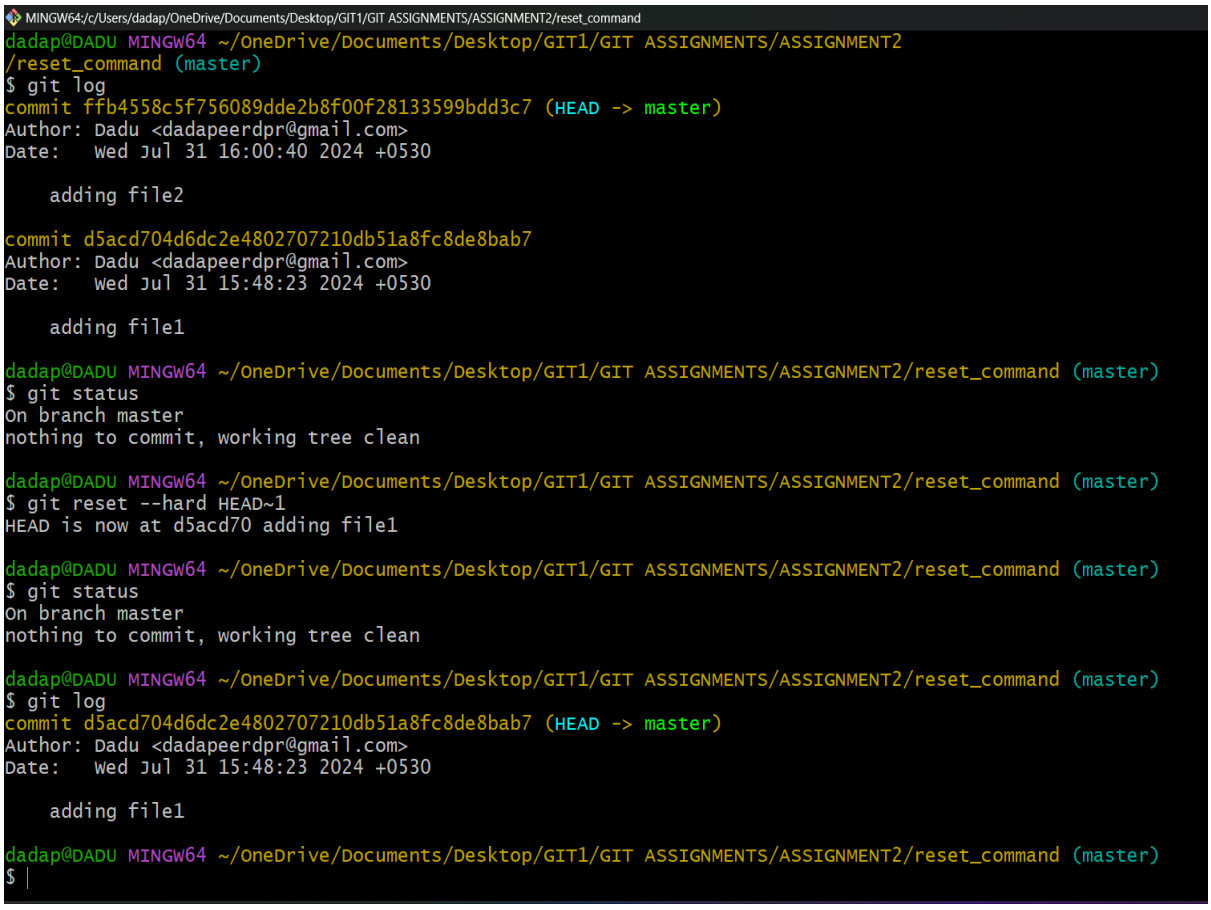
2. --soft Mode

The --soft option moves the HEAD to the specified commit, but leaves the index and working directory unchanged. This means that all changes that were in the commit will appear as staged for commit.

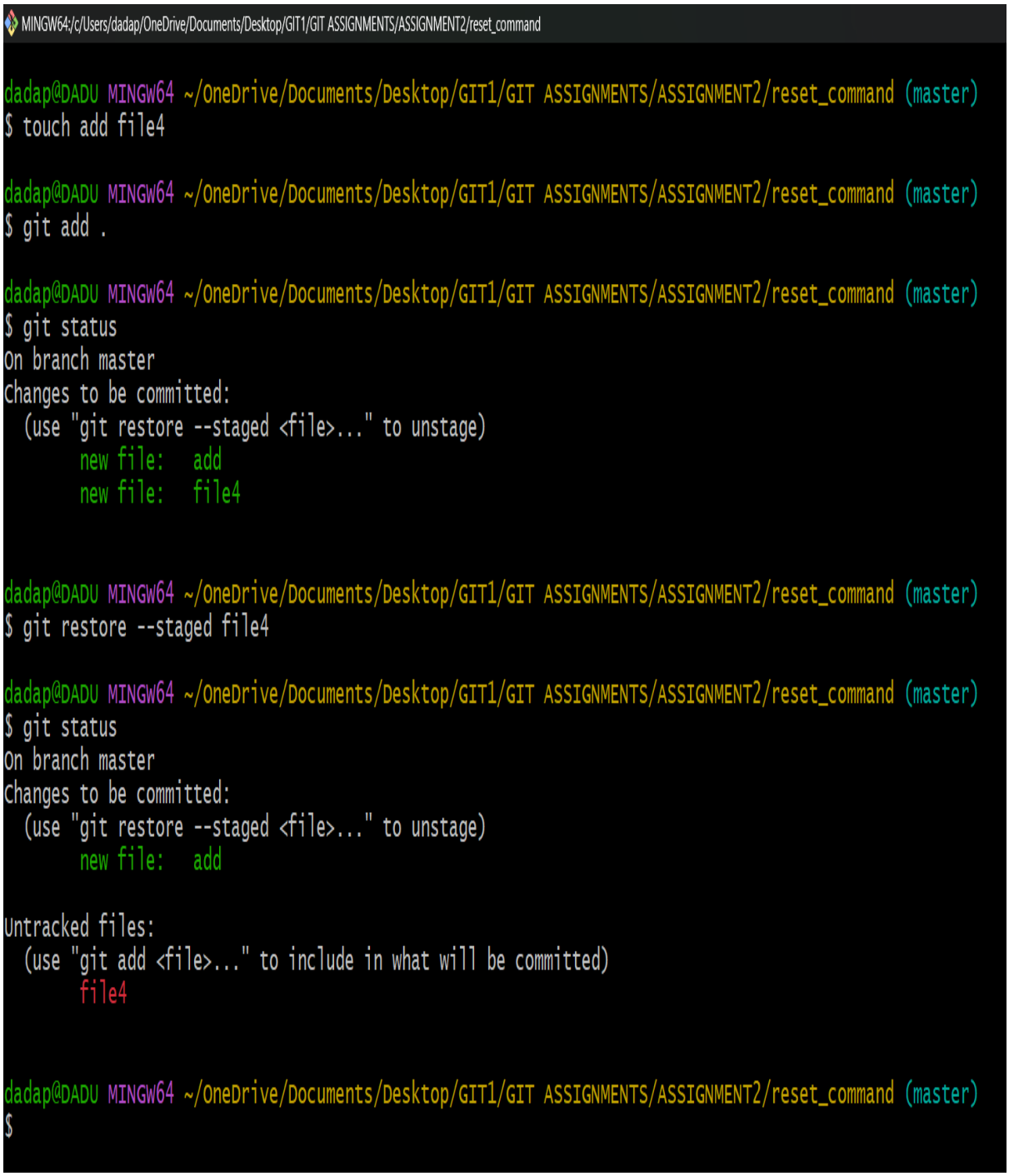


3. --hard Mode

The --hard option resets the HEAD, index, and working directory to the specified commit. This means all changes will be lost, and the working directory will be clean.



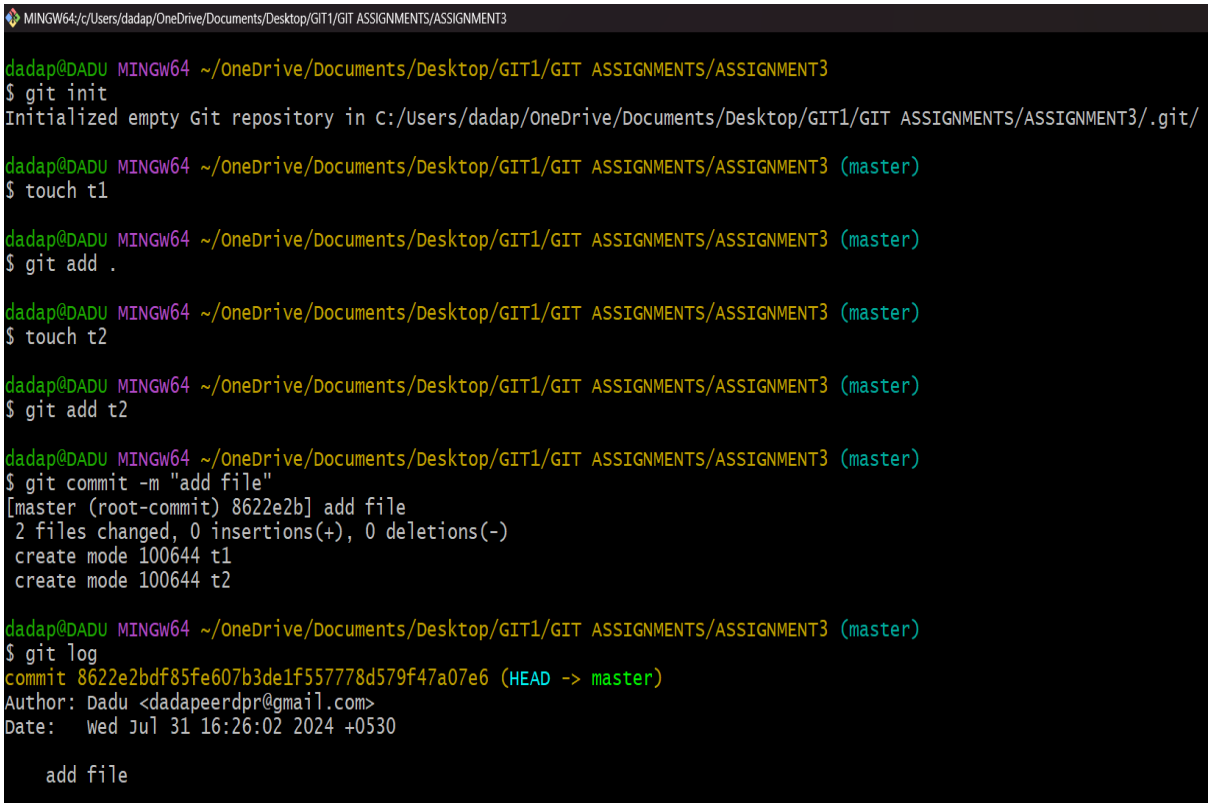
Git restore: The git restore command is used to restore working tree files and staging area entries. It can be used to undo changes in the working directory or to unstage files

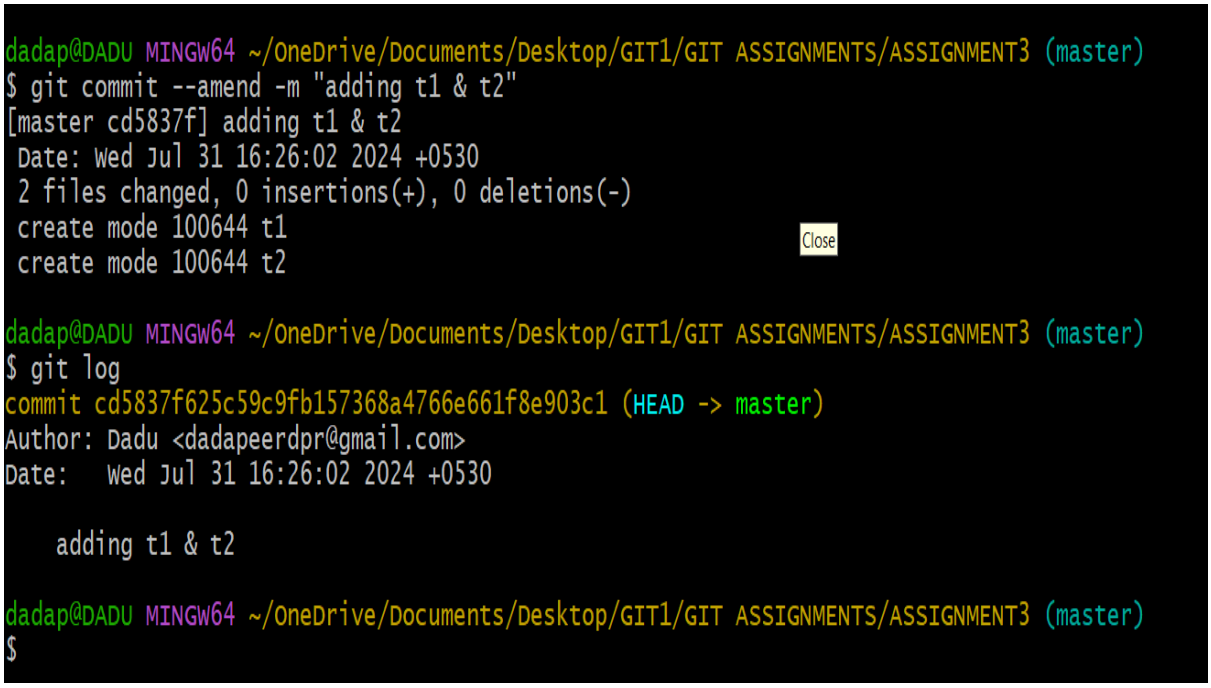


Assignment3 :

==> **git amend** : To modify your last commit.you can change your log message and files that appear in the commit. the old commit is replaced by new commit.it means old commit will not be there.

--> it will help you edit the commit message





==> .**gitignore :** to ignore a file in the project

--> it will not allow the files to be commited.it will ignore those files.

Common uses for .gitignore:

1. Exclude sensitive information: Ignore files containing passwords, API keys, or other sensitive data.

2. Ignore build artifacts: Exclude compiled files, logs, or other generated files that don't need to be version-controlled.

