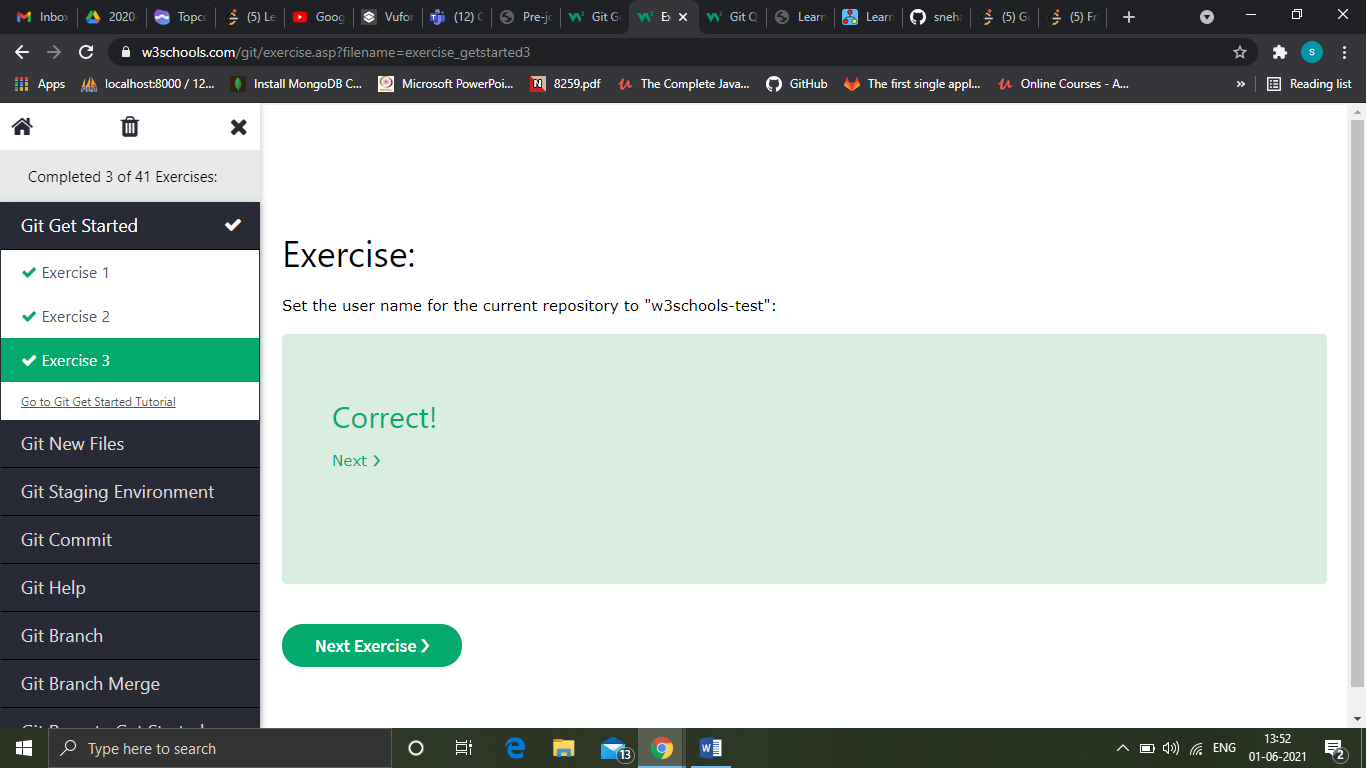
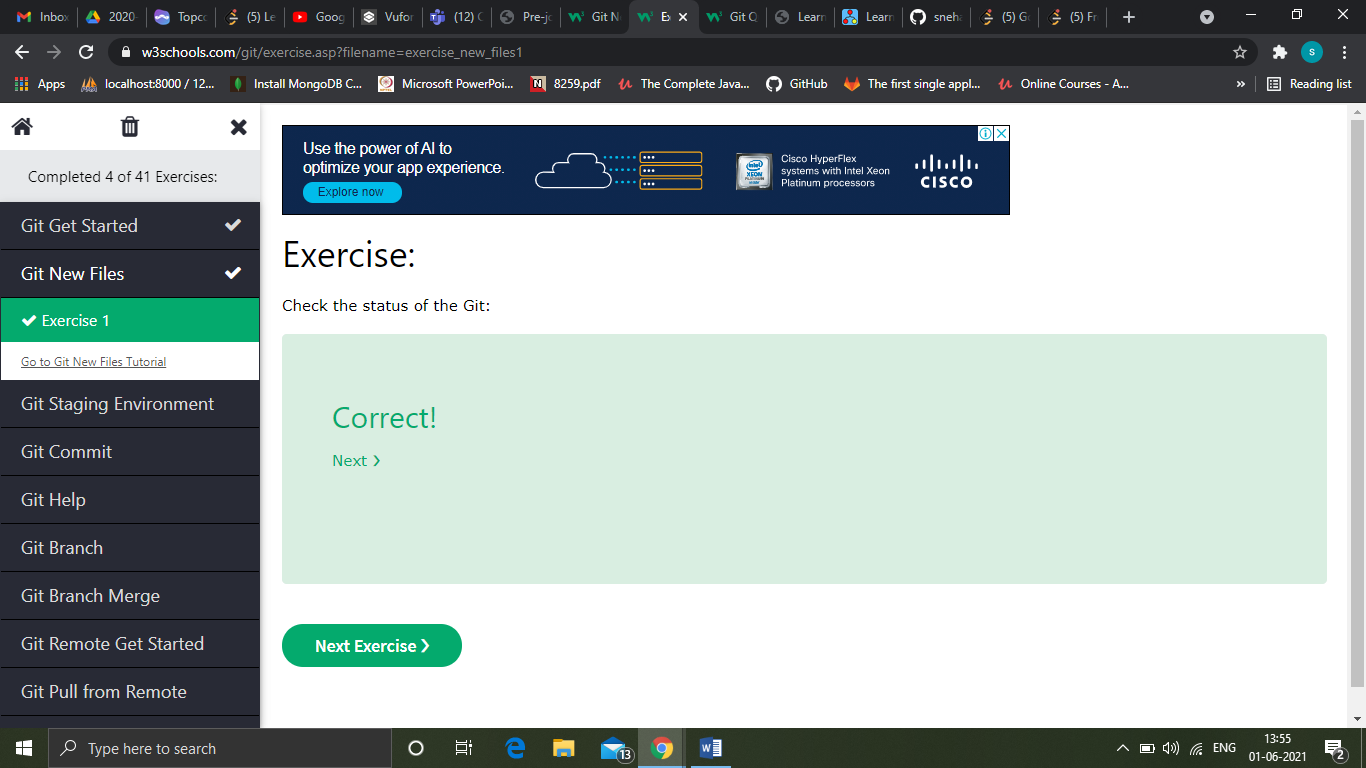
DAY 6

