Sirma

e: office@sirma.bg
m: +359 2 9768310
a: 135 Tsarigradsko Shosse, blvd.
1784 Sofia, Bulgaria

# Git - Exercises

## Contents



## 1. Adding Files to the Repository

Your task is to add a new file to the repository and check the status of the Git repo to ensure the changes are staged.

1. Create a new file called 'notes.txt' and write "Git is fun!" in it.
2. Use git status to see the current state of the repository.
3. Add 'notes.txt' to the staging area.
4. Use git status again to verify that 'notes.txt' is staged.

## 2. Basic Commit



e: office@sirma.bg
m: +359 2 9768310
a: 135 Tsarigradsko Shosse, blvd.
1784 Sofia, Bulgaria

After adding files or making changes, it's time to commit them. Let's practice that.

1. Edit 'notes.txt' and append "Learning new commands." to it.
2. Use git status to see the current state.
3. Add 'notes.txt' to the staging area.
4. Commit the changes with a message "Updated notes.txt with new content."
5. Use git log --oneline to see the commit history.

## 3. Working with Branches

Branches are a core concept in Git. Let's create a new branch, make changes, and then switch back to the main branch.

1. Create a branch named 'new-feature' and switch to it.
2. In 'new-feature', create a new file named 'feature.txt' and write "This is a new feature." in it.
3. Commit the changes with the message "Added feature.txt."
4. Switch back to the master/main branch.

Does 'feature.txt' exist in the master/main branch? Check with ls command.

## 4. Detached Head

When we end up in a "detached head" state, surely it`s a scary situation, but as we know, Git is not scary.

We want to have a branch called the-beginning that is made from the first commit with message A.

Can you do this by first causing a detached head?

1) Run git status and git log --oneline --graph --all to see what is going on.
2) Checkout the commit with message A.
3) Create the-beginning branch.

## 5. 3-Way Merge

You again live in your own branch, this time we will be doing a bit of juggling with branches, to show how lightweight branches are in git.

1) Create a branch called greeting and switch to it.
2) Edit the greeting.txt to contain your favorite greeting.
3) Add greeting.txt files to the staging area

e: office@sirma.bg
m: +359 2 9768310
a: 135 Tsarigradsko Shosse, blvd.
1784 Sofia, Bulgaria

4) Commit
5) Switch back to the master branch
6) Edit file README.md with information about this repository
7) Add the README.md file to staging area and make the commit
8) What is the output of git log --oneline --graph --all?
9) Diff the branches
10) Merge the greeting branch into master
11) What is the output of git log --oneline --graph --all now? Observe the extra merge commit created with the message "Merge branch 'greeting'".

## 6. Merge Conflict

Figure out how to merge the content added on feature1 with the content on master.

Both changes need to be on master when you're done, or you can create your own feature and merge both.

1) Use git merge to bring the changes from feature1 on to master.
2) What does git status now report.
3) Fix the conflict with your favorite editor.
4) Follow the instructions in git status to complete the merge.
5) What does git log --oneline --graph --all show?

## 7. Basic Revert

In this task a few changes snuck in, that we'd like to get out. Our history is public, so we can't just change it. Rather we need to use revert to remove the unwanted changes in a safe way.

1) Use git log --oneline to look at the history
2) Use cat to view the content of greeting.txt
3) Use git revert on the newest commit, to remove the changes the last commit added
4) Use git log --oneline to view the history
5) Did the revert command add or remove a commit?
6) Use cat to view the content of greeting.txt
7) Use ls to see the content of the workspace
8) Use git log --oneline to find the sha of the commit adding credentials to the repository
9) Use git revert to revert the commit that added the credentials
10) Use git log --oneline to view the history
11) Use ls to see the content of the workspace
12) How many commits were added or changed by the last revert?
13) Use git show with the sha of the commit you reverted to see that the credentials file is stilll in the history
14) As you have now reverted the credentials file, so it is removed from your working directory, is it also removed from git?

e: office@sirma.bg
m: +359 2 9768310
a: 135 Tsarigradsko Shosse, blvd.
1784 Sofia, Bulgaria

# 8. Reverted Merge

Your Project uses the cool library-1.2.3 for its development, which is maintained by another team.