3. Exclude operating system files: Ignore files created by your operating system, like .DS_Store or Thumbs.db.

4. Ignore editor or IDE settings: Exclude configuration files specific to your editor or IDE.

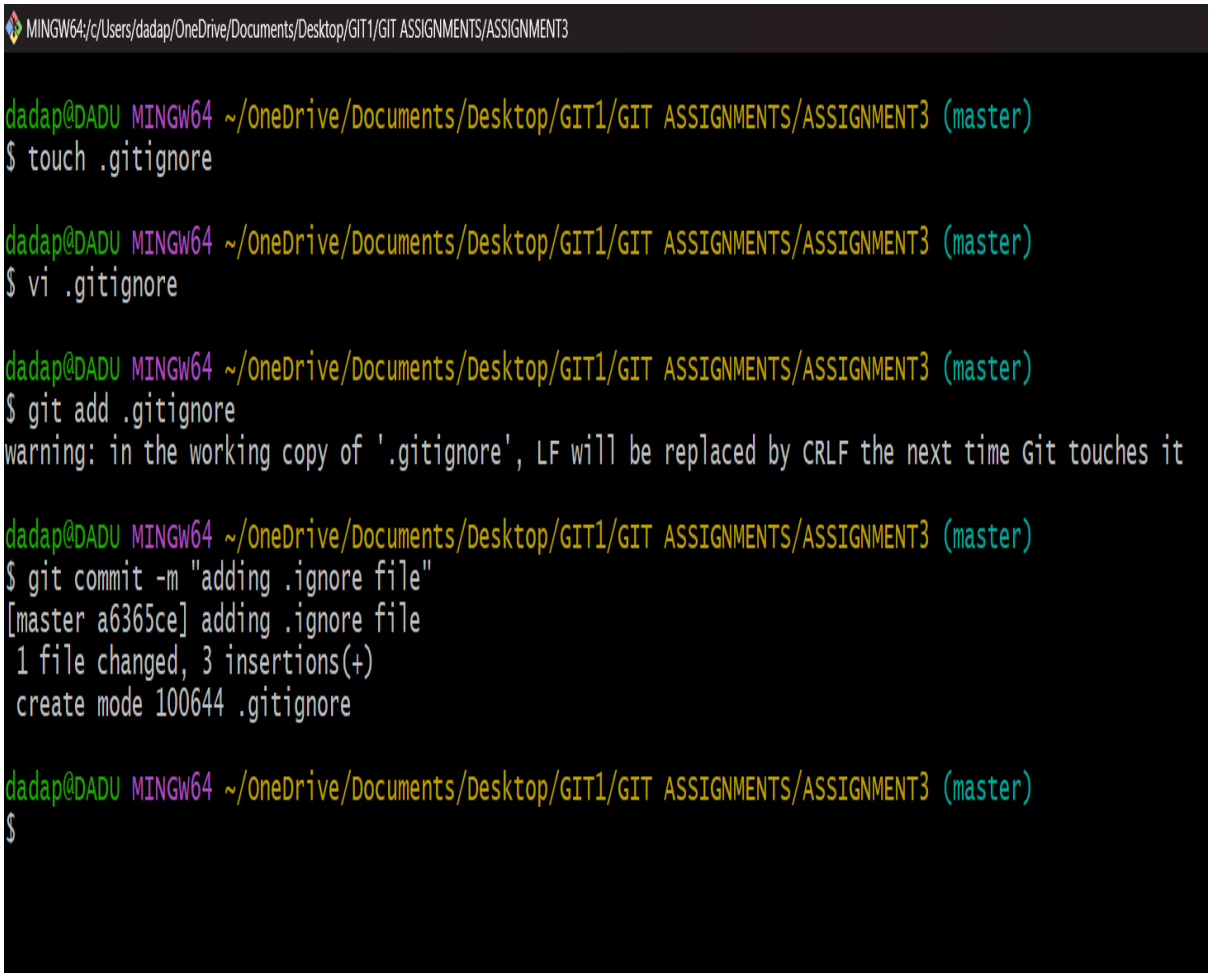
How to use .gitignore:

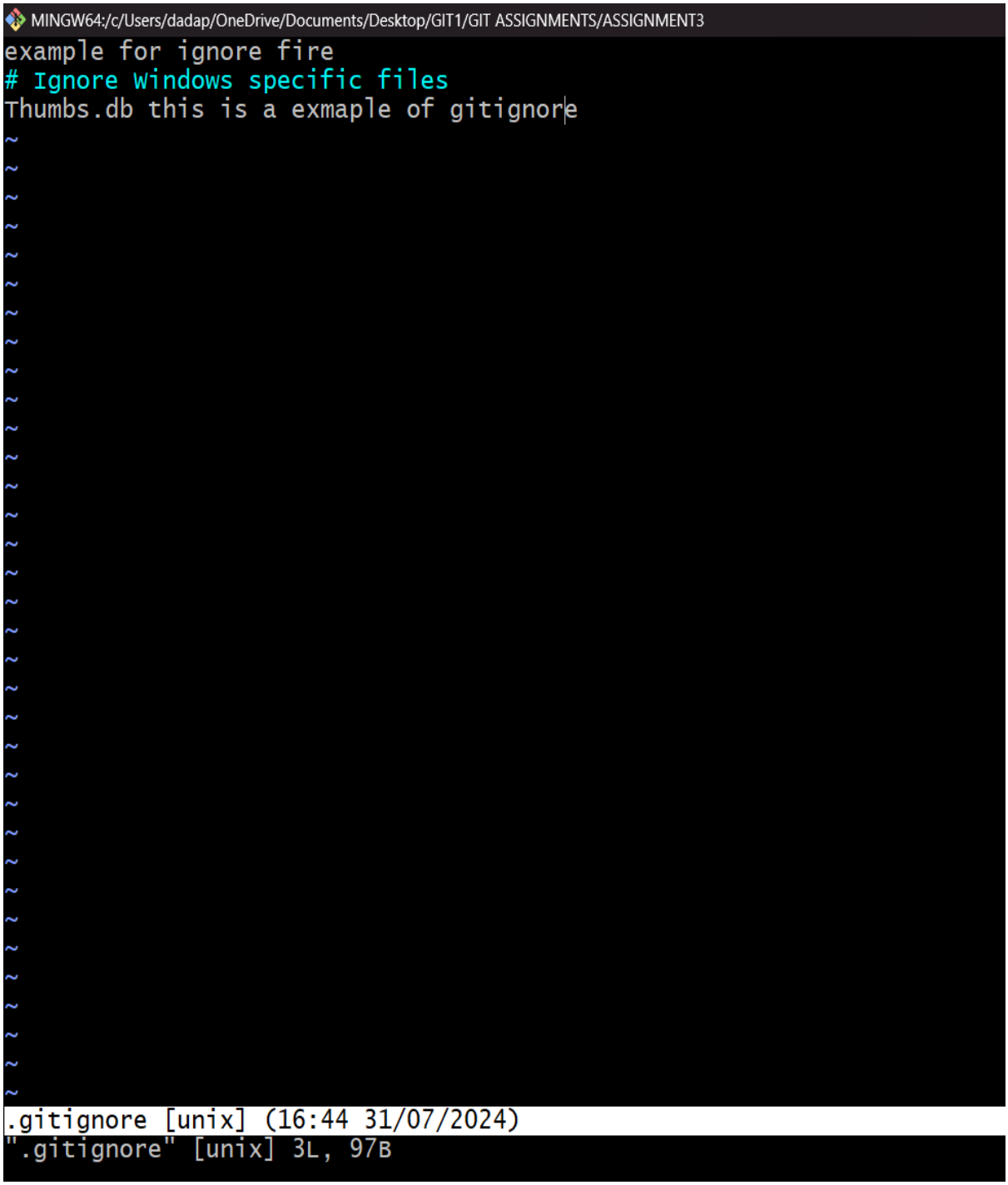
1. Create a .gitignore file in the root of your repository.

2. Add patterns or file names to ignore, one per line.

3. Use wildcards (*) to match multiple files or directories.

4. Use ! to negate a pattern and include a file or directory.




MINGW64:/c/Users/dadap/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT3
example for ignore fire
Ignore Windows specific files
Thumbs.db this is a exmaple of gitignore
.gitignore [unix] (16:44 31/07/2024)
".gitignore" [unix] 3L, 97B

Assignment4:

Git Diff: Git diff is a command used to show changes between commits, branches, or files in a Git repository. It displays the differences in a human-readable format, allowing you to review and understand the changes made.

Common uses of git diff:

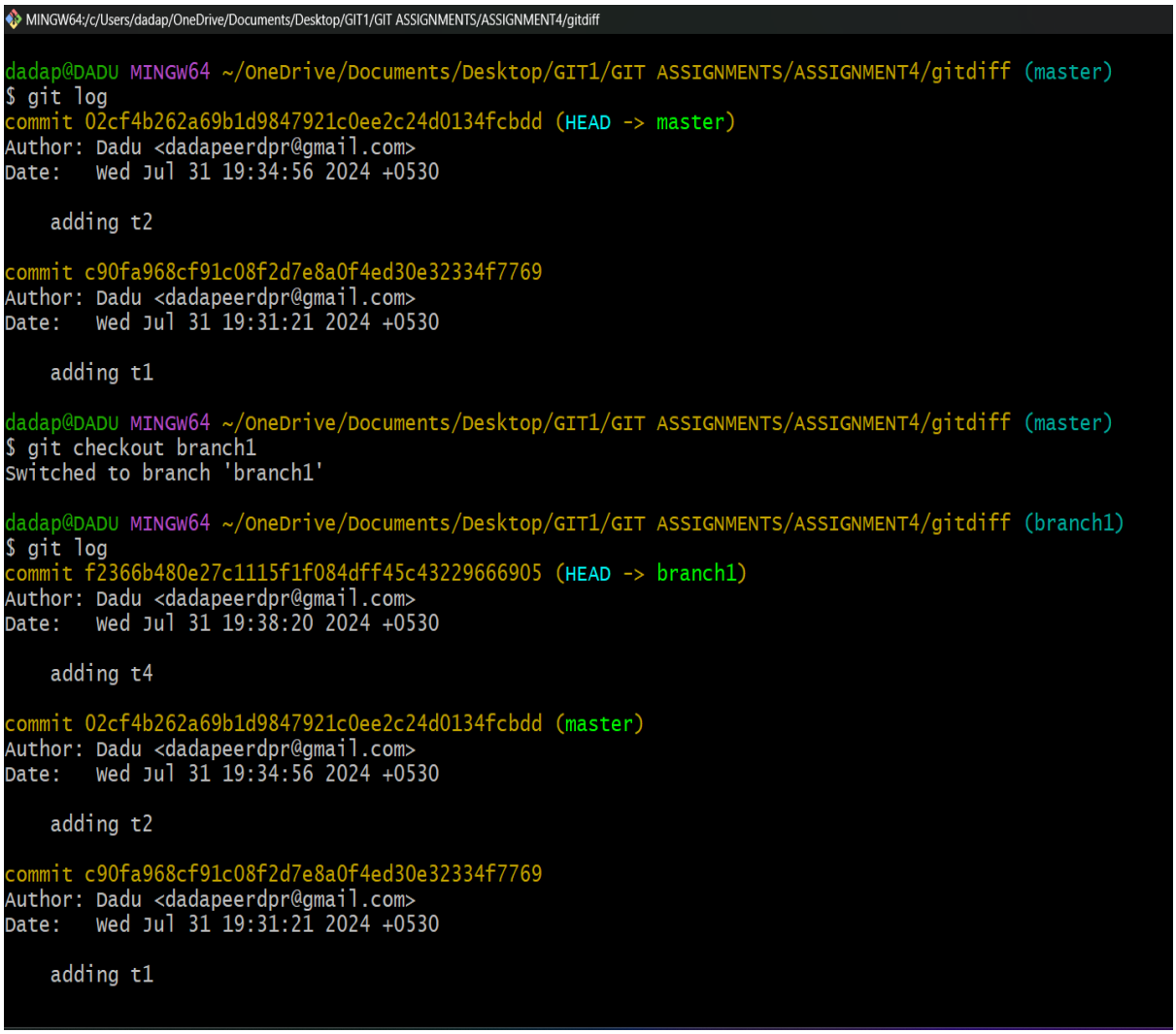
1. Compare changes between commits: git diff <commit1> <commit2> shows changes between two specific commits.

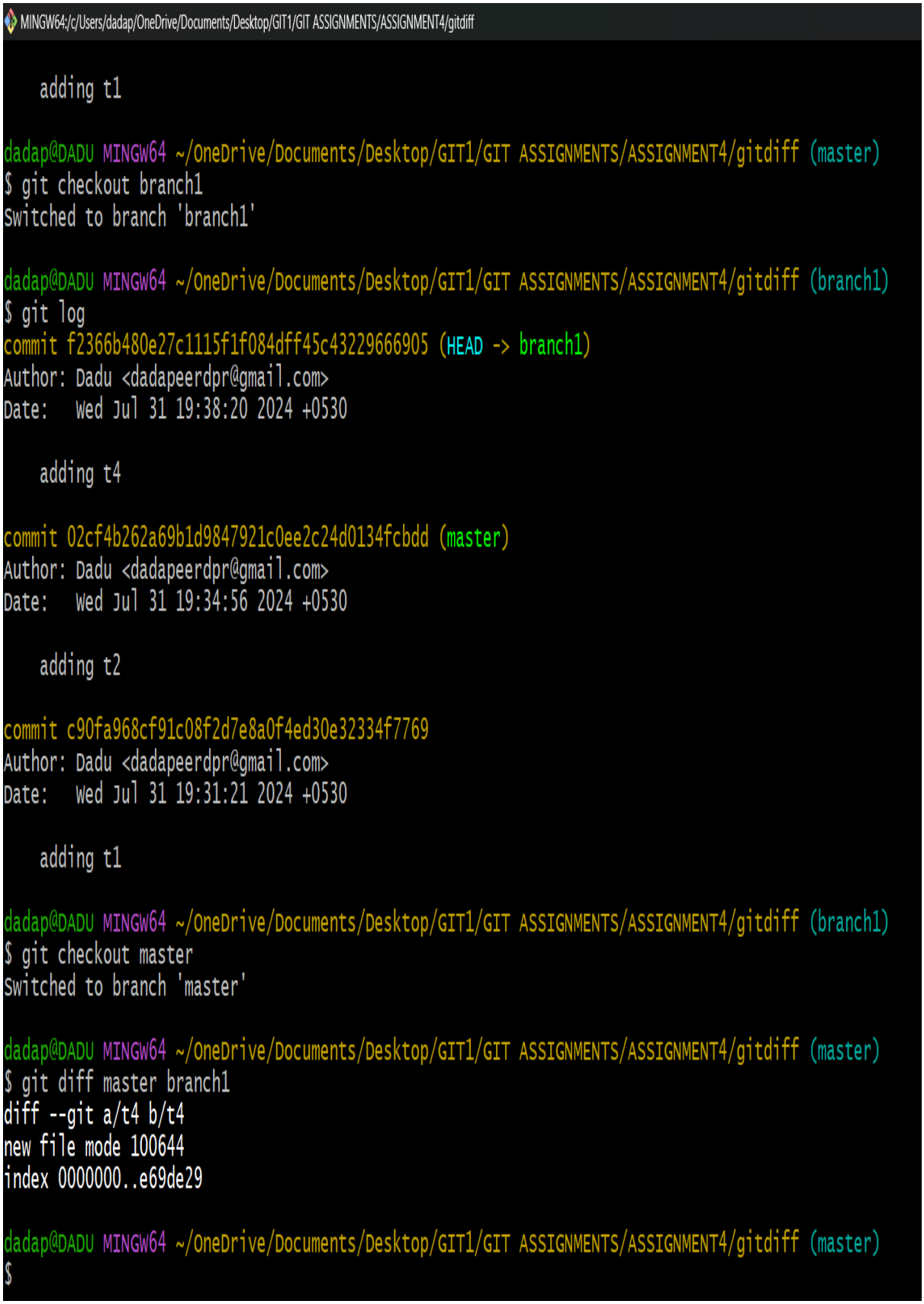
2. Compare changes between branches: git diff <branch1> <branch2> shows changes between two branches.

3. Compare changes in a file: git diff <file> shows changes made to a specific file.

4. Compare staged and unstaged changes: git diff (without arguments) shows changes between the staging area and the working directory.

5. Compare changes since last commit: git diff HEAD shows changes made since the last commit.




MINGW64:/c/Users/dadap/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT4/gitdiff
adding t1
dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT4/gitdiff (master)
$ git checkout branch1
Switched to branch 'branch1'
dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT4/gitdiff (branch1)
$ git log
commit f2366b480e27c1115f1f084dff45c43229666905 (HEAD -> branch1)
Author: Dadu <dadapeerdpr@gmail.com>
Date: Wed Jul 31 19:38:20 2024 +0530
adding t4
commit 02cf4b262a69b1d9847921c0ee2c24d0134fcbdd (master)
Author: Dadu <dadapeerdpr@gmail.com>
Date: Wed Jul 31 19:34:56 2024 +0530
adding t2
commit c90fa968cf91c08f2d7e8a0f4ed30e32334f7769
Author: Dadu <dadapeerdpr@gmail.com>
Date: Wed Jul 31 19:31:21 2024 +0530
adding t1
dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT4/gitdiff (branch1)
$ git checkout master
Switched to branch 'master'
dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT4/gitdiff (master)
$ git diff master branch1
diff --git a/t4 b/t4
new file mode 100644
index 0000000..e69de29
dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT4/gitdiff (master)
$

Git bisect: is a powerful command used to find the commit that introduced a bug or issue in your code. It uses a binary search algorithm to narrow down the possible commits until it finds the culprit.