GIT

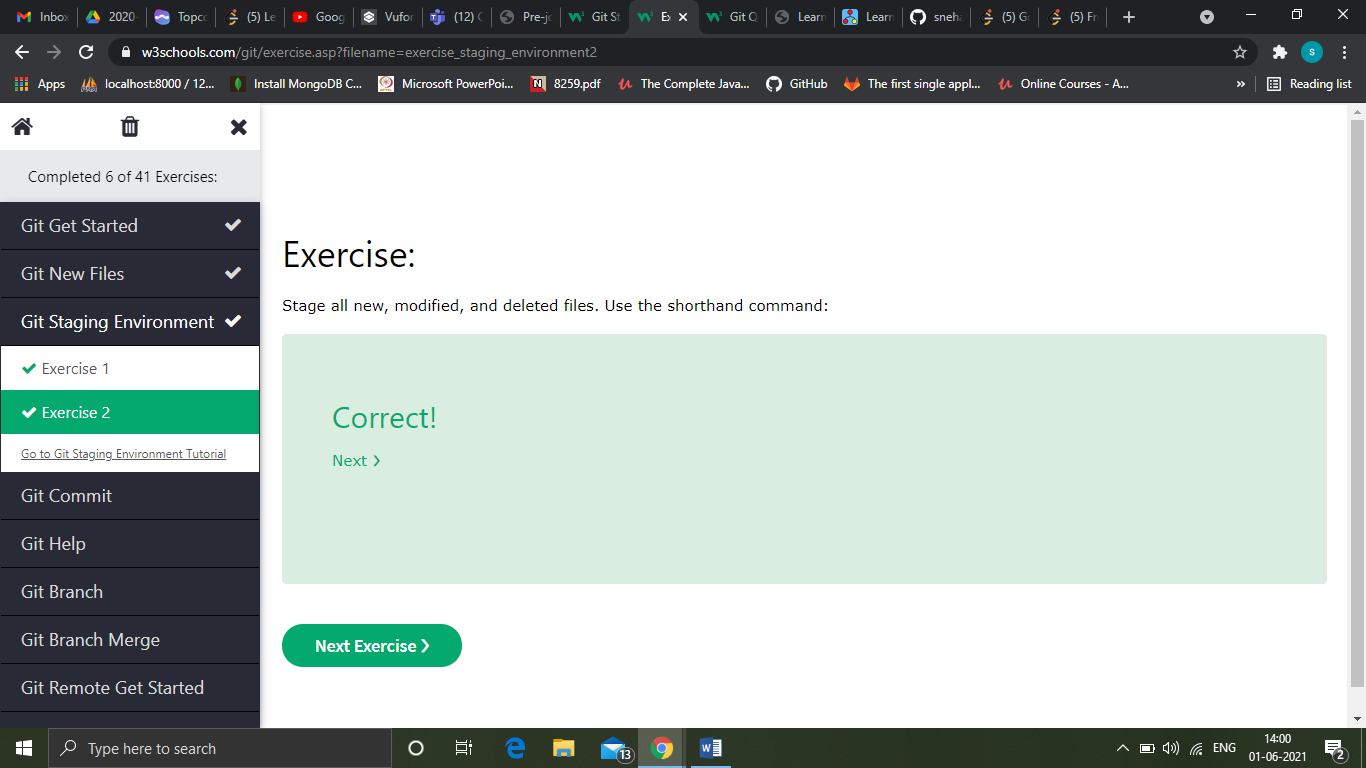
* Manage projects with **Repositories**
* **Clone** a project to work on a local copy
* Control and track changes with **Staging** and **Committing**
* **Branch** and **Merge** to allow for work on different parts and versions of a project
* **Pull** the latest version of the project to a local copy
* **Push** local updates to the main project