At some point, your team integrates a new version library-1.2.4. Because you are best practitioner, you do this on a branch integrate-library-1.2.4.

Unfortunately, you discover after your merge that the library has a bug, which has to be fixed by this other team. To prevent the bug from being released into production, you decide to revert the merge commit.

1) Revert the merge commit. To resolve the conflict, you need to determine what features should be included in mymodule.txt.
   - You can tell whether feature X should be included or not by when it was commited.
   - You may assume that feature Y is also working with the old library version.
2) Take the role of the library team and fix the bug in the library on the integrate branch, e.g. change lib.txt.
3) Next we explore how you can get the changes from the branch into the master again. First try to merge to see what happens. The lib.txt file changes as expected, but 'mymodule.txt does not. For an in depth discussion of the reason why, consult this gist: Reverting a faulty merge.

reverting a merge commit also undoes the data that the commit changed, but it does absolutely nothing to the effects on history that the merge had.

So the merge will still exist, and it will still be seen as joining the two branches together, and future merges will see that merge as the last shared state

4) Undo the merge with a reset --hard
5) Revert the revert and try the merge again. This time it works.

## Useful commands

```
git revert -m 1 <merge-sha1>

git revert --continue

git switch <branch-name>

git merge <branch-name>

git reset --hard <sha1>

git revert <sha1>
```

# 9. Restoring Files

e: office@sirma.bg
m: +359 2 9768310
a: 135 Tsarigradsko Shosse, blvd.
1784 Sofia, Bulgaria

We all make mistakes. Sometimes we make changes we'd rather not have made, or accidentally stage changes we didn't intend to stage. This is where git restore comes into play.

1) Call git status
   What changes were made to this repository?
2) Restore the foo.txt file using git restore foo.txt
3) Call git status again
   What happened to foo.txt?
4) Unstage the changes to bar.txt using git restore --staged bar.txt
5) Call git status once more
   What happened to bar.txt?
6) Restore the bar.txt file using git restore bar.txt
7) Call git status once more
   What happened to bar.txt?
8) Call git log --oneline
   Do you spot the tag?
9) Restore foo.txt's contents to their previous version using git restore -s v1.0.0 foo.txt
10) Call git status one last time
    What happened to foo.txt?

### Useful commands

```
git restore

git restore --staged
```

## 10. Reorder History

The commits here have obviously been made by a mad man. Fortunately, they contain useful information - it's just that the history is weird. You should fix this such that our git log looks great!

Reorder the history such that it actually makes sense - add the files in the order that matches their name and remove unneeded commits.

1) Use git log --oneline --graph to view the commits.
2) Also try git reflog to view the commits. git reflog defaults to git reflog show and this is an alias for git log -g --abbrev-commit --pretty=oneline
3) Use git rebase -i <after-this-commit> to reorder the commits. There are comments in the file you edit that explain the commands available.
4) Use git log --oneline --graph to view the result.

### Bonus task*



e: office@sirma.bg
m: +359 2 9768310
a: 135 Tsarigradsko Shosse, blvd.
1784 Sofia, Bulgaria

If you`ve chose to delete file3, I have a nice surprise, it was really important file, and you need to restore it right away. Try git restore -s <source> <file>

Because vim editor is not easy to use, you can change your configuration, check here: https://docs.github.com/en/get-started/getting-started-with-git/associating-text-editors-with-git

## 11. Interactive rebase with --autosquash option

You have worked on a new feature called Hello World. This feature ends up being complete with both documentation and unit test, but there is a typo in the documentation.

You need to fix it and then rebase to have a beautiful history.

Luckily, we have a release tag v0.0 from just before we started the feature.

There is a way to easily fix it with advanced options for git commit and git rebase.

1) Explore the repo and the history so you know when the documentation file was added.
2) Fix README.md file and add it.
3) Add your commit by using git commit --fixup=<commit id to be fixed>.
4) Use git rebase --autosquash --interactive v0.0 to view the rebase recipe automatically generated.
5) Use git log to view your new beautiful history.