Here's how to use git bisect:

1. Start the bisect process: git bisect start

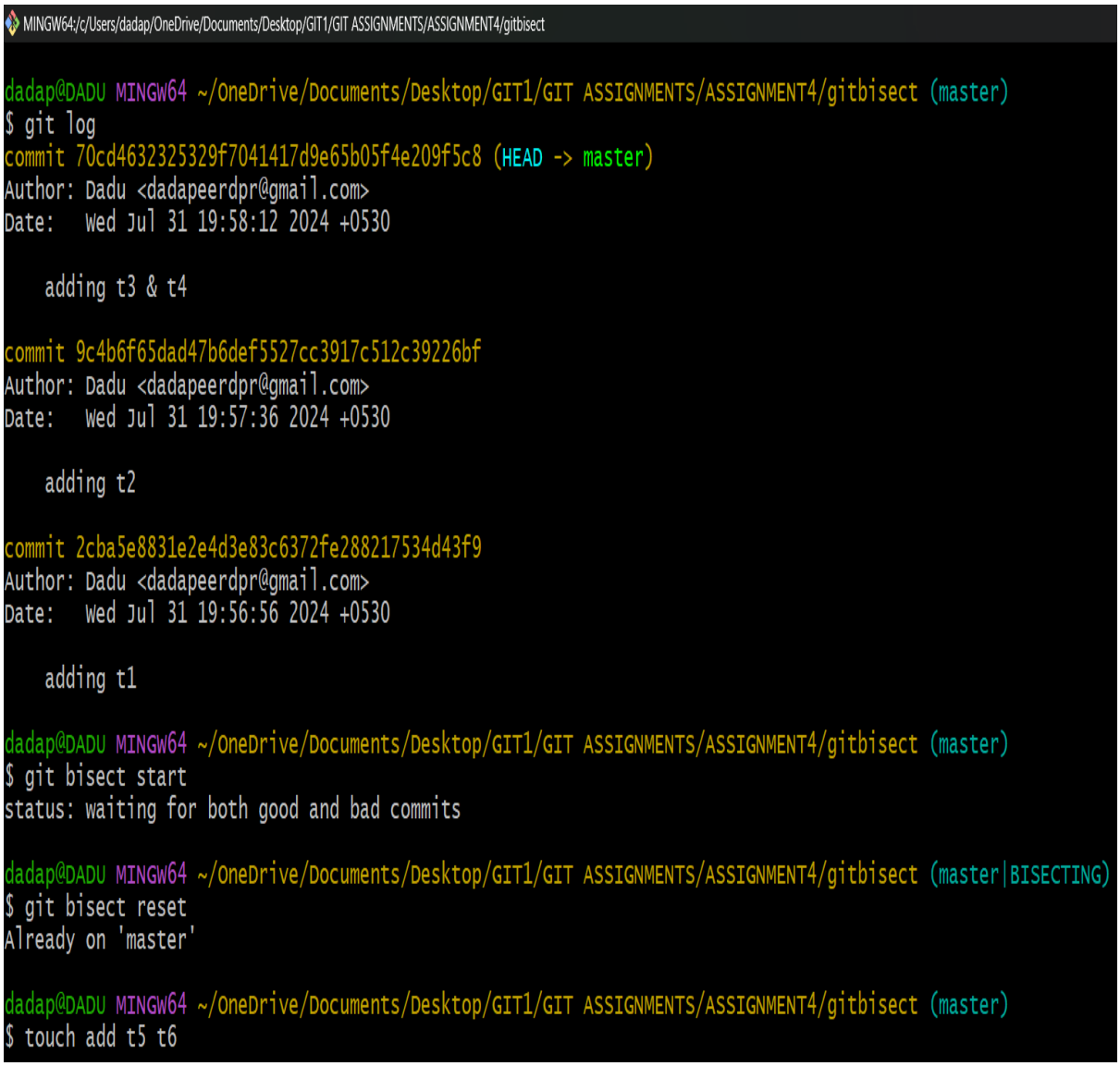
2. Mark the current commit as bad: git bisect bad

3. Mark a known good commit: git bisect good <commit>

4. Git will checkout a commit in the middle of the range.

5. Test the code and mark it as good or bad: git bisect good or git bisect bad

6. Repeat steps 4-5 until Git finds the bad commit.



```
MINGW64:/c/Users/dadap/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT4/gitbisect
dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT4/gitbisect (master)
$ git add .

dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT4/gitbisect (master)
$ git commit -m "adding t5 & t6"
[master 63315bd] adding t5 & t6
 2 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 t5
 create mode 100644 t6

dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT4/gitbisect (master)
$ git bisect start
status: waiting for both good and bad commits

dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT4/gitbisect (master|BISECTING)
$ git bisect bad 70cd4632325329f7041417d9e65b05f4e209f5c8
status: waiting for good commit(s), bad commit known

dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT4/gitbisect (master|BISECTING)
$ git bisect good 9c4b6f65dad47b6def5527cc3917c512c39226bf
70cd4632325329f7041417d9e65b05f4e209f5c8 is the first bad commit
commit 70cd4632325329f7041417d9e65b05f4e209f5c8
Author: Dadu <dadapeerdpr@gmail.com>
Date:   Wed Jul 31 19:58:12 2024 +0530

    adding t3 & t4

 t3 | 0
 t4 | 0
 2 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 t3
 create mode 100644 t4

dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT4/gitbisect (master|BISECTING)
$ git log
commit 63315bd78d50ce45c6cc882be7bd9823831f1916 (HEAD -> master)
Author: Dadu <dadapeerdpr@gmail.com>
Date:   Wed Jul 31 19:59:55 2024 +0530
```

```
dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT4/gitbisect (master|BISECTING)
$ git log
commit 63315bd78d50ce45c6cc882be7bd9823831f1916 (HEAD -> master)
Author: Dadu <dadapeerdpr@gmail.com>
Date:   Wed Jul 31 19:59:55 2024 +0530

    adding t5 & t6

commit 70cd4632325329f7041417d9e65b05f4e209f5c8
Author: Dadu <dadapeerdpr@gmail.com>
Date:   Wed Jul 31 19:58:12 2024 +0530

    adding t3 & t4

commit 9c4b6f65dad47b6def5527cc3917c512c39226bf
Author: Dadu <dadapeerdpr@gmail.com>
Date:   Wed Jul 31 19:57:36 2024 +0530

    adding t2

commit 2cba5e8831e2e4d3e83c6372fe288217534d43f9
Author: Dadu <dadapeerdpr@gmail.com>
Date:   Wed Jul 31 19:56:56 2024 +0530

    adding t1

dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT4/gitbisect (master|BISECTING)
$ git bisect reset
Already on 'master'

dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT4/gitbisect (master)
$
```

Git insta web: git insta web is a command that allows you to instantly visualize your Git repository as a web-based Git repository browser. It's a quick way to explore your repository's history, commits, and files without leaving the command line.

Here's how to use git insta web:

1. Run git instaweb in your terminal.

2. Open a web browser and navigate to http://localhost:1234 (or the port number specified by Git).

3. You'll see a web-based interface showing your repository's history, commits, and files.

Amazon EMR: Amazon EMR (Elastic MapReduce) is a cloud-based big data processing service offered by Amazon Web Services (AWS). It allows users to easily process and analyze large amounts of data using popular open-source frameworks like Apache Hadoop, Apache Spark, and Presto.

Key Features:

1. Scalability: Scale up or down to match your workload, without worrying about infrastructure management.

2. Flexibility: Choose from various processing frameworks, including Hadoop, Spark, and Presto.

3. Security: Leverage AWS security features, such as encryption and access controls.

4. Cost-effective: Pay only for the resources used, with options for spot pricing and reserved instances.

5. Integration: Seamlessly integrate with other AWS services, like S3, Glue, and Redshift.

Use cases:

1. Data processing: Process large datasets for analytics, machine learning, and data science workloads.

2. Data transformation: Transform and prepare data for analysis, reporting, or loading into data warehouses.

3. Machine learning: Train and deploy machine learning models using popular frameworks like TensorFlow and Scikit-learn.

4. Data warehousing: Use EMR to extract, transform, and load data into Amazon Redshift or other data warehouses.

Benefits:

1. Faster insights: Quickly process and analyze large datasets to gain insights and make data-driven decisions.

2. Increased productivity: Focus on data analysis and processing, without managing infrastructure.

3. Cost savings: Reduce costs by paying only for resources used and leveraging spot pricing.

Common applications:

1. Log analysis: Process and analyze log data for application monitoring and troubleshooting.

2. Clickstream analysis: Analyze user behavior and clickstream data for e-commerce and web applications.

3. Genomic analysis: Process and analyze large genomic datasets for research and healthcare applications.

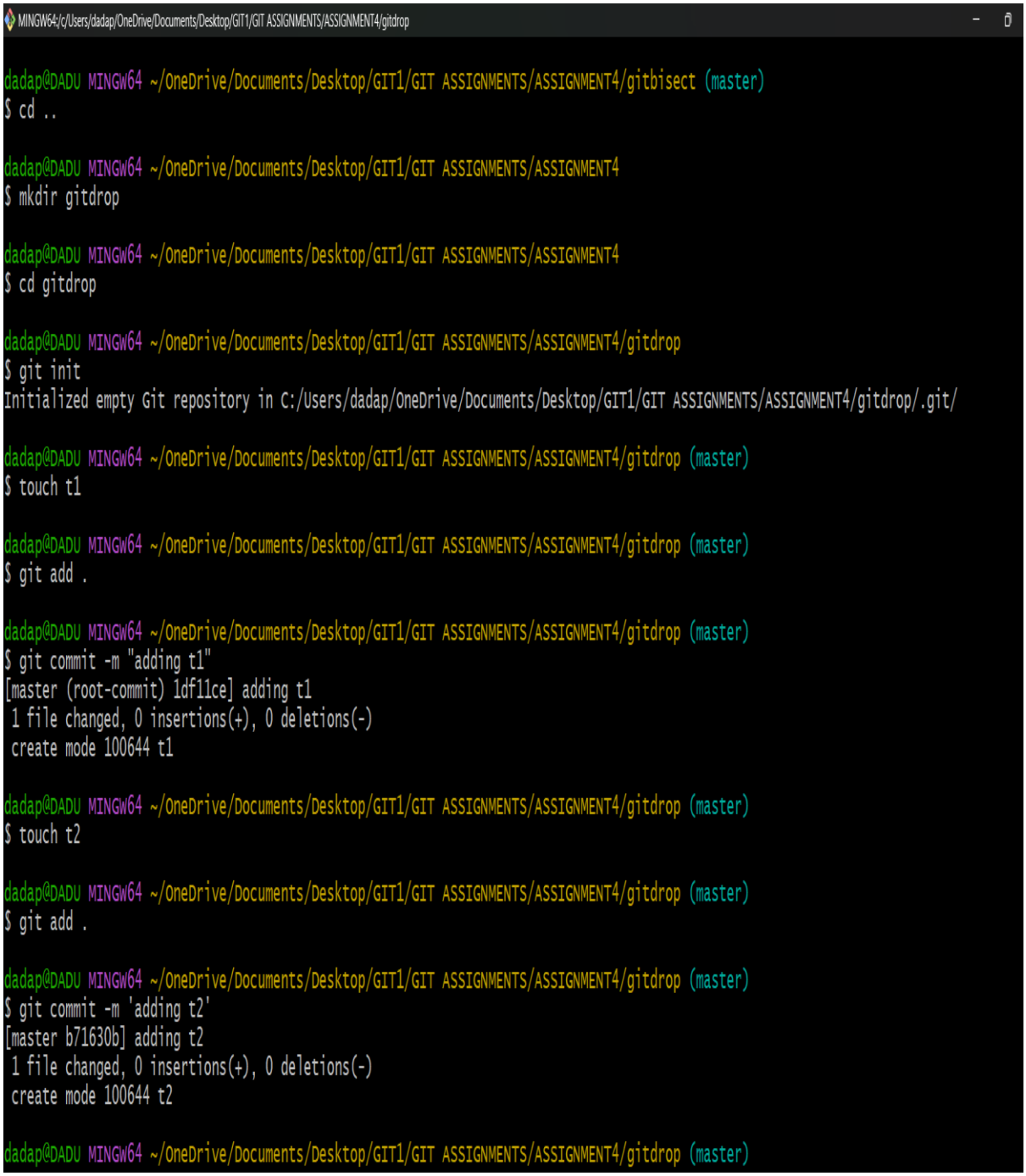
4. Financial analysis: Analyze financial data for risk management, portfolio optimization, and compliance.

Git drop: git drop is not a built-in Git command. However, you can use git reset or git restore to achieve similar results.

- git reset: Resets the current branch to a specific commit, discarding changes.

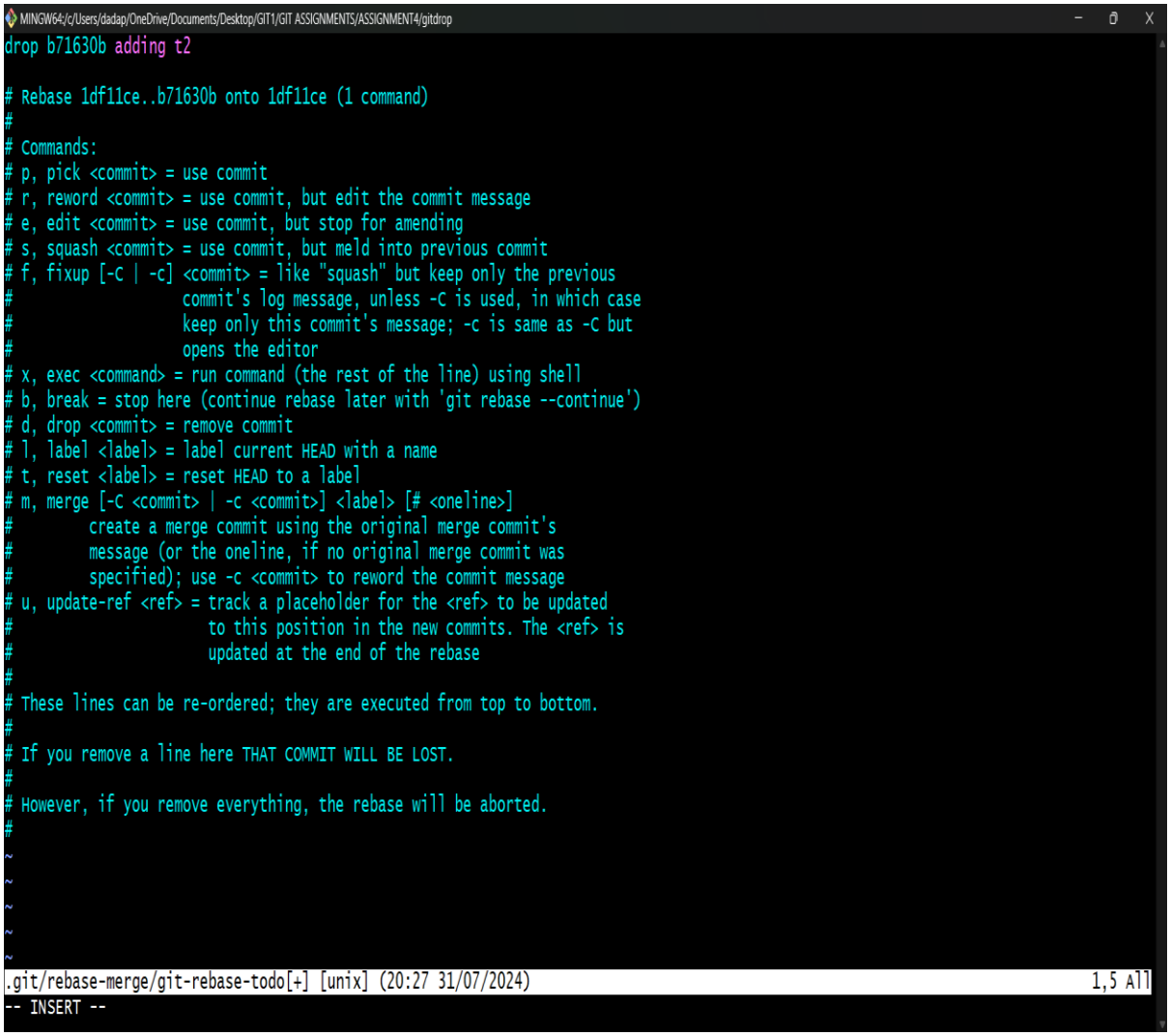
- git restore: Restores files to a previous state, discarding changes.