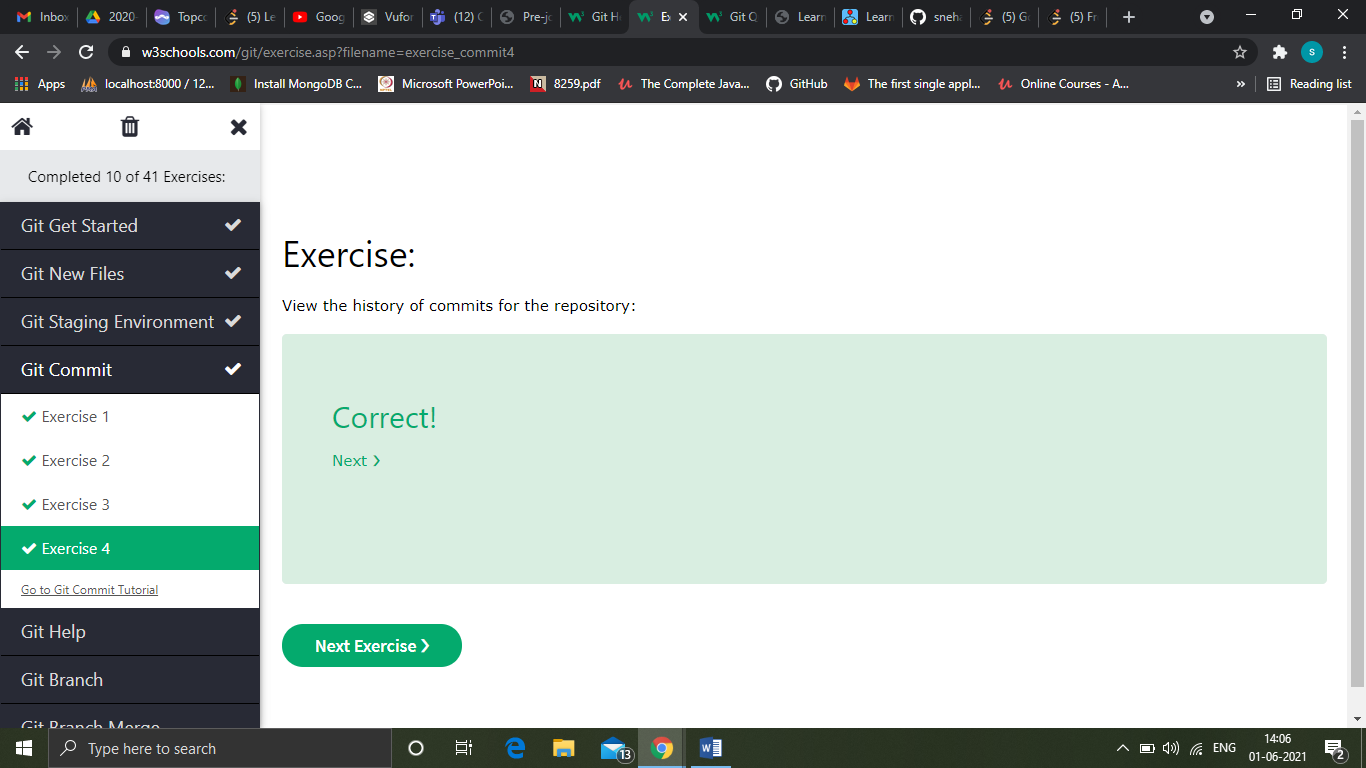
Git new files



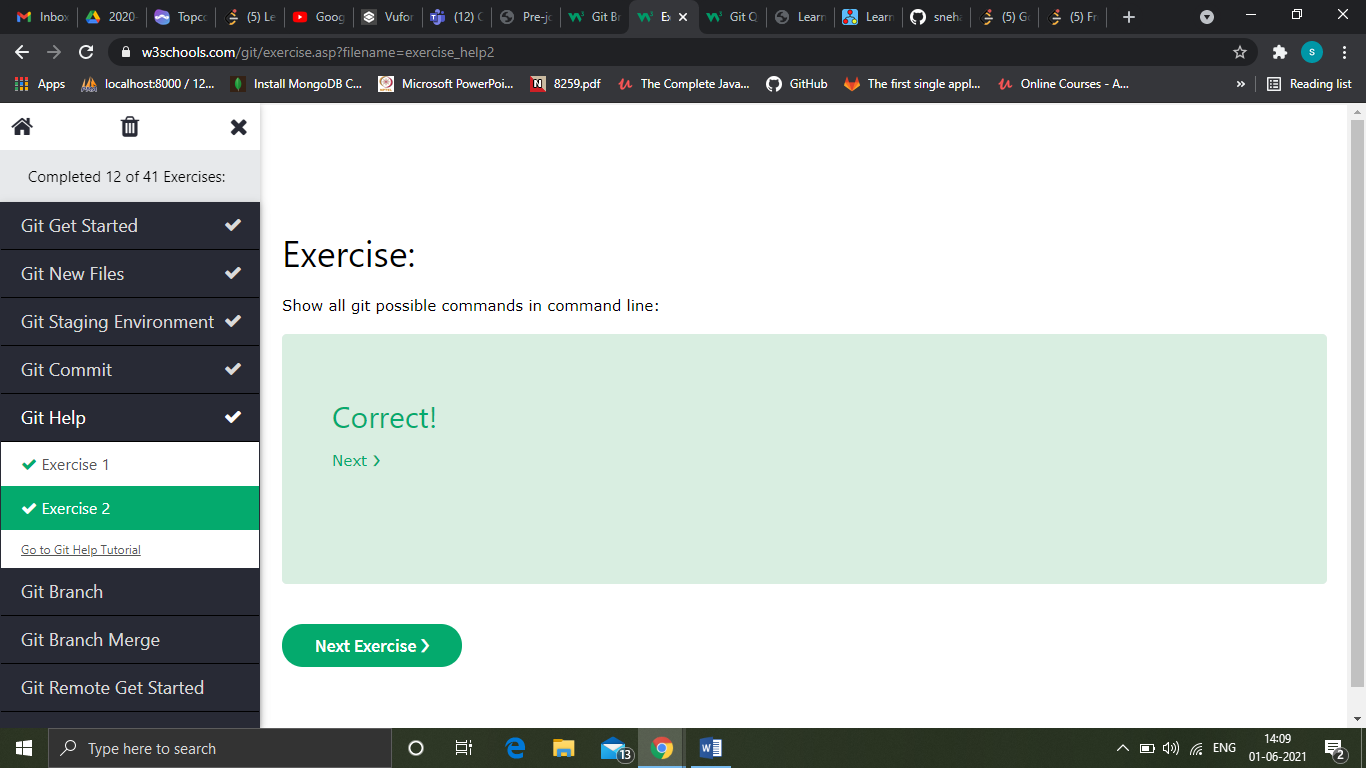
Git staging environment



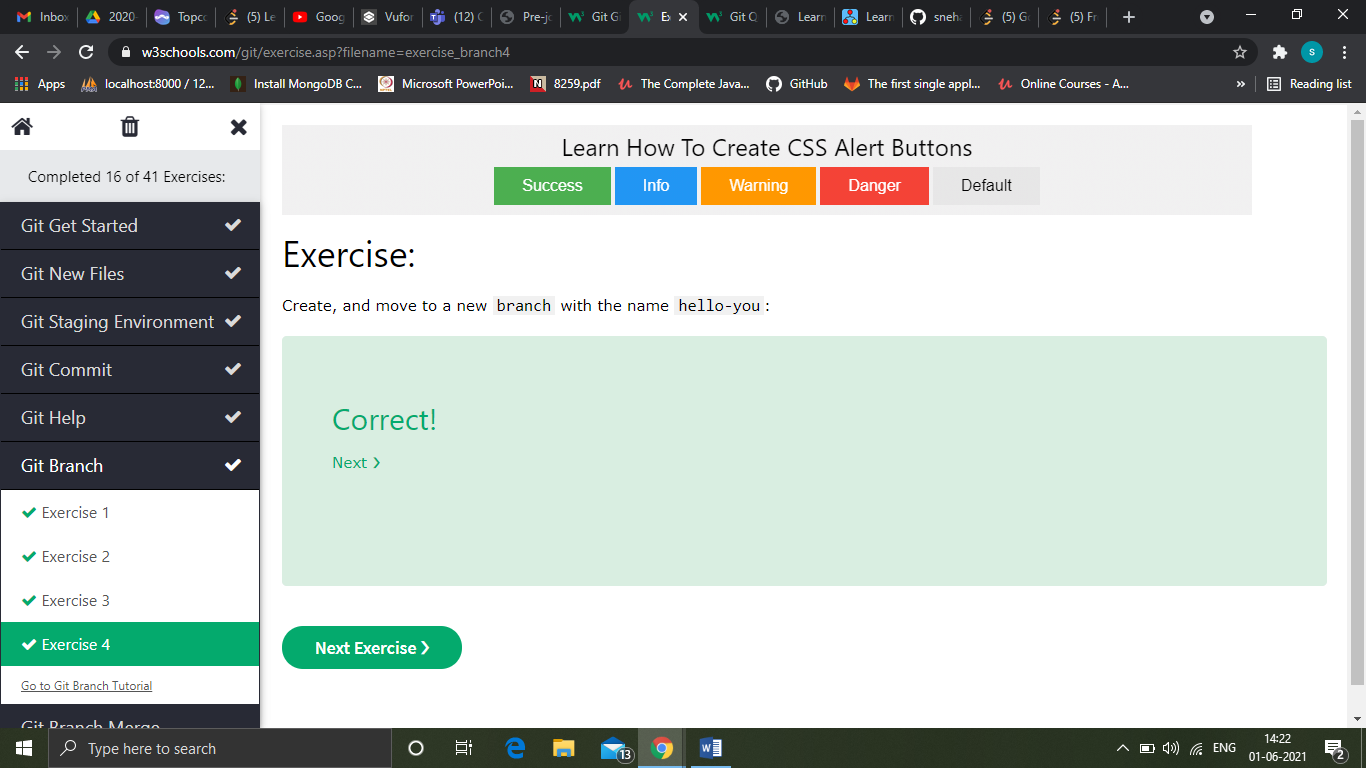