## 12. Advanced interactive rebase

You have worked on a new feature called Hello World. This feature ends up being complete with both documentation and unit test, but there are a few problems. The history looks really messy, with lots of small half-finished steps, and there are things included that should never have been there.

You should fix this such that your git log looks great!

To do this we will use our good friend git rebase --interactive

Luckily, we have a release tag v0.0 from just before we started the feature.

As this is an advanced exercise, there are no specific steps to follow and no single solution.

1) Explore the repo and the history so you know what happened
2) Use git rebase --interactive v0.0 to let you edit the "recipe" for the entire feature development.

e: office@sirma.bg
m: +359 2 9768310
a: 135 Tsarigradsko Shosse, blvd.
1784 Sofia, Bulgaria

3) Clean up the history such that it actually makes sense. Try to use as many of the rebase "features" (e.g. reword, squash, fixup, drop) as possible. You decide yourself if you want to rewrite the whole thing in one go, or apply a few changes first, then run a new git rebase --interactive v0.0 to keep cleaning.

### Useful commands

```
ls -l # list files

tail -n +1 * # show content of all files

git log --stat # log which files changed

git log --patch # log with diff

git rebase -i <ref> # run the interactive rebase back to
```

## 13. Basic Cherry Pick

In this task we want have two branches, master and feature. We have worked and progressed on both branches seperately. However, there are a few changes on the feature branch that we want to take and add onto the master branch. Without getting the entire changeset from the feature branch.

Git has functionality for this "take-just-these-changes" and it is called cherry-pick. You tell Git which commits you would like to cherry pick and Git will add those commits onto your branch's commit history.

Git can cherry pick either a single commit or a range of commits from a branch.

We currently have this git history in our exercise repository:

```
A - B - C - D       master
     \
      E - F - G - H feature
```

As you can see the feature branch and the master branch have progressed with different commits. We want to cherry pick the commits F and G and add them onto the master branch, so that our Git history looks like this:

```
A - B - C - D - F - G master
     \
      E - F - G - H feature
```

1) Use git log --oneline --graph --all to look at the history
2) Use cat to view the content of names.txt. This file is changed in commit F
3) Use cat to view the content of sentence.txt. This file is changed in commit G

e: office@sirma.bg
m: +359 2 9768310
a: 135 Tsarigradsko Shosse, blvd.
1784 Sofia, Bulgaria

4) Use git cherry-pick <commit_hash_F> to cherry pick just the F commit onto your branch
5) Use git log --oneline to see the change to the history and that commit F should now be the newest commit on the master branch
6) Use cat to view the content of names.txt look how it has now changed!
7) Use git reset --hard HEAD^ to delete that cherry picking from the history so that we can now try again and cherry pick a range of commits
8) Use git log --oneline --graph to check the the cherry picked commit is now removed from the branch
9) We are now essentially back to where we began, now use git cherry-pick <commit_hash_F>^..<commit_hash_G> to cherry pick the range of commits from F to G (the two commits). Pay close attention and do not forget the caret ^ symbol after the first commit hash (see the section Useful Note below to understand why this is needed)
10) Use git log --oneline --graph to view the history
11) Use cat to view the contents of names.txt and sentence.txt look how they have changed!
12) How many commits were added due to the cherry pick?

**Explanation**

When using range of commits with the cherry pick command, the first commit hash specified for the oldest (left side of the range) is not actually included in the cherry pick, as in that commit is excluded but all others between and including the newest commit are.

So to bypass this issue it is useful to use the caret ^ after the first commit hash to tell Git that you want the commit BEFORE this commit, therefore including it in the cherry pick process.

For example

```
git cherry-pick ABCD..EFGH
```

would not include the commit ABCD, instead you should add a caret symbol to the end of the ABCD to tell git to include it, like this:

```
git cherry-pick ABCD^..EFGH
```

After the bug fix, you continue to work on the new feature. After you committed the second part of the feature, you realize that you have done your commit on the master branch instead of the feature branch.

1) Move the faulty commit from the master branch to the new-feature branch.
2) How would you also bring the bugfix to your feature branch?

Sirma

e: office@sirma.bg
m: +359 2 9768310

a: 135 Tsarigradsko Shosse, blvd.
1784 Sofia, Bulgaria