If you want to "drop" a commit, you can use git reset --hard to reset the branch to the previous commit, effectively deleting the most recent commit.



```
MINGW64:/c/Users/dadap/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT4/gitdrop

commit 1df11ce903762d55174cdb9a9c2045c312169034
Author: Dadu <dadapeerdpr@gmail.com>
Date:   Wed Jul 31 20:24:40 2024 +0530

    adding t1

dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT4/gitdrop (master)
$ git rebase -i HEAD~1
Successfully rebased and updated refs/heads/master.

dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT4/gitdrop (master)
$ git log
commit b71630bdc73114920b9c8093cf227b28606267be (HEAD -> master)
Author: Dadu <dadapeerdpr@gmail.com>
Date:   Wed Jul 31 20:24:59 2024 +0530

    adding t2

commit 1df11ce903762d55174cdb9a9c2045c312169034
Author: Dadu <dadapeerdpr@gmail.com>
Date:   Wed Jul 31 20:24:40 2024 +0530

    adding t1

dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT4/gitdrop (master)
$ git rebase -i HEAD~1
Successfully rebased and updated refs/heads/master.

dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT4/gitdrop (master)
$ git log
commit 1df11ce903762d55174cdb9a9c2045c312169034 (HEAD -> master)
Author: Dadu <dadapeerdpr@gmail.com>
Date:   Wed Jul 31 20:24:40 2024 +0530

    adding t1

dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT4/gitdrop (master)
$
```

```
MINGW64:/c/Users/dadap/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT4/gitdrop
pick b71630b adding t2

# Rebase 1df11ce..b71630b onto 1df11ce (1 command)
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
# f, fixup [-C | -c] <commit> = like "squash" but keep only the previous
#                    commit's log message, unless -C is used, in which case
#                    keep only this commit's message; -c is same as -C but
#                    opens the editor
# x, exec <command> = run command (the rest of the line) using shell
# b, break = stop here (continue rebase later with 'git rebase --continue')
# d, drop <commit> = remove commit
# l, label <label> = label current HEAD with a name
# t, reset <label> = reset HEAD to a label
# m, merge [-C <commit> | -c <commit>] <label> [# <oneline>]
#         create a merge commit using the original merge commit's
#         message (or the oneline, if no original merge commit was
#         specified); use -c <commit> to reword the commit message
# u, update-ref <ref> = track a placeholder for the <ref> to be updated
#                       to this position in the new commits. The <ref> is
#                       updated at the end of the rebase
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
~
~
~
~
~
.git/rebase-merge/git-rebase-todo [unix] (20:25 31/07/2024)                                                    1,1 All
"~/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT4/gitdrop/.git/rebase-merge/git-rebase-todo" [unix] 32L, 1503B
```


MINGW64:/c/Users/dadap/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT4/gitdrop
drop b71630b adding t2
Rebase 1df11ce..b71630b onto 1df11ce (1 command)
Commands:
p, pick <commit> = use commit
r, reword <commit> = use commit, but edit the commit message
e, edit <commit> = use commit, but stop for amending
s, squash <commit> = use commit, but meld into previous commit
f, fixup [-C | -c] <commit> = like "squash" but keep only the previous
commit's log message, unless -C is used, in which case
keep only this commit's message; -c is same as -C but
opens the editor
x, exec <command> = run command (the rest of the line) using shell
b, break = stop here (continue rebase later with 'git rebase --continue')
d, drop <commit> = remove commit
l, label <label> = label current HEAD with a name
t, reset <label> = reset HEAD to a label
m, merge [-C <commit> | -c <commit>] <label> [# <oneline>]
create a merge commit using the original merge commit's
message (or the oneline, if no original merge commit was
specified); use -c <commit> to reword the commit message
u, update-ref <ref> = track a placeholder for the <ref> to be updated
to this position in the new commits. The <ref> is
updated at the end of the rebase
#
These lines can be re-ordered; they are executed from top to bottom.
#
If you remove a line here THAT COMMIT WILL BE LOST.
#
However, if you remove everything, the rebase will be aborted.
#
.git/rebase-merge/git-rebase-todo[+] [unix] (20:27 31/07/2024) 1,5 All
-- INSERT --

Assignment 5: Raise a pull request and merge to the master branch from git hub

Pull request: A pull request is a way to propose changes to a repository's codebase. It allows developers to review and discuss changes before merging them into the main branch.

Here's a step-by-step overview of the pull request process:

1. **Create a new branch**: Make a new branch from the main branch (e.g., feature/new-feature).

2. **Make changes**: Code your changes, commit them, and push the branch to the remote repository.

3. **Create a pull request**: Go to the repository's web interface (e.g., GitHub, GitLab, Bitbucket) and create a pull request from your branch to the main branch.

4. **Review:** Others review your changes, leave comments, and discuss improvements.

5. **Update**: Address feedback by making additional changes, committing, and pushing the updated branch.

6. **Approve**: Once approved, a maintainer merges the pull request into the main branch.

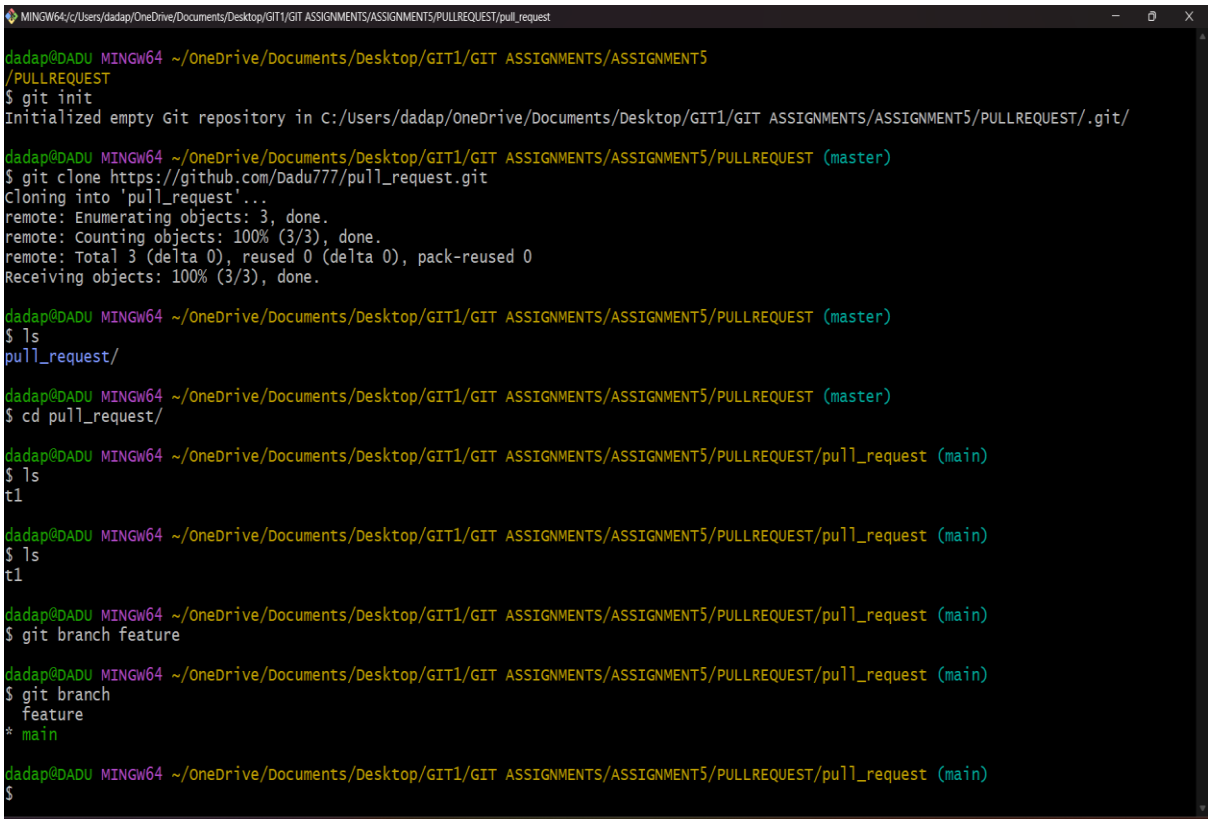
7. **Close**: The pull request is closed, and the branch can be deleted.

Pull request benefits:

- Code review: Ensures changes meet quality and style standards.

- Collaboration: Facilitates discussion and feedback among team members.

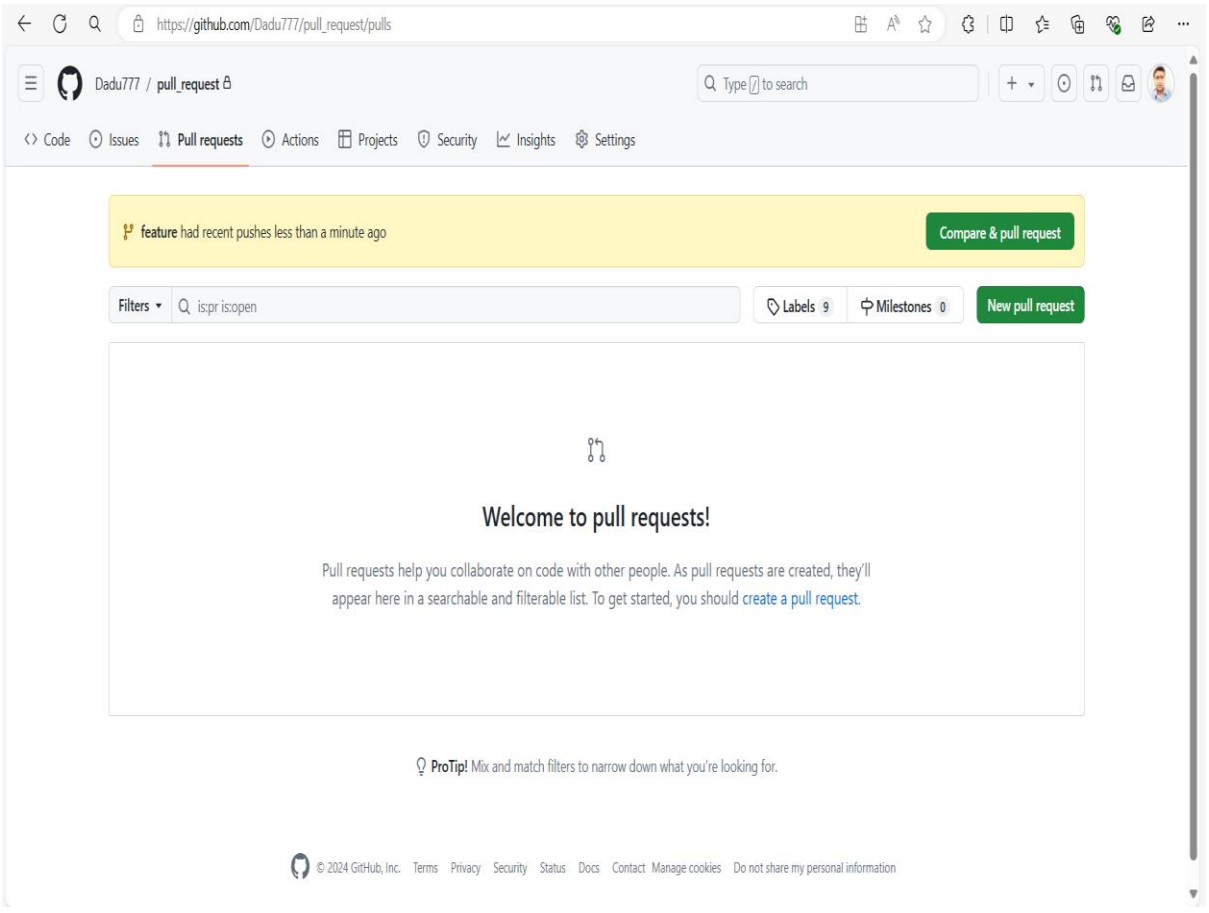
- Version control: Tracks changes and maintains a record of updates.

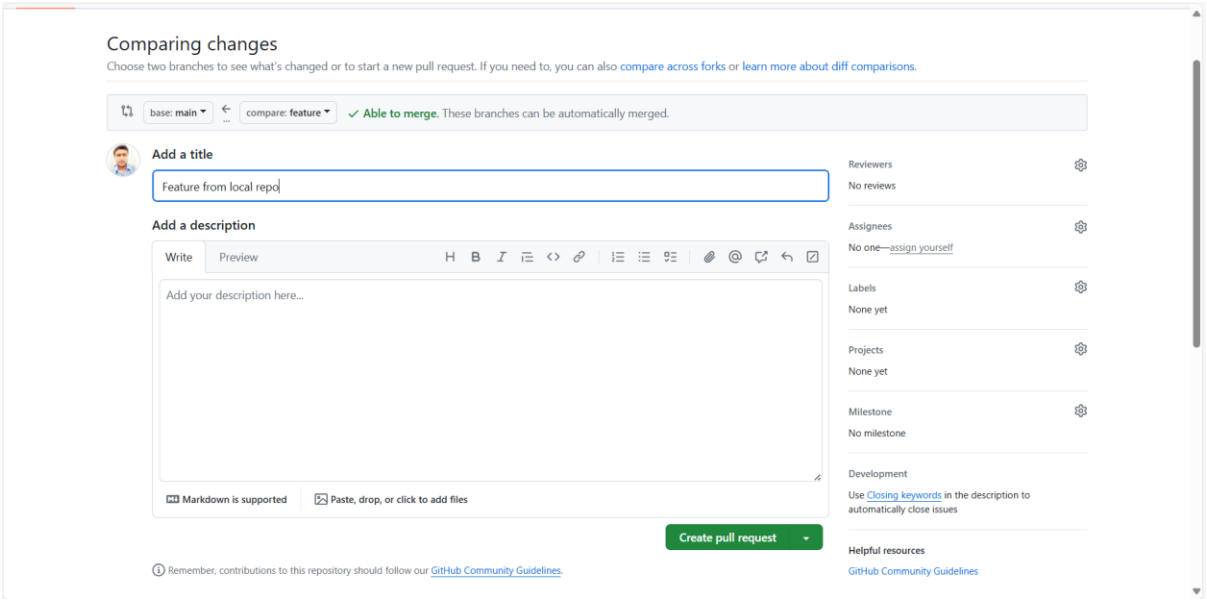


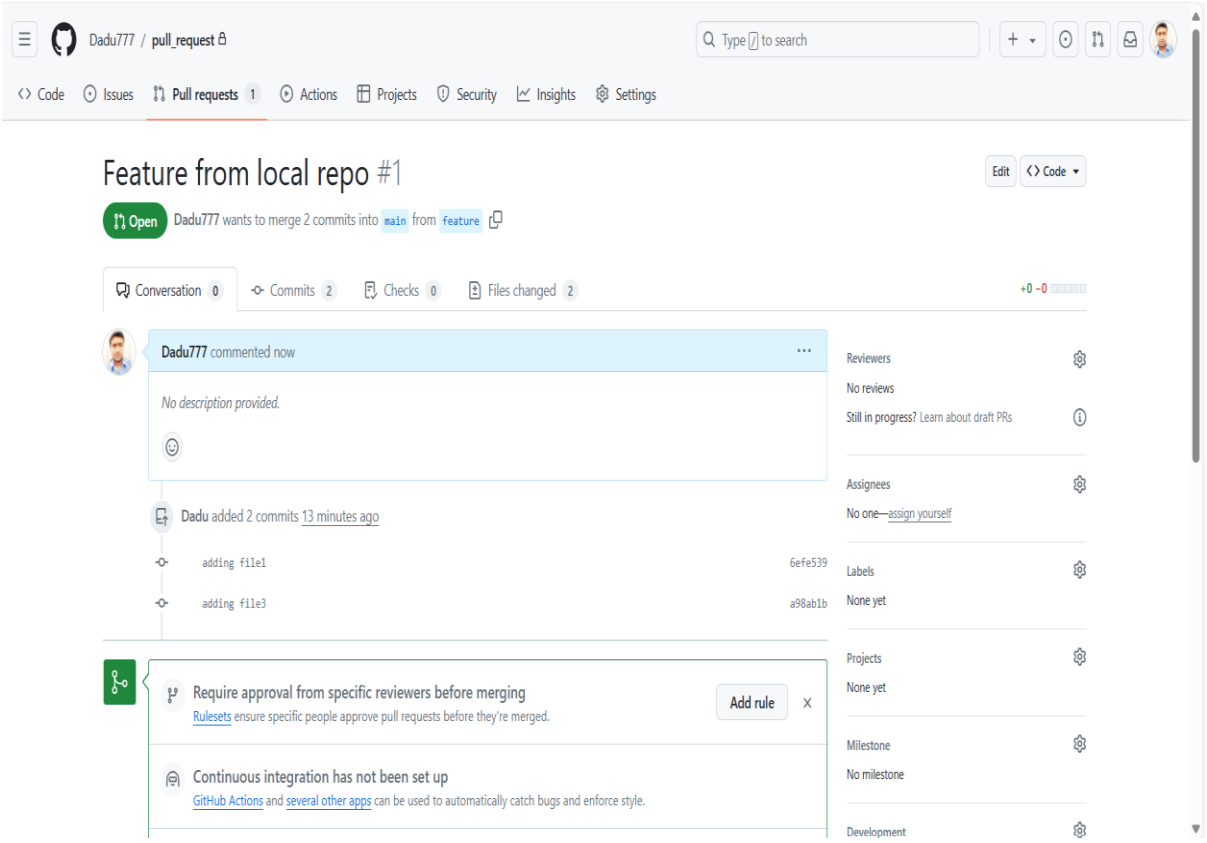
```
dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT5/PULLREQUEST/pull_request (main)
$ git checkout feature
Switched to branch 'feature'

dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT5/PULLREQUEST/pull_request (feature)
$ touch file1

dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT5/PULLREQUEST/pull_request (feature)
$ git add .

dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT5/PULLREQUEST/pull_request (feature)
$ git commit -m "adding file1"
[feature 6efe539] adding file1
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 file1

dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT5/PULLREQUEST/pull_request (feature)
$ git checkout main
Switched to branch 'main'
Your branch is up to date with 'origin/main'.
```