Git commit



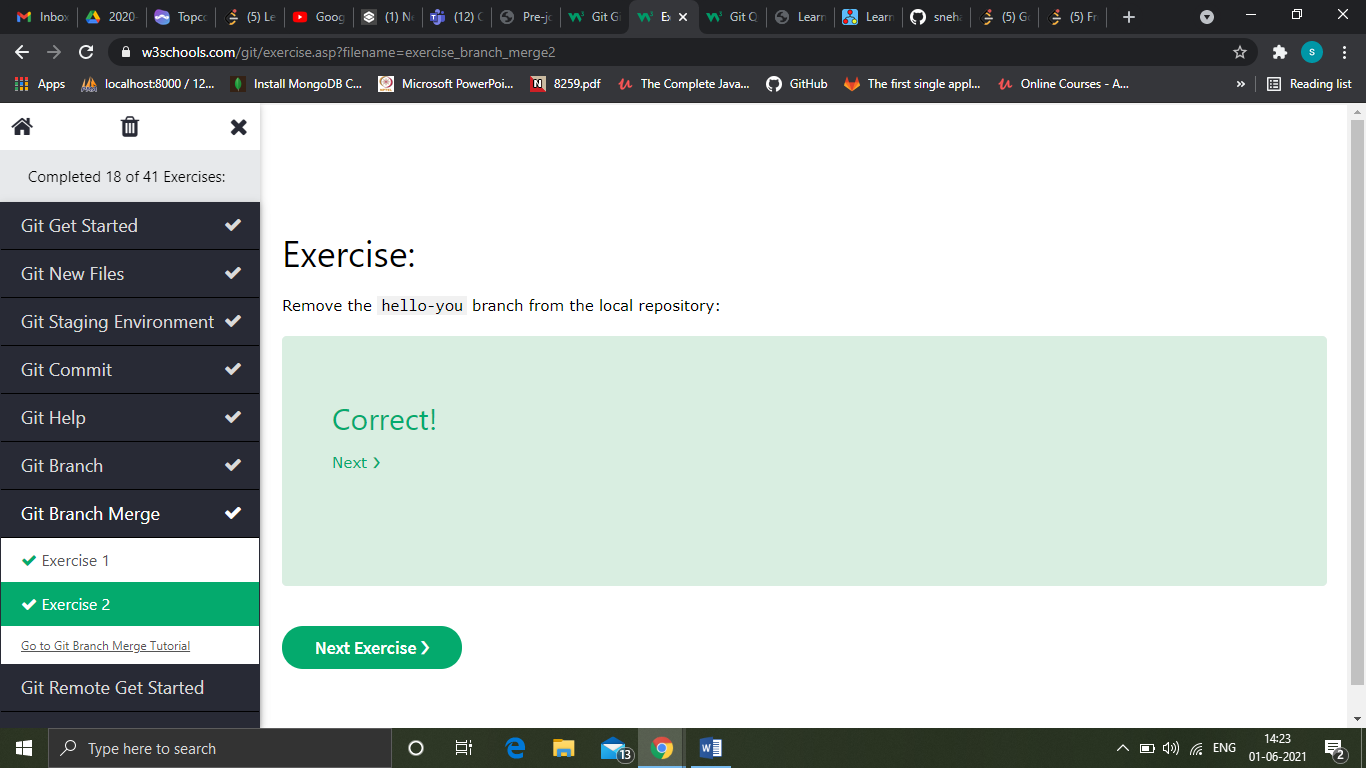
Git help



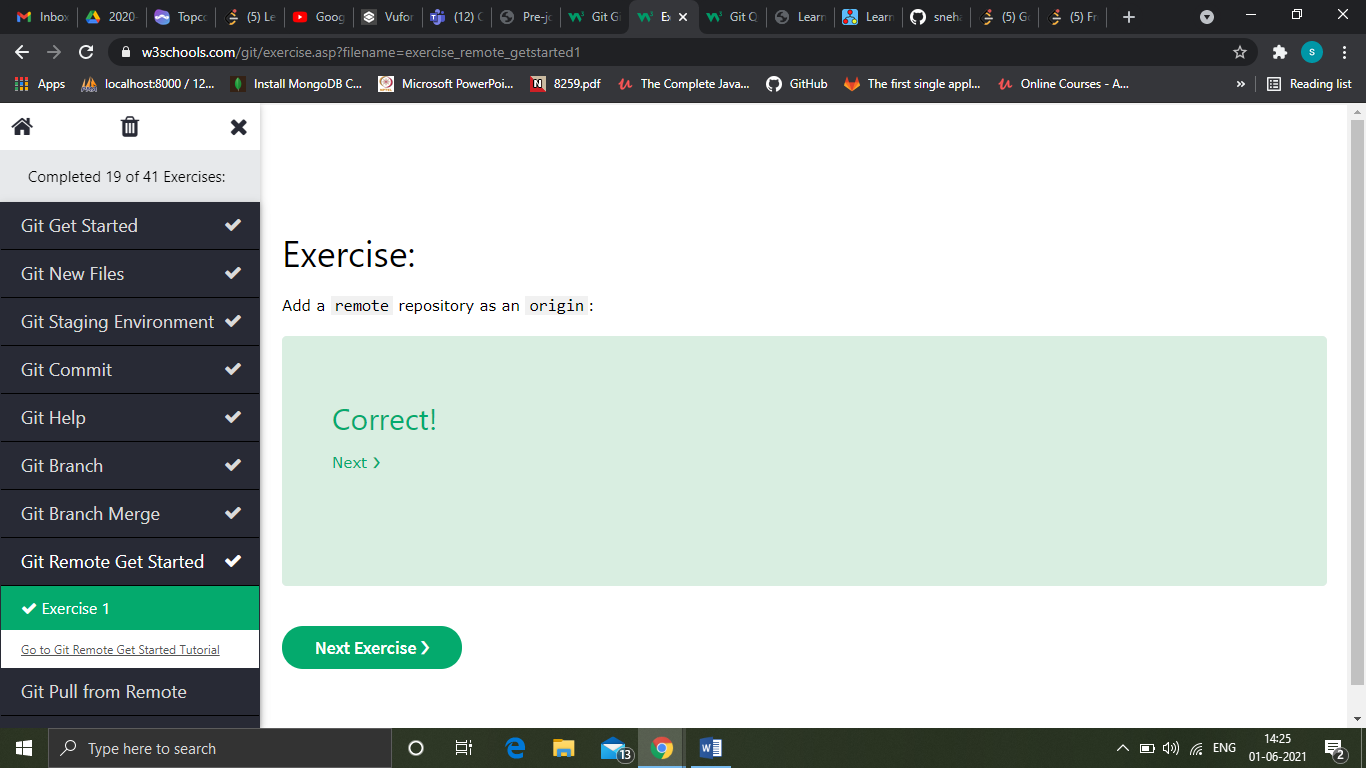
Git branch