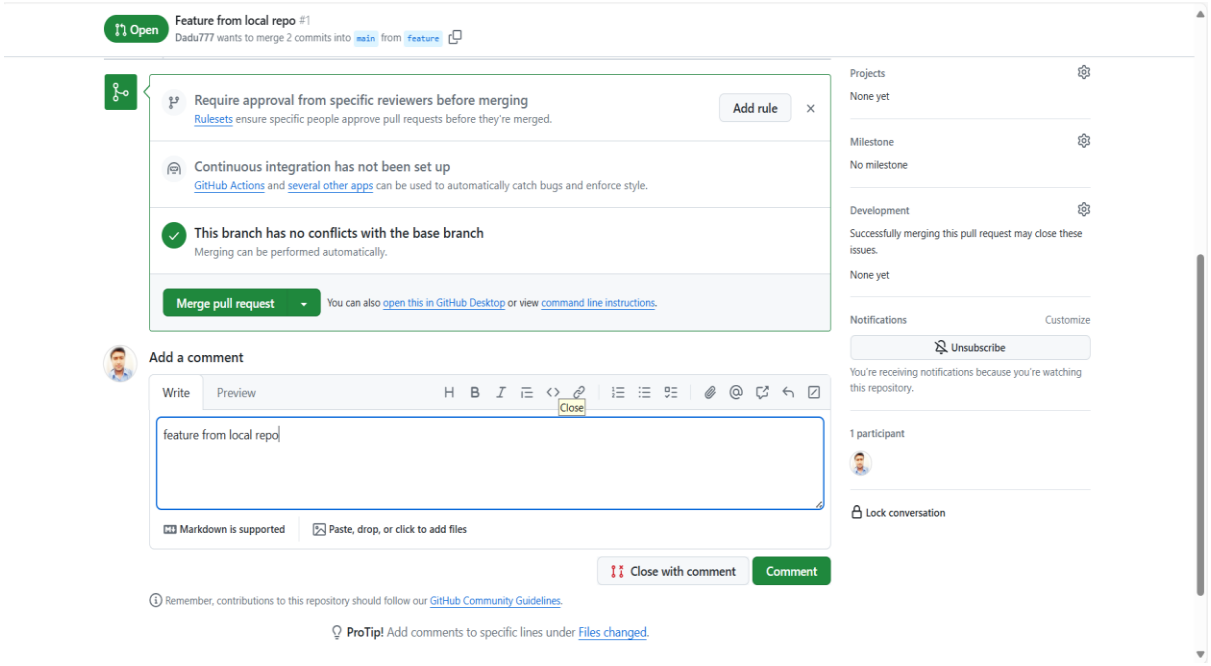
```
MINGW64:/c/Users/dadap/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT5/PULLREQUEST/pull_request
dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT5/PULLREQUEST/pull_request (feature)
$ touch file3

dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT5/PULLREQUEST/pull_request (feature)
$ git add .

dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT5/PULLREQUEST/pull_request (feature)
$ git commit -m "adding file3"
[feature a98ab1b] adding file3
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 file3

dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT5/PULLREQUEST/pull_request (feature)
$ git push
fatal: The current branch feature has no upstream branch.
To push the current branch and set the remote as upstream, use

    git push --set-upstream origin feature

To have this happen automatically for branches without a tracking
upstream, see 'push.autoSetupRemote' in 'git help config'.


dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT5/PULLREQUEST/pull_request (feature)
$ git checkout main
Switched to branch 'main'
Your branch is up to date with 'origin/main'.

dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT5/PULLREQUEST/pull_request (main)
$ git push
Everything up-to-date

dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT5/PULLREQUEST/pull_request (main)
$ git checkout feature
Switched to branch 'feature'
```

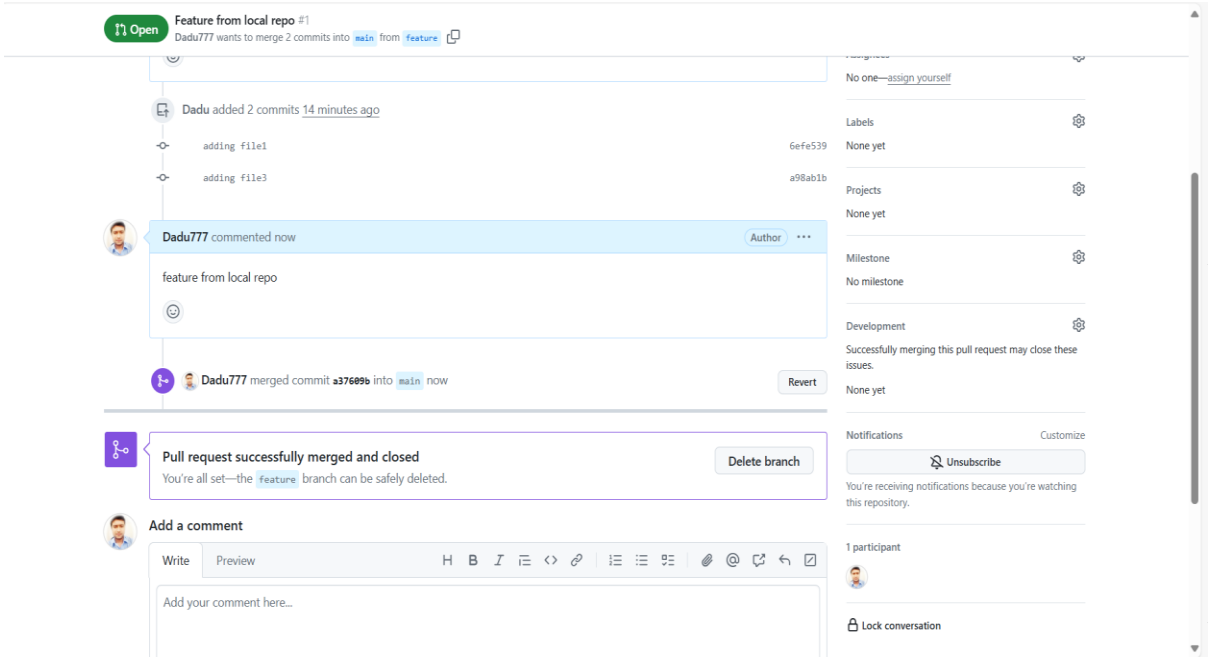
```
dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT5/PULLREQUEST/pull_request (main)
$ git push
Everything up-to-date

dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT5/PULLREQUEST/pull_request (main)
$ git checkout feature
Switched to branch 'feature'

dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT5/PULLREQUEST/pull_request (feature)
$ git push --set-upstream origin feature
Enumerating objects: 6, done.
Counting objects: 100% (6/6), done.
Delta compression using up to 4 threads
Compressing objects: 100% (4/4), done.
Writing objects: 100% (5/5), 483 bytes | 483.00 KiB/s, done.
Total 5 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
remote:
remote: Create a pull request for 'feature' on GitHub by visiting:
remote:      https://github.com/Dadu777/pull_request/pull/new/feature
remote:
To https://github.com/Dadu777/pull_request.git
 * [new branch]      feature -> feature
branch 'feature' set up to track 'origin/feature'.

dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/GIT1/GIT ASSIGNMENTS/ASSIGNMENT5/PULLREQUEST/pull_request (feature)
$
```




Comparing changes
Choose two branches to see what's changed or to start a new pull request. If you need to, you can also compare across forks or learn more about diff comparisons.
base: main
compare: feature
Able to merge. These branches can be automatically merged.
Add a title
Feature from local repo
Add a description
Write
Preview
Add your description here...
Markdown is supported
Paste, drop, or click to add files
Create pull request
Remember, contributions to this repository should follow our GitHub Community Guidelines.
Reviewers
No reviews
Assignees
No one—assign yourself
Labels
None yet
Projects
None yet
Milestone
No milestone
Development
Use Closing keywords in the description to automatically close issues
Helpful resources
GitHub Community Guidelines


Dadu777 / pull_request
Type / to search
Code
Issues
Pull requests 1
Actions
Projects
Security
Insights
Settings
Feature from local repo #1
Edit
Code
Open
Dadu777 wants to merge 2 commits into main from feature
Conversation 0
Commits 2
Checks 0
Files changed 2
+0 −0
Dadu777 commented now
No description provided.
Dadu added 2 commits 13 minutes ago
adding file1
6efe539
adding file3
a98ab1b
Require approval from specific reviewers before merging
Rulesets ensure specific people approve pull requests before they're merged.
Add rule
Continuous integration has not been set up
GitHub Actions and several other apps can be used to automatically catch bugs and enforce style.
Reviewers
No reviews
Still in progress? Learn about draft PRs
Assignees
No one—assign yourself
Labels
None yet
Projects
None yet
Milestone
No milestone
Development


Open
Feature from local repo #1
Dadu777 wants to merge 2 commits into main from feature
Require approval from specific reviewers before merging
Rulesets ensure specific people approve pull requests before they're merged.
Add rule
Continuous integration has not been set up
GitHub Actions and several other apps can be used to automatically catch bugs and enforce style.
This branch has no conflicts with the base branch
Merging can be performed automatically.
Merge pull request
You can also open this in GitHub Desktop or view command line instructions.
Add a comment
Write
Preview
Close
feature from local repo
Markdown is supported
Paste, drop, or click to add files
Close with comment
Comment
Remember, contributions to this repository should follow our GitHub Community Guidelines.
ProTip! Add comments to specific lines under Files changed.
Projects
None yet
Milestone
No milestone
Development
Successfully merging this pull request may close these issues.
None yet
Notifications
Customize
Unsubscribe
You're receiving notifications because you're watching this repository.
1 participant
Lock conversation


Open
Feature from local repo #1
Dadu777 wants to merge 2 commits into main from feature
No one—assign yourself
Dadu added 2 commits 14 minutes ago
adding file1
6efe539
adding file3
a98ab1b
Dadu777 commented now
Author
feature from local repo
Dadu777 merged commit a37609b into main now
Revert
Pull request successfully merged and closed
You're all set—the feature branch can be safely deleted.
Delete branch
Add a comment
Write
Preview
Add your comment here...
Labels
None yet
Projects
None yet
Milestone
No milestone
Development
Successfully merging this pull request may close these issues.
None yet
Notifications
Customize
Unsubscribe
You're receiving notifications because you're watching this repository.
1 participant
Lock conversation

Left out git commands in notes:

Git cherry -pick: git cherry-pick is a command used in Git to apply changes from specific commits from one branch to another. It allows you to select individual commits from one branch and apply them to your current branch, without merging all changes from the source branch.

Here's how to use git cherry-pick:

1. Identify the commit hash: Find the hash of the commit you want to cherry-pick. You can find this using git log or any Git visual tool.
2. Switch to the target branch: Ensure you are on the branch where you want to apply the commit. Use git checkout <branch-name> to switch branches.
3. Cherry-pick the commit: Use git cherry-pick <commit-hash> to apply the changes from the identified commit to your current branch.

Example

Assume you have the following situation:

- You have a feature branch feature-branch.
- There is a commit in feature-branch with the hash a1b2c3d4.
- You want to apply this commit to your main branch.

Here are the steps:

Step 1: Switch to the main branch

git checkout main

Step 2: Cherry-pick the commit

git cherry-pick a1b2c3d4

Handling Conflicts

If there are conflicts during the cherry-pick, Git will pause and allow you to resolve them. Once you have resolved the conflicts, you can continue the cherry-pick process:

After resolving conflicts

git add <resolved-files>

git cherry-pick --continue

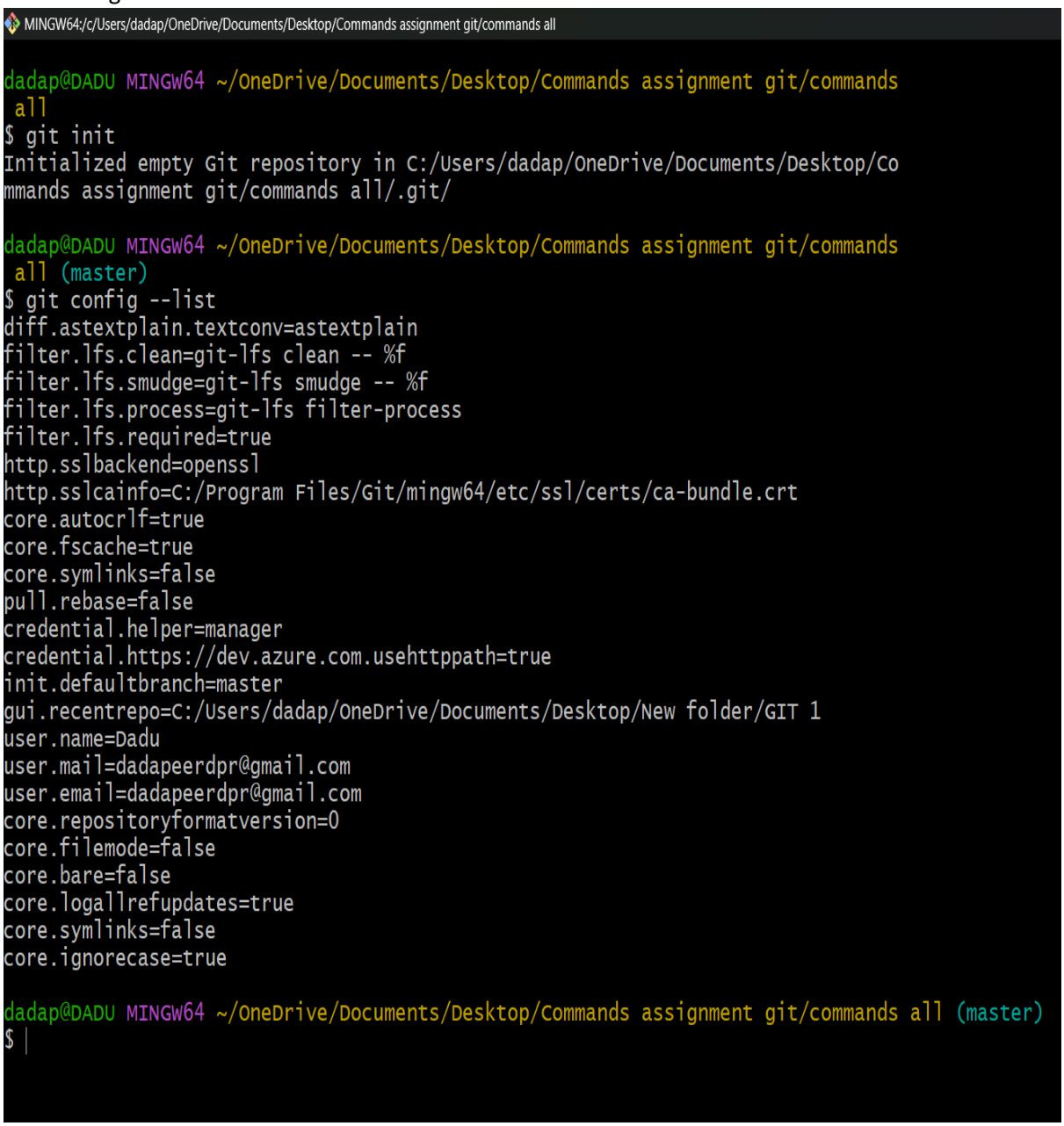
Additional Options

- Abort cherry-pick: If you want to abort the cherry-pick process due to conflicts or any other reason, you can use git cherry-pick --abort.
- Skip commit: If you want to skip the current commit in case of conflicts, you can use git cherry-pick --skip.

git cherry-pick is a powerful tool for selectively applying changes, especially when you need to port bug fixes or features from one branch to another without merging all changes.

```
dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/Commands assignment git/commands all (branch1)
$ git checkout master
Switched to branch 'master'

dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/Commands assignment git/commands all (master)
$ git cherry-pick 2e2f61c35410d4c91060d3207c77f6c82158144a
[master 1c795f1] adding t4
 Date: Sun Aug 4 17:04:26 2024 +0530
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 t4

dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/Commands assignment git/commands all (master)
$ git log
commit 1c795f11cb0c2013592317cc15aa15aa2f6d5f87 (HEAD -> master)
Author: Dadu <dadapeerdpr@gmail.com>
Date:   Sun Aug 4 17:04:26 2024 +0530

    adding t4

commit 184858e9ca0a30f917258bbb5003ad654e5bc9f5
Author: Dadu <dadapeerdpr@gmail.com>
Date:   Sun Aug 4 17:02:27 2024 +0530

    adding t2

commit 320c6f4086fe41403a18db9e36eb795efed6435c
Author: Dadu <dadapeerdpr@gmail.com>
Date:   Sun Aug 4 17:02:04 2024 +0530

    adding t1

dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/Commands assignment git/commands all (master)
$
```

Git config: git config is a Git command used to configure Git settings and preferences. These settings can be global (applying to all repositories for a user) or local (specific to a single repository). The git config command can be used to set user information, configure behavior, and define aliases, among other things.