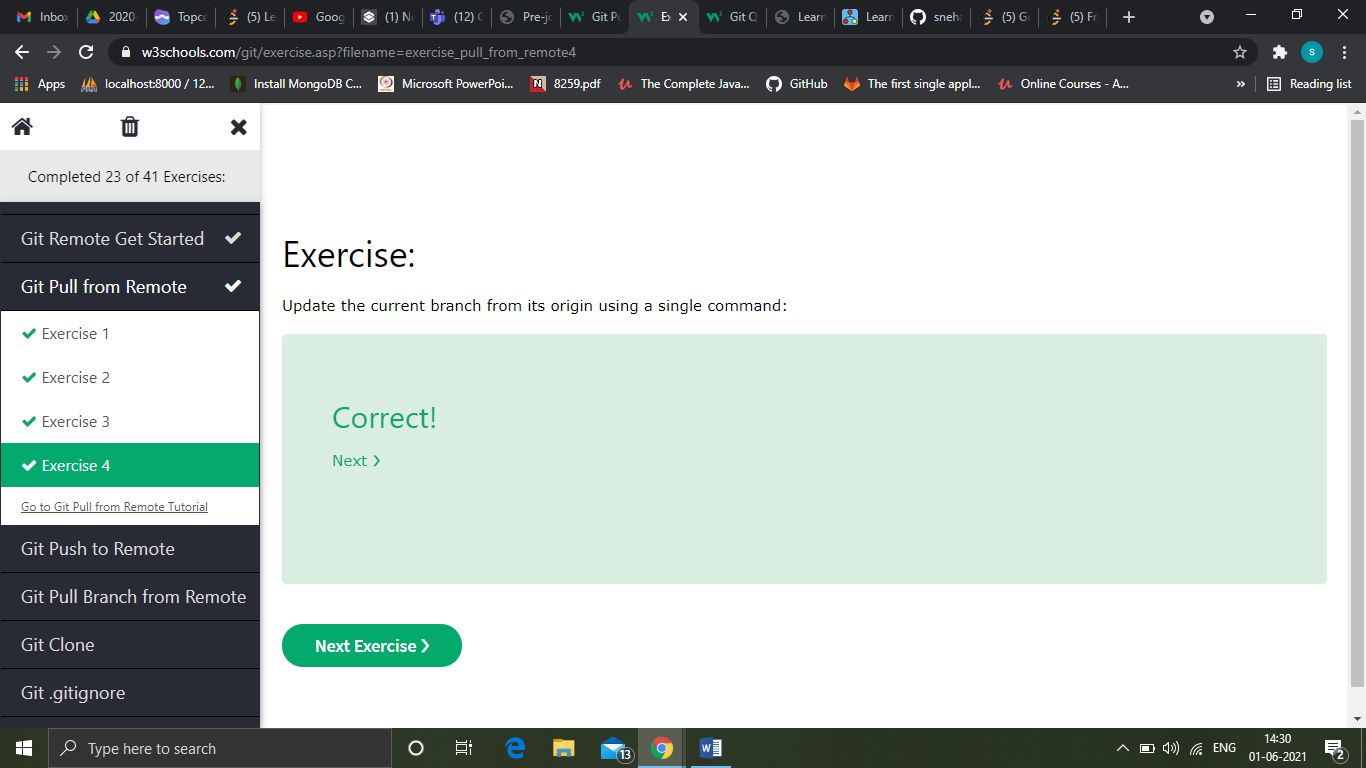
Git branch merge



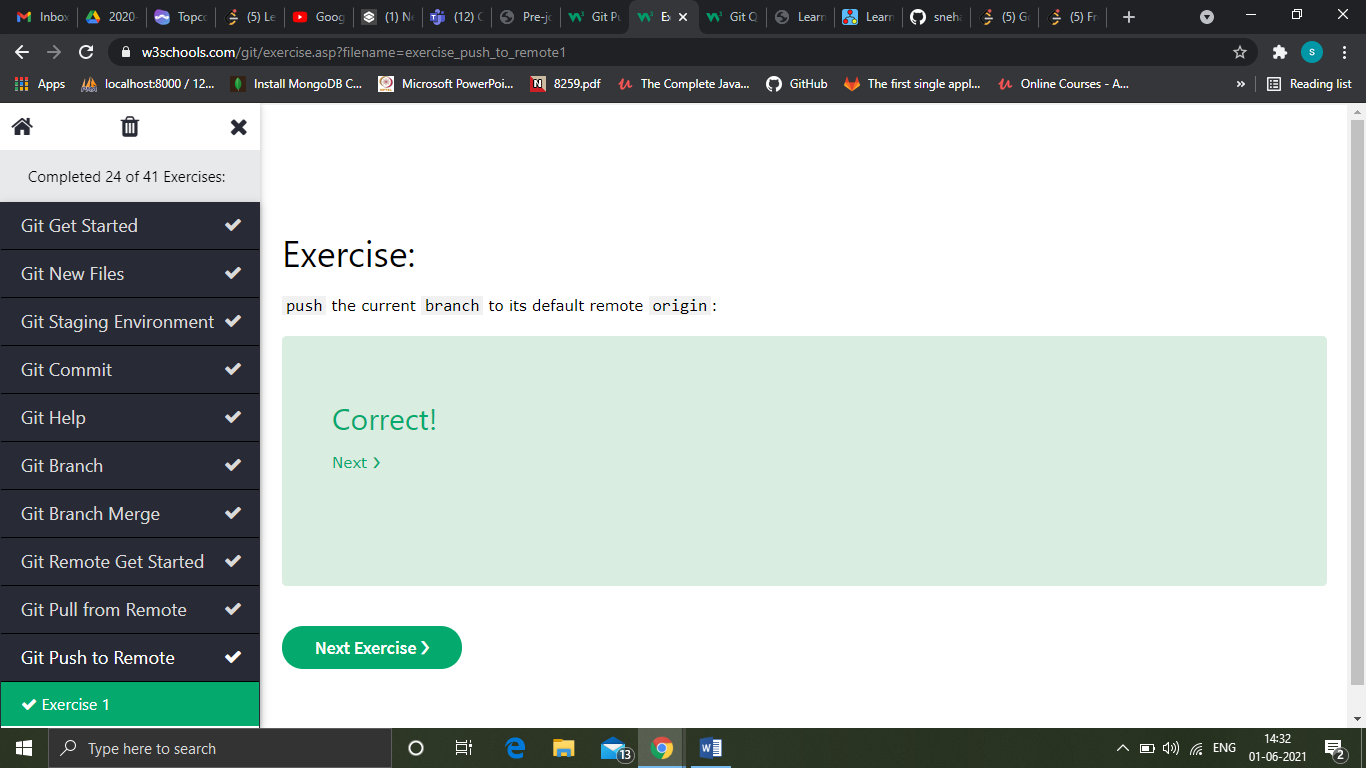
Git remote get started



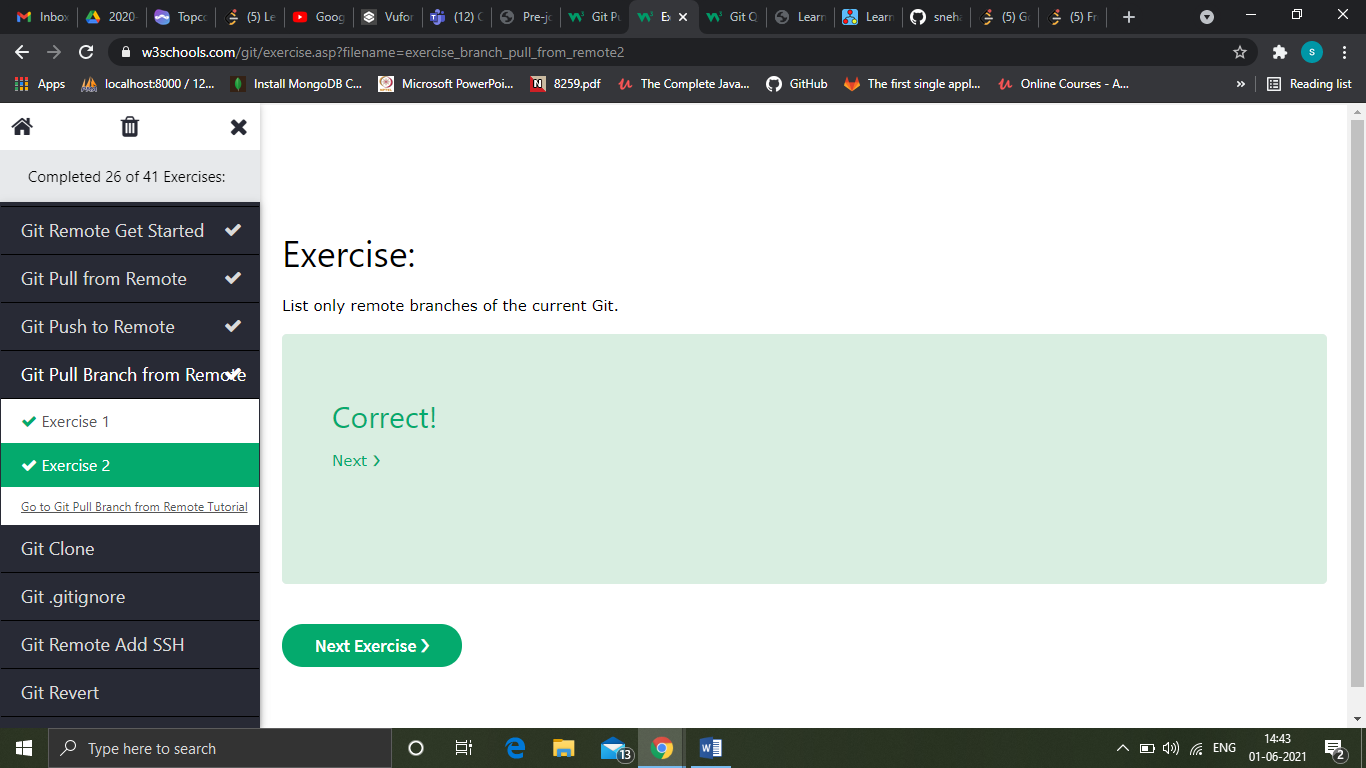