Git merge branch status:

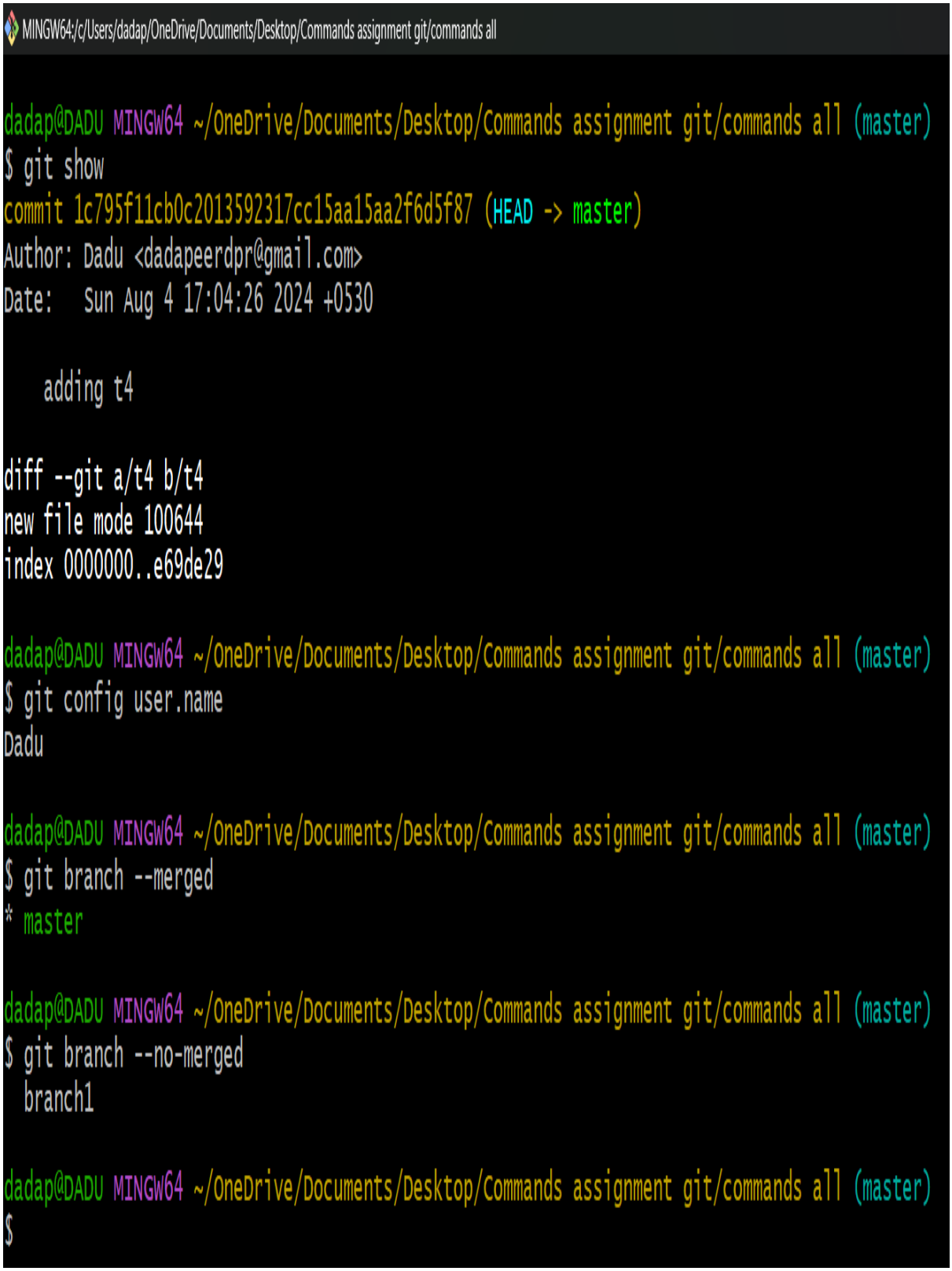
To check merged and unmerged branches in Git, you can use the git branch command with specific options

Checking Merged Branches
git branch –merged

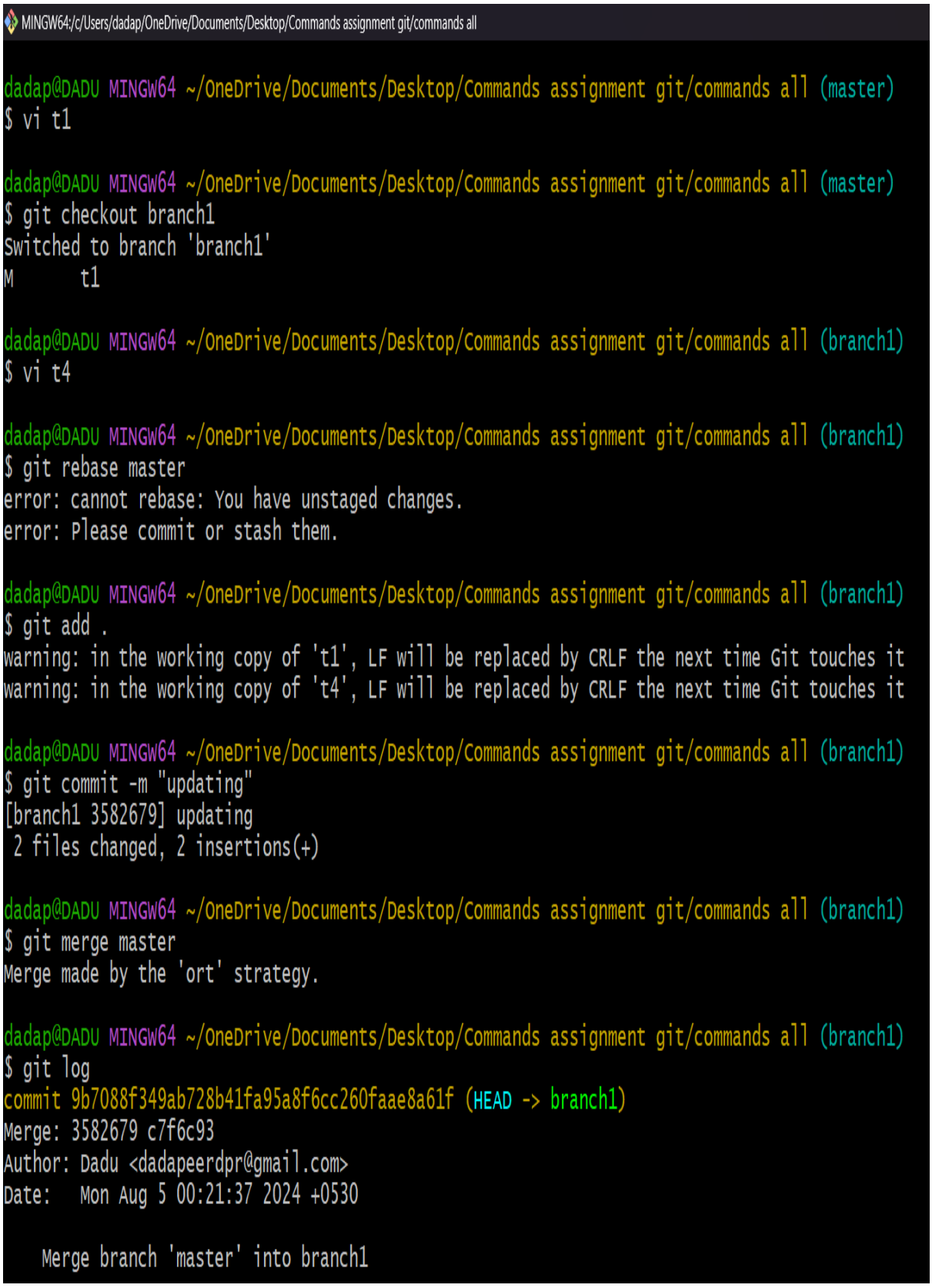
Checking Unmerged Branches

git branch --no-merged

git show: git show is a command in Git that displays various types of objects, like commits, trees, and blobs. It provides a detailed view of a specific commit, including the changes made to the files, the author, the date, and the commit message.



Git merge: git merge is a command in Git that allows you to integrate changes from different branches. It combines the contents of two branches into one.



```
MINGW64:/c/Users/dadap/OneDrive/Documents/Desktop/Commands assignment git/commands all

dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/Commands assignment git/commands all (branch1)
$ git log
commit 9b7088f349ab728b41fa95a8f6cc260faae8a61f (HEAD -> branch1)
Merge: 3582679 c7f6c93
Author: Dadu <dadapeerdpr@gmail.com>
Date:   Mon Aug 5 00:21:37 2024 +0530

    Merge branch 'master' into branch1

commit 35826794e8a1a1dd1e47ed99b448550b392b157d
Author: Dadu <dadapeerdpr@gmail.com>
Date:   Mon Aug 5 00:21:26 2024 +0530

    updating

commit c7f6c93c3307b489759c95c468b7e73c012f5ce9 (master)
Merge: b54a907 a8cce73
Author: Dadu <dadapeerdpr@gmail.com>
Date:   Mon Aug 5 00:19:19 2024 +0530

    Merge branch 'branch1'

commit a8cce73320d7b921b051ade43888903a5da40218
Author: Dadu <dadapeerdpr@gmail.com>
Date:   Mon Aug 5 00:18:23 2024 +0530

    updating t3

commit b54a907735018f59ef197a4ccf37c941f428e07e
Author: Dadu <dadapeerdpr@gmail.com>
Date:   Mon Aug 5 00:15:58 2024 +0530

    adding t3

commit 1c795f11cb0c2013592317cc15aa15aa2f6d5f87
Author: Dadu <dadapeerdpr@gmail.com>
Date:   Sun Aug 4 17:04:26 2024 +0530
```

```
MINGW64:/c/Users/dadap/OneDrive/Documents/Desktop/Commands assignment git/commands all
Merge branch 'master' into branch1
# Please enter a commit message to explain why this merge is necessary,
# especially if it merges an updated upstream into a topic branch.
#
# Lines starting with '#' will be ignored, and an empty message aborts
# the commit.
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
.git/MERGE_MSG [unix] (00:21 05/08/2024)
"~/OneDrive/Documents/Desktop/Commands assignment git/commands all/.git/MERGE_MSG" [unix] 6L, 261B
```

Git squash:

Squashing commits in Git means combining multiple commits into a single commit. This is often used to clean up commit history before merging a feature branch into the main branch.

Squashing Commits Using Interactive Rebase

One of the most common ways to squash commits is by using interactive rebase. Here's how to do it:

1. **Start interactive rebase:**

git rebase -i HEAD~n

Replace n with the number of commits you want to squash. For example, if you want to squash the last 3 commits, you would use HEAD~3.

2. **Mark commits for squashing:** An editor will open with a list of commits. The first commit will be marked as pick. Change pick to squash (or s) for the commits you want to squash into the previous commit.

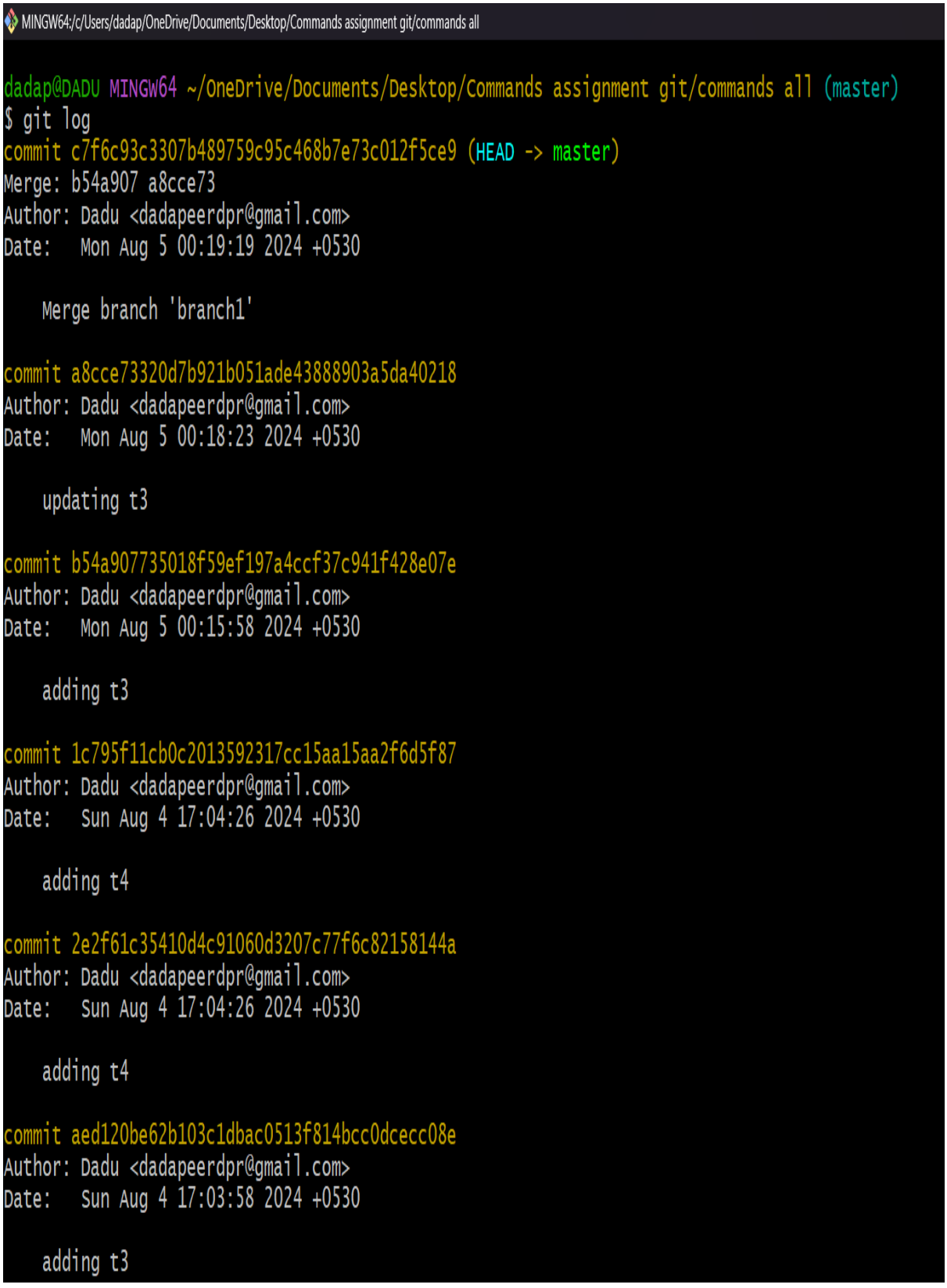
pick abc1234 First commit message

squash def5678 Second commit message

squash ghi9012 Third commit message

3. **Save and close the editor:** After marking the commits, save and close the editor. A new editor will open to combine the commit messages.
4. **Combine commit messages:** Edit the commit message as desired, then save and close the editor.
5. **Complete the rebase:** Git will squash the commits and reapply them. If there are conflicts, you will need to resolve them manually, then continue the rebase with:

git rebase --continue

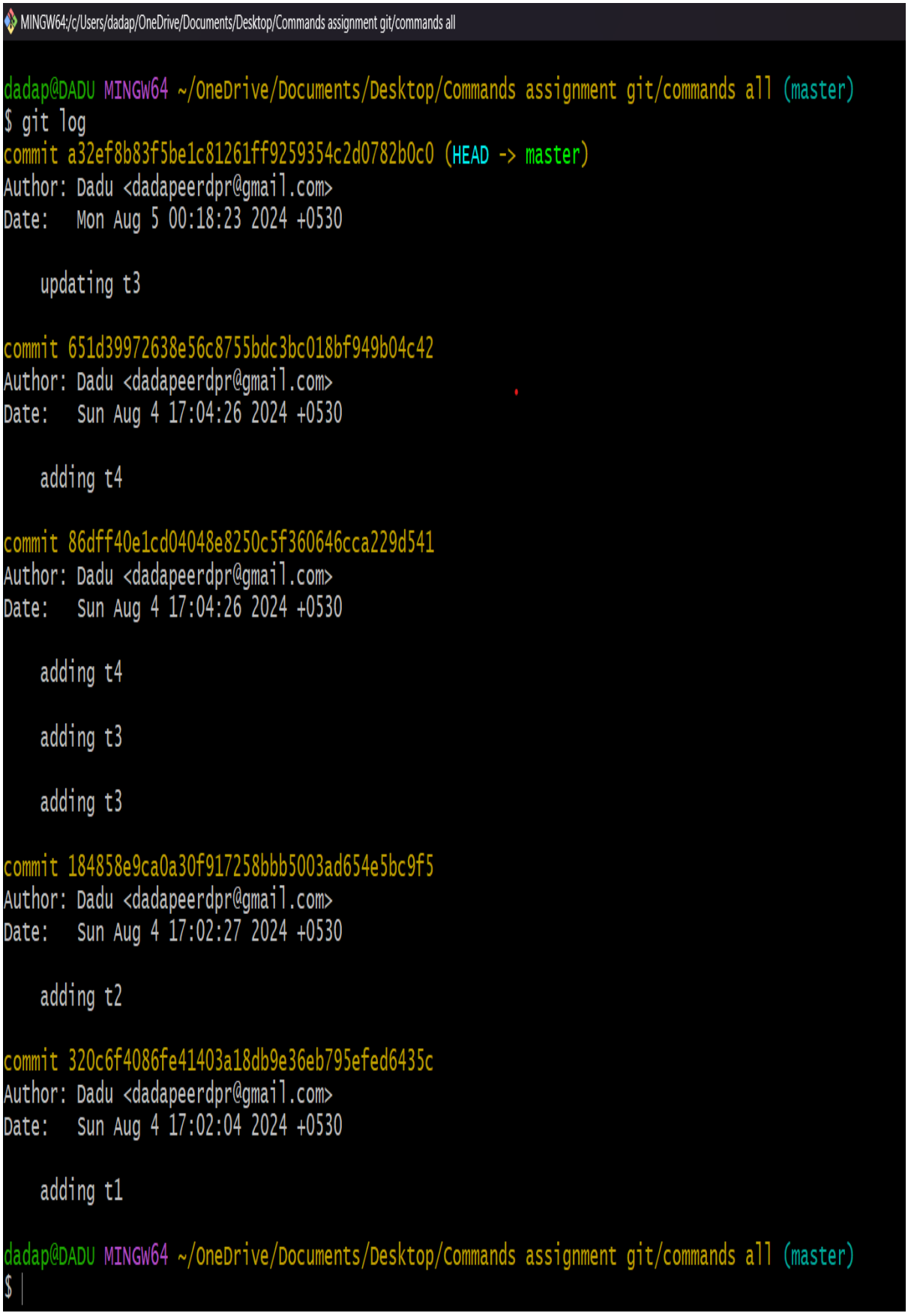

MINGW64:/c/Users/dadap/OneDrive/Documents/Desktop/Commands assignment git/commands all
dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/Commands assignment git/commands all (master)
$ git log
commit c7f6c93c3307b489759c95c468b7e73c012f5ce9 (HEAD -> master)
Merge: b54a907 a8cce73
Author: Dadu <dadapeerdpr@gmail.com>
Date: Mon Aug 5 00:19:19 2024 +0530
Merge branch 'branch1'
commit a8cce73320d7b921b051ade43888903a5da40218
Author: Dadu <dadapeerdpr@gmail.com>
Date: Mon Aug 5 00:18:23 2024 +0530
updating t3
commit b54a907735018f59ef197a4ccf37c941f428e07e
Author: Dadu <dadapeerdpr@gmail.com>
Date: Mon Aug 5 00:15:58 2024 +0530
adding t3
commit 1c795f11cb0c2013592317cc15aa15aa2f6d5f87
Author: Dadu <dadapeerdpr@gmail.com>
Date: Sun Aug 4 17:04:26 2024 +0530
adding t4
commit 2e2f61c35410d4c91060d3207c77f6c82158144a
Author: Dadu <dadapeerdpr@gmail.com>
Date: Sun Aug 4 17:04:26 2024 +0530
adding t4
commit aed120be62b103c1dbac0513f814bcc0dcecc08e
Author: Dadu <dadapeerdpr@gmail.com>
Date: Sun Aug 4 17:03:58 2024 +0530
adding t3

```
MINGW64:/c/Users/dadap/OneDrive/Documents/Desktop/Commands assignment git/commands all
Date:   Sun Aug 4 17:03:58 2024 +0530

    adding t3

commit 184858e9ca0a30f917258bbb5003ad654e5bc9f5

dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/Commands assignment git/commands all (master)
$ git rebase -i HEAD~3
[detached HEAD 86dff40] adding t4
 Date: Sun Aug 4 17:04:26 2024 +0530
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 t3
Successfully rebased and updated refs/heads/master.

dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/Commands assignment git/commands all (master)
$ git log
commit a32ef8b83f5be1c81261ff9259354c2d0782b0c0 (HEAD -> master)
Author: Dadu <dadapeerdpr@gmail.com>
Date:   Mon Aug 5 00:18:23 2024 +0530

    updating t3

commit 651d39972638e56c8755bdc3bc018bf949b04c42
Author: Dadu <dadapeerdpr@gmail.com>
Date:   Sun Aug 4 17:04:26 2024 +0530

    adding t4

commit 86dff40e1cd04048e8250c5f360646cca229d541
Author: Dadu <dadapeerdpr@gmail.com>
Date:   Sun Aug 4 17:04:26 2024 +0530

    adding t4

    adding t3

    adding t3

commit 184858e9ca0a30f917258bbb5003ad654e5bc9f5
```

```
MINGW64:/c/Users/dadap/OneDrive/Documents/Desktop/Commands assignment git/commands all
pick 1c795f1 adding t4
pick b54a907 adding t3
pick aed120b adding t3
pick 2e2f61c adding t4
pick a8cce73 updating t3

# Rebase 184858e..c7f6c93 onto 184858e (5 commands)
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
# f, fixup [-C | -c] <commit> = like "squash" but keep only the previous
#                    commit's log message, unless -C is used, in which case
#                    keep only this commit's message; -c is same as -C but
#                    opens the editor
# x, exec <command> = run command (the rest of the line) using shell
# b, break = stop here (continue rebase later with 'git rebase --continue')
# d, drop <commit> = remove commit
# l, label <label> = label current HEAD with a name
# t, reset <label> = reset HEAD to a label
# m, merge [-C <commit> | -c <commit>] <label> [# <oneline>]
#         create a merge commit using the original merge commit's
#         message (or the oneline, if no original merge commit was
#         specified); use -c <commit> to reword the commit message
# u, update-ref <ref> = track a placeholder for the <ref> to be updated
#                       to this position in the new commits. The <ref> is
#                       updated at the end of the rebase
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
~
.git/rebase-merge/git-rebase-todo [unix] (00:28 05/08/2024)
"~/OneDrive/Documents/Desktop/Commands assignment git/commands all/.git/rebase-merge/git-rebase-todo" [unix] 36L, 1598B
```

```
MINGW64:/c/Users/dadap/OneDrive/Documents/Desktop/Commands assignment git/commands all
pick 1c795f1 adding t4
squash b54a907 adding t3
squash aed120b adding t3
pick 2e2f61c adding t4
pick a8cce73 updating t3

# Rebase 184858e..c7f6c93 onto 184858e (5 commands)
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
# f, fixup [-C | -c] <commit> = like "squash" but keep only the previous
#                    commit's log message, unless -C is used, in which case
#                    keep only this commit's message; -c is same as -C but
#                    opens the editor
# x, exec <command> = run command (the rest of the line) using shell
# b, break = stop here (continue rebase later with 'git rebase --continue')
# d, drop <commit> = remove commit
# l, label <label> = label current HEAD with a name
# t, reset <label> = reset HEAD to a label
# m, merge [-C <commit> | -c <commit>] <label> [# <oneline>]
#         create a merge commit using the original merge commit's
#         message (or the oneline, if no original merge commit was
#         specified); use -c <commit> to reword the commit message
# u, update-ref <ref> = track a placeholder for the <ref> to be updated
#                       to this position in the new commits. The <ref> is
#                       updated at the end of the rebase
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
~
.git/rebase-merge/git-rebase-todo[+] [unix] (00:28 05/08/2024)
-- INSERT --
```

```
MINGW64:/c/Users/dadap/OneDrive/Documents/Desktop/Commands assignment git/commands all
# This is a combination of 3 commits.
# This is the 1st commit message:

adding t4

# This is the commit message #2:

adding t3

# This is the commit message #3:

adding t3

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# Date:      Sun Aug 4 17:04:26 2024 +0530
#
# interactive rebase in progress; onto 184858e
# Last commands done (3 commands done):
#    squash b54a907 adding t3
#    squash aed120b adding t3
# Next commands to do (2 remaining commands):
#    pick 2e2f61c adding t4
#    pick a8cce73 updating t3
# You are currently rebasing branch 'master' on '184858e'.
#
# Changes to be committed:
#       new file:   t3
#
~
~
~
~
~
~
~
.git/COMMIT_EDITMSG [unix] (00:33 05/08/2024)
"~/OneDrive/Documents/Desktop/Commands assignment git/commands all/.git/COMMIT_EDITMSG" [unix] 30L, 714B
```


MINGW64:/c/Users/dadap/OneDrive/Documents/Desktop/Commands assignment git/commands all
dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/Commands assignment git/commands all (master)
$ git log
commit a32ef8b83f5be1c81261ff9259354c2d0782b0c0 (HEAD -> master)
Author: Dadu <dadapeerdpr@gmail.com>
Date: Mon Aug 5 00:18:23 2024 +0530
updating t3
commit 651d39972638e56c8755bdc3bc018bf949b04c42
Author: Dadu <dadapeerdpr@gmail.com>
Date: Sun Aug 4 17:04:26 2024 +0530
adding t4
commit 86dff40e1cd04048e8250c5f360646cca229d541
Author: Dadu <dadapeerdpr@gmail.com>
Date: Sun Aug 4 17:04:26 2024 +0530
adding t4
adding t3
adding t3
commit 184858e9ca0a30f917258bbb5003ad654e5bc9f5
Author: Dadu <dadapeerdpr@gmail.com>
Date: Sun Aug 4 17:02:27 2024 +0530
adding t2
commit 320c6f4086fe41403a18db9e36eb795efed6435c
Author: Dadu <dadapeerdpr@gmail.com>
Date: Sun Aug 4 17:02:04 2024 +0530
adding t1
dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/Commands assignment git/commands all (master)
$

Git stash & Git pop:

git stash is a powerful command in Git that allows you to temporarily save your work-in-progress changes so you can switch branches or work on something else without committing those changes. Once you're ready to resume your work, you can apply the stashed changes using git stash pop.

Basic Usage

Stashing Changes

1. Stash your changes:

git stash

This command will save your modified tracked files and the working directory will be clean, as if you've just checked out the current branch.

2. Stash with a message:

git stash save "your message here"

Adding a message helps you identify the stash later.

3. Stash including untracked files:

git stash -u

The -u option (short for --include-untracked) will also stash your untracked files.

Viewing Stashed Changes

1. List stashes:

git stash list

This will display a list of all stashed changes.

Applying Stashed Changes

1. Apply the latest stash:

git stash pop

This will apply the most recently stashed changes and remove them from the stash list.

2. Apply a specific stash:

git stash pop stash@{n}

Replace n with the index of the stash you want to apply. You can find the index from the git stash list.

3. Apply the latest stash without removing it:

git stash apply

This command applies the latest stashed changes but keeps them in the stash list.

4. Apply a specific stash without removing it:

git stash apply stash@{n}

Again, replace n with the appropriate index.

Cleaning Up Stashes

1. Drop a specific stash:

git stash drop stash@{n}

This removes the specified stash from the list.

2. Clear all stashes:

git stash clear

This will remove all stashes from the stash list.

Example Workflow

1. Work on some changes: Make some changes to your files.
2. Stash your changes:

git stash save "work in progress"

3. Switch to another branch:

git checkout other-branch

4. Do some work and commit: Make changes, commit, etc.
5. Switch back to your original branch:

git checkout original-branch

6. Apply your stashed changes:

git stash pop

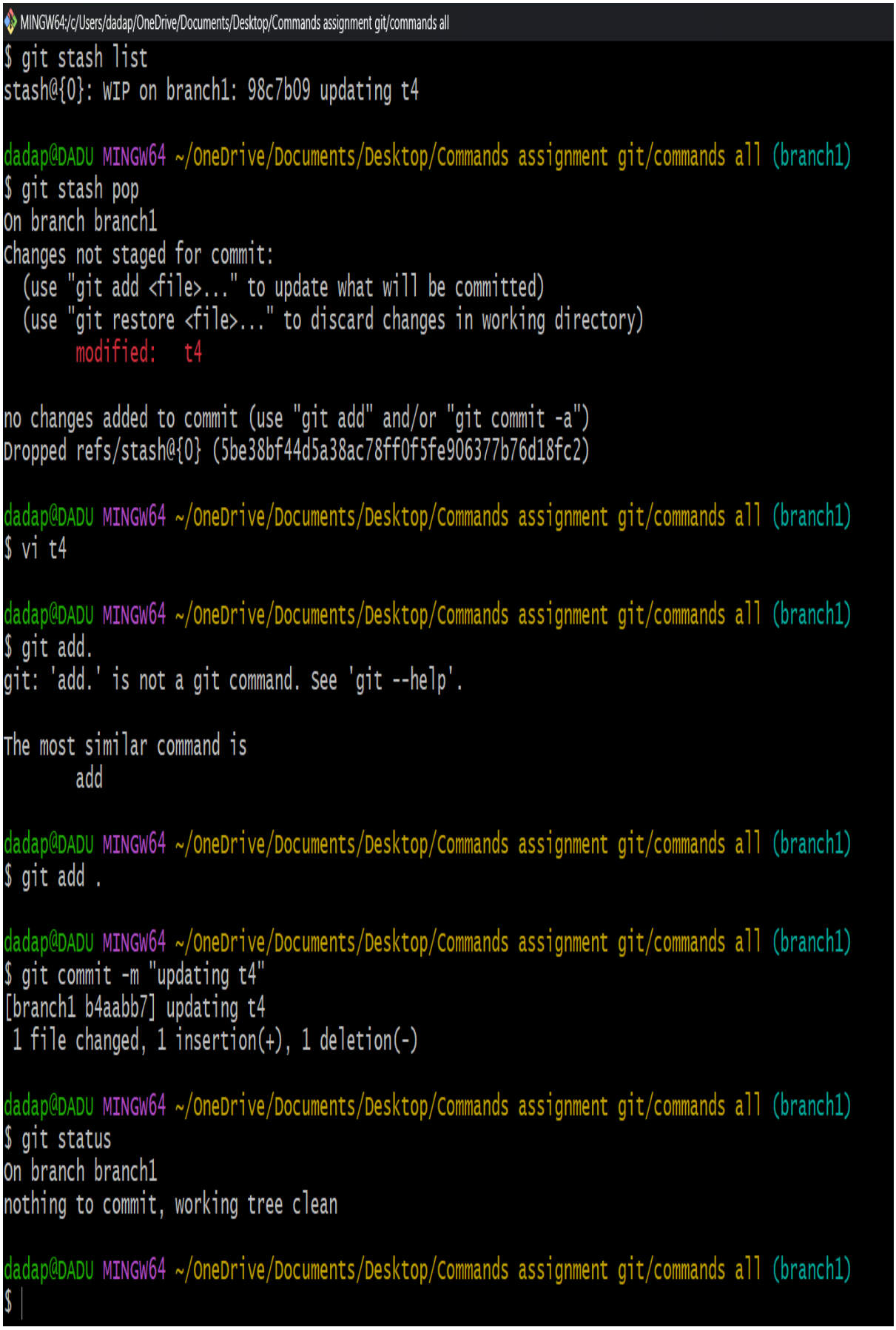
Summary

- Stash changes: git stash or git stash save "message"
- List stashes: git stash list
- Apply latest stash: git stash pop
- Apply specific stash: git stash pop stash@{n}
- Drop specific stash: git stash drop stash@{n}
- Clear all stashes: git stash clear