Git pull from remote



Git push to remote



Git pull branch from remote



Git contribute

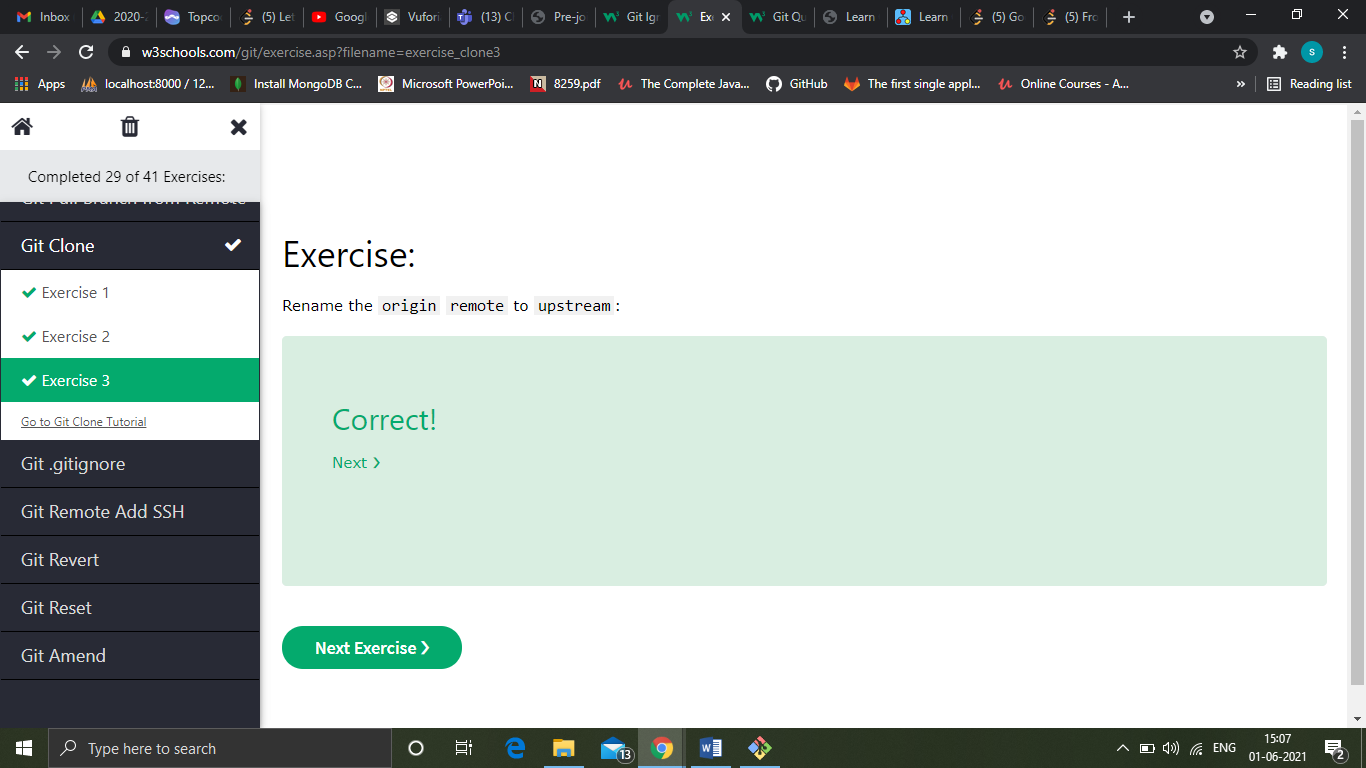
A fork is a copy of a repository. This is useful when you want to contribute to someone else's project or start your own project based on theirs.