Using git stash and git stash pop, you can manage your work-in-progress changes efficiently, allowing you to switch contexts without losing your uncommitted work.

```
MINGW64:/c/Users/dadap/OneDrive/Documents/Desktop/Commands assignment git/commands all

dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/Commands assignment git/commands all (branch1)
$ git status
On branch branch1
nothing to commit, working tree clean

dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/Commands assignment git/commands all (branch1)
$ vi t4

dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/Commands assignment git/commands all (branch1)
$ git status
On branch branch1
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:   t4

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

dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/Commands assignment git/commands all (branch1)
$ git stash
Saved working directory and index state WIP on branch1: 98c7b09 updating t4

dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/Commands assignment git/commands all (branch1)
$ git checkout master
Switched to branch 'master'

dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/Commands assignment git/commands all (master)
$ git status
On branch master
nothing to commit, working tree clean

dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/Commands assignment git/commands all (master)
$ git checkout branch1
Switched to branch 'branch1'

dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/Commands assignment git/commands all (branch1)
$ git stash list
stash@{0}: WIP on branch1: 98c7b09 updating t4
```

```
MINGW64:/c/Users/dadap/OneDrive/Documents/Desktop/Commands assignment git/commands all
$ git checkout branch1
Switched to branch 'branch1'

dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/Commands assignment git/commands all (branch1)
$ git stash list
stash@{0}: WIP on branch1: 98c7b09 updating t4

dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/Commands assignment git/commands all (branch1)
$ git stash pop
On branch branch1
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:   t4

no changes added to commit (use "git add" and/or "git commit -a")
Dropped refs/stash@{0} (5be38bf44d5a38ac78ff0f5fe906377b76d18fc2)

dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/Commands assignment git/commands all (branch1)
$ vi t4

dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/Commands assignment git/commands all (branch1)
$ git add.
git: 'add.' is not a git command. See 'git --help'.

The most similar command is
        add

dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/Commands assignment git/commands all (branch1)
$ git add .

dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/Commands assignment git/commands all (branch1)
$ git commit -m "updating t4"
[branch1 b4aabb7] updating t4
 1 file changed, 1 insertion(+), 1 deletion(-)

dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/Commands assignment git/commands all (branch1)
$ git status
On branch branch1
```


MINGW64:/c/Users/dadap/OneDrive/Documents/Desktop/Commands assignment git/commands all
$ git stash list
stash@{0}: WIP on branch1: 98c7b09 updating t4
dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/Commands assignment git/commands all (branch1)
$ git stash pop
On branch branch1
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: t4
no changes added to commit (use "git add" and/or "git commit -a")
Dropped refs/stash@{0} (5be38bf44d5a38ac78ff0f5fe906377b76d18fc2)
dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/Commands assignment git/commands all (branch1)
$ vi t4
dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/Commands assignment git/commands all (branch1)
$ git add.
git: 'add.' is not a git command. See 'git --help'.
The most similar command is
add
dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/Commands assignment git/commands all (branch1)
$ git add .
dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/Commands assignment git/commands all (branch1)
$ git commit -m "updating t4"
[branch1 b4aabb7] updating t4
1 file changed, 1 insertion(+), 1 deletion(-)
dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/Commands assignment git/commands all (branch1)
$ git status
On branch branch1
nothing to commit, working tree clean
dadap@DADU MINGW64 ~/OneDrive/Documents/Desktop/Commands assignment git/commands all (branch1)
$

Git branching strategy:

Implementing an effective Git branching strategy can greatly enhance collaboration and streamline the development process.

Git Flow Branching Strategy

1. Main Branches:

- main (or master): This branch holds the production-ready code. Every commit to this branch should be deployable. It's typically protected and only updated through pull requests (PRs).
- develop: This is the integration branch for new features and changes. All feature branches are merged into develop after they have been tested and reviewed.

2. Supporting Branches:

- Feature Branches (feature/*):
    - Purpose: Used to develop new features for the upcoming or future releases.
    - Naming Convention: feature/feature-name
    - Base: Branch off from develop
    - Merge Back: Merge back into develop
- Release Branches (release/*):
    - Purpose: To prepare a new production release. It allows for last-minute fixes and minor bug corrections.
    - Naming Convention: release/release-version
    - Base: Branch off from develop
    - Merge Back: Merge into both main and develop
    - Tag: A version tag is created in main for the release (e.g., v1.0.0).
- Hotfix Branches (hotfix/*):
    - Purpose: For critical fixes that need to go directly into production.
    - Naming Convention: hotfix/hotfix-description
    - Base: Branch off from main
    - Merge Back: Merge into both main and develop
    - Tag: A version tag is created in main for the hotfix (e.g., v1.0.1).

Benefits of Git Flow

- Isolation of work: Each branch isolates work, reducing the risk of conflicts and broken code in the main branch.
- Parallel development: Multiple features, releases, and hotfixes can be worked on simultaneously without interfering with each other.

- Continuous delivery: main always contains production-ready code, facilitating continuous integration and deployment.
- Release management: Release branches allow for thorough testing and preparation before pushing to production.

This branching strategy can be adjusted according to the team's workflow, project requirements, and release schedules.

HTTP Status codes

What is an HTTP response code?

HTTP status codes are standardized responses issued by web servers in response to requests made by clients (such as web browsers) to indicate the status of those requests. These codes are part of the HTTP/1.1 standard and are grouped into five classes, each represented by the first digit of the three-digit code:

1xx codes (Information)

1xx status codes in the HTTP protocol are a kind of first link in the dialogue between the server and the client. Instead of providing a complete response to a request, they provide information about the current status, making data exchange more efficient. Let's take a closer look at them:

100 Continue. HTTP response code in which the server gives the green light to the user, allowing him to safely continue sending a large request.

101 Switching Protocols. The server tells the client that it is changing the rules of the game, for example, moving from HTTP to the more secure HTTPS. In this case, the “Upgrade” header is used for the protocol change.

102 Processing. This code is like a message that the server has accepted the request, but is still busy with a complex operation.

103 Early Hints. Here the server sends several indicative headers to the client before the main response, warning about something that may be relevant in the near future.

2xx code (Successful)

HTTP error codes in the group 2xx indicate a successful request from the server. They essentially act as a “green light” in the scope of web communications, confirming that everything is going according to plan and has been successfully completed.

200 OK. This status is used when the server processes a request by GET method without problems and returns the requested data in response. The "Content-Type" header reports the content type in the response. It just informs the client that the request was successful.

201 Created. Here the server announces the creation of a new resource.

202 Accepted. The server lets the user know that the request has been accepted, but will take time to respond.

203 Non —Authoritative Information. This code provides the client with data that may not be official, but can be used for comparison.

204 No Content. The server has processed the request but is not returning any additional content.

205 Reset Content. Here the client is instructed to reset the current view or data after sending.

206 Partial Content. This case indicates that the response contains only part of the requested content. The "Content-Range" header indicates the partial content range.

207 Multi-Status. The server has successfully completed a multi-operation request from the client, and the response contains information about the status of each of the operations.

226 IM Used. This code indicates that the server used the Incremental Metadata (IM) method and responded by passing only the modified resource parts to the client.

3xx codes (Redirects)

3xx codes in the HTTP protocol are like pointers that guide the user to a new resource location. They inform the client that follow-up steps must be taken to obtain the requested content or to be redirected to another resource. Let's immerse into the details of each of them:

300 Multiple Choices. The client receives a signal that there are several possible locations for the resource and is given a choice in response. In current circumstances, the "Location" header may indicate alternative options for the resource.

301 Moved Permanently. The server reports back to the user that the resource has been permanently moved to another location.

302 Found. This HTTP code is similar to a temporary redirect. The server informs the consumer that the resource is temporarily available at a different URL. The "Location" header points to the new URL for the temporary redirect.

303 See Other. The client is told that the resource is available at a different URL and must make a GET request to this new address.

304 Not Modified. This status tells the client that the resource has remained unchanged since the last request and does not need to be downloaded again. When making a request, the "If-Modified-Since" header is used to check if the resource has been modified.

305 Use Proxy. As a response, the server reports that it should use the specified proxy to access the requested resource.

306 (reserved) — The code has been reserved, but in fact it is not used.

307 Temporary Redirect. This code is similar to 302 Found, but requires the client to remain in the request method that was used in the original request.

308 Permanent Redirect. Indicates that the resource has made a permanent move to a new URI and the client should use the new URI for all future requests.

4xx HTTP Error (Client errors)

HTTP 4xx error codes indicate client errors. This means that the problem is on the user side, such as the web browser or app.

400 Bad Request. The server cannot process the request due to syntax errors, invalid data, or other errors on the client side.

401 Unauthorized. The server cannot process the request due to syntax errors, invalid data, or other errors on the client side.

402 Payment Required. The code is not active at the moment and is reserved for future use. It may indicate the need to pay before accessing the resource in the future.

HTTP Error 403 Forbidden. The client does not have sufficient rights to access the requested resource.

404 Not Found. The requested resource does not exist on the server. This is one of the most common user errors.

405 Method Not Allowed. The server does not support the specified request method in during this resource. The “Allow” header indicates the allowed methods for the resource. With this code,

406 Not Acceptable. The server cannot provide data in a format that can be accepted by the client.

407 Proxy Authentication Required. Authentication on proxy server is required for access the requested resource.

408 Request Timeout. The server was waiting to receive a request from the client, but timed out. The "Retry-After" header may indicate the time after which the request can be retried.

409 Conflict. The request cannot be completed due to a conflict with the current resource state.

410 Gone. The requested resource previously existed but has now been deleted and its restoration is not expected.

411 Length Required. The server demands to specify the content length in the request; the absence of this information is considered an error.

412 Precondition Failed. A precondition in the request is not met, that prevents it from executing.

413 Payload Too Large. The size of the request data exceeds the server limits.

414 URI Too Long. URI length in the request exceeds acceptable limits.

415 Unsupported Media Type. The server cannot process the data type provided in the request.

416 Range Not Satisfiable. HTTP error where the requested range does not match the current server data.

417 Expectation Failed. The expected condition in the "Expect" header was not met.

418 I’m a teapot. This code is included as a joke and does not imply any real action for the user or server, and is not a full-fledged error. It indicates that the server is a teapot and is not capable of making coffee.

421 Misdirected Request. The server does not process the request due to an error in the request or server configuration.

422 Unprocessable Entity. The server understands the request, but does not process it due to data errors.

423 Locked. The resource is blocked and cannot be processed.

424 Failed Dependency. The request depends on another unexecuted request.

425 Too Early. The server is not ready to process the request due to its early coming.

426 Upgrade Required. The server requires the use of a more advanced protocol to process the request.

428 Precondition Required. The server requires certain preconditions to be specified in the request.

429 Too Many Requests. The client sent too many requests in a short time, exceeding the server's limits.

431 Request Header Fields Too Large. Request headers exceed the maximum allowed size.

449 Retry with. Indicates that the request cannot be run by the current server, but can be successfully processed by another server, and the client should retry the request with a new URI.

451 Unavailable for Legal Reasons. The resuource is unavailable for legal reasons.

499 Client Closed Request. The server received the request, but the connection was closed by the client before processing completion.

HTTP 5xx error (Server errors)

HTTP 5xx error codes indicate the server problems. These codes indicate problems that have occurred on the server side, making the server unable to process the user's request in a right way. Let's take a closer look at them:

HTTP Error 500 Internal Server Error. The server encounters unexpected circumstances that prevent it from the request completion The "Server" header may indicate the server on which the error occurred.

501 Not Implemented. The server does not support the functionality required to process the client's request. The "Via" header may indicate the proxy server through which the error occurred.

502 Bad Gateway. This code means that the server that acts as proxy received an incorrect response from another server.

HTTP Error 503 Service Unavailable. The server is temporarily unable to process requests.

504 Gateway Timeout. The server, that acts as proxy, did not receive a timely response from another server.

505 HTTP Version Not Supported. The server does not support the HTTP protocol version specified in the request. As a backup option, the "Upgrade" header may indicate supported protocols.

506 Variant Also Negotiates. This status is not used in HTTP/1.1; however, if the server detects an internal configuration that results in content negotiation ambiguity, it may use this response.

507 Insufficient Storage. The server cannot fulfill the request due to insufficient storage space on the server.

508 Loop Detected. The server has detected a loop while processing the request, and refuses to complete the request in order to avoid an infinite loop.

509 Bandwidth Limit Exceeded. The error occurs when the server's bandwidth is exceeded due to high volume of requests or traffic.

510 Not Extended. The client must transfer additional extensions to continue the request.

511 Network Authentication Required. The client must authenticate itself in order to gain access to the network.

Why HTTP Status Codes Happen

1. **1xx: Informational**
    - Indicate that the request was received and understood, and the server is continuing to process it.
2. **2xx: Success**
    - Indicate that the request was successfully received, understood, and accepted.
3. **3xx: Redirection**
    - Indicate that further action needs to be taken by the client to complete the request.
4. **4xx: Client Error**
    - Indicate that there was an error with the request from the client, such as bad syntax, unauthorized access, or requesting a nonexistent resource.
5. **5xx: Server Error**
    - Indicate that the server failed to fulfill a valid request due to an internal issue, misconfiguration, or temporary overload.

How to Debug HTTP Status Codes

Debugging 1xx Codes

- **1xx Informational** responses are usually not visible to end-users and typically don't require debugging. These are mostly used for internal processing and protocol upgrades.

Debugging 2xx Codes

- **2xx Success** codes indicate successful requests and generally don't require debugging unless the client is not behaving as expected.

Debugging 3xx Codes

- **301 Moved Permanently / 302 Found**: Check server configuration files (e.g., .htaccess, nginx.conf) for redirect rules.
- **303 See Other / 307 Temporary Redirect**: Ensure correct URLs and redirect logic in your application.
- **304 Not Modified**: Make sure that cache headers are correctly implemented and that the client correctly uses caching.

Debugging 4xx Codes

- **400 Bad Request**: Inspect the request syntax, parameters, and data being sent to ensure they meet the server's requirements.
- **401 Unauthorized / 403 Forbidden**: Check authentication credentials and permissions settings.
- **404 Not Found**: Verify the requested URL exists on the server and check routing or URL rewriting rules.
- **405 Method Not Allowed**: Confirm that the HTTP method (GET, POST, PUT, DELETE) used in the request is supported by the endpoint.

Debugging 5xx Codes

- **500 Internal Server Error**: Review server logs to identify script or configuration errors. Ensure sufficient server resources (CPU, memory).
- **501 Not Implemented**: Ensure that the requested feature or HTTP method is supported by the server.
- **502 Bad Gateway / 504 Gateway Timeout**: Check the upstream server's health and network connectivity between servers. Ensure proper configuration of proxies or load balancers.
- **503 Service Unavailable**: Look for server overload, maintenance schedules, and ensure adequate server resources. Configure retry mechanisms and rate limiting.

General Debugging Steps

1. **Check Logs**: Review server logs (e.g., Apache, Nginx, application logs) to get detailed error messages and stack traces.
2. **Inspect Network Traffic**: Use tools like curl, Postman, or browser developer tools to inspect HTTP requests and responses.
3. **Verify Server Configuration**: Review server configuration files for syntax errors or misconfigurations.
4. **Check Application Code**: Look for bugs, unhandled exceptions, or incorrect logic in your application code.
5. **Monitor Server Resources**: Ensure that your server has enough CPU, memory, and disk space to handle requests.
6. **Review Security Settings**: Ensure proper authentication, authorization, and firewall settings to avoid unauthorized access or blocking legitimate requests.
7. **Use Debugging Tools**: Utilize APM (Application Performance Management) tools like New Relic, Datadog, or others to monitor application performance and identify issues.
8. **Consult Documentation**: Refer to the documentation of your web server, frameworks, and APIs for guidance on handling specific status codes.