fork is not a command in Git, but something offered in GitHub and other repository hosts.

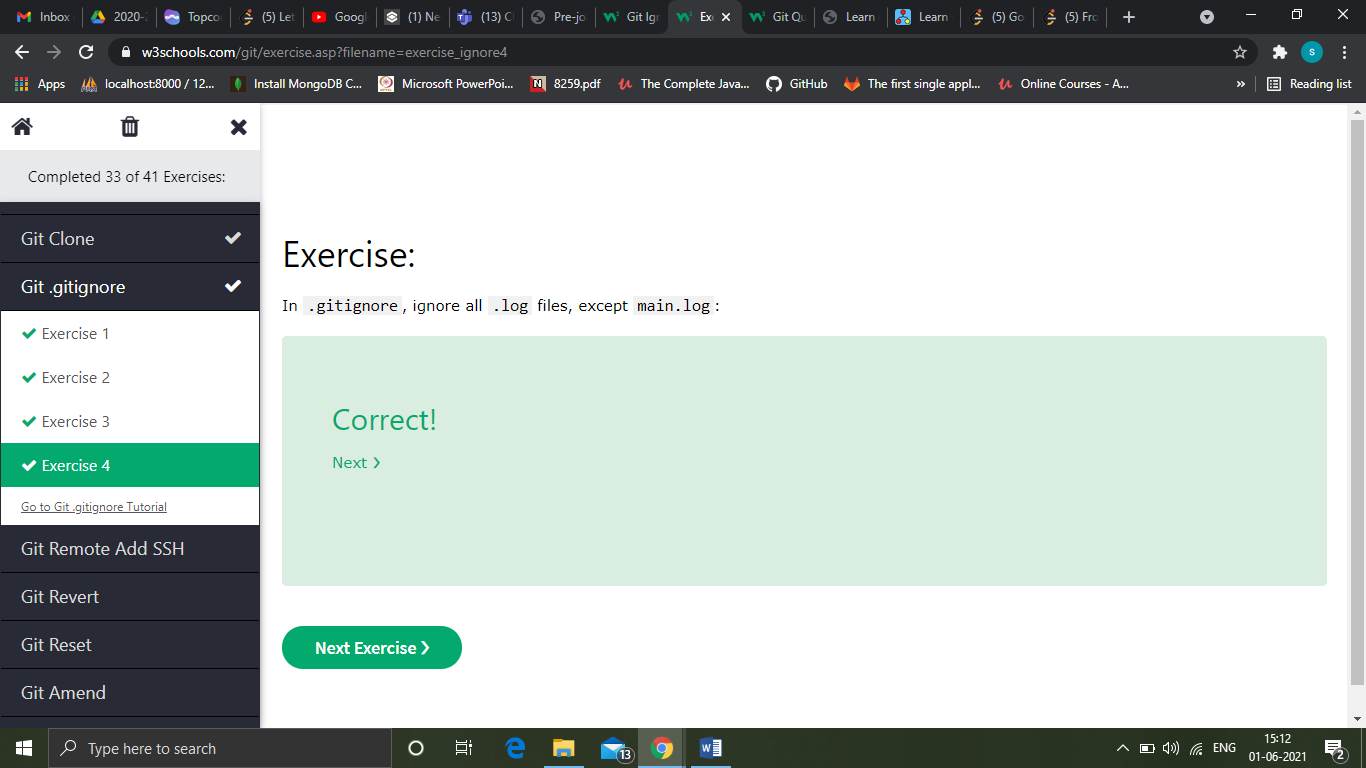
According to Git naming conventions, it is recommended to name your own repository origin, and the one you forked for upstream

Now we have 2 remotes:

* origin - our own fork, where we have read and write access
* upstream - the original, where we have read-only access



Git gitignore



Git remote add ssh

SSH is a secure shell network protocol that is used for network management, remote file transfer, and remote system access.

SSH uses a pair of SSH keys to establish an authenticated and encrypted secure network protocol. It allows for secure remote communication on unsecured open networks.

SSH keys are used to initiate a secure "handshake". When generating a set of keys, you will generate a "public" and "private" key.

The "public" key is the one you share with the remote party. Think of this more as the lock.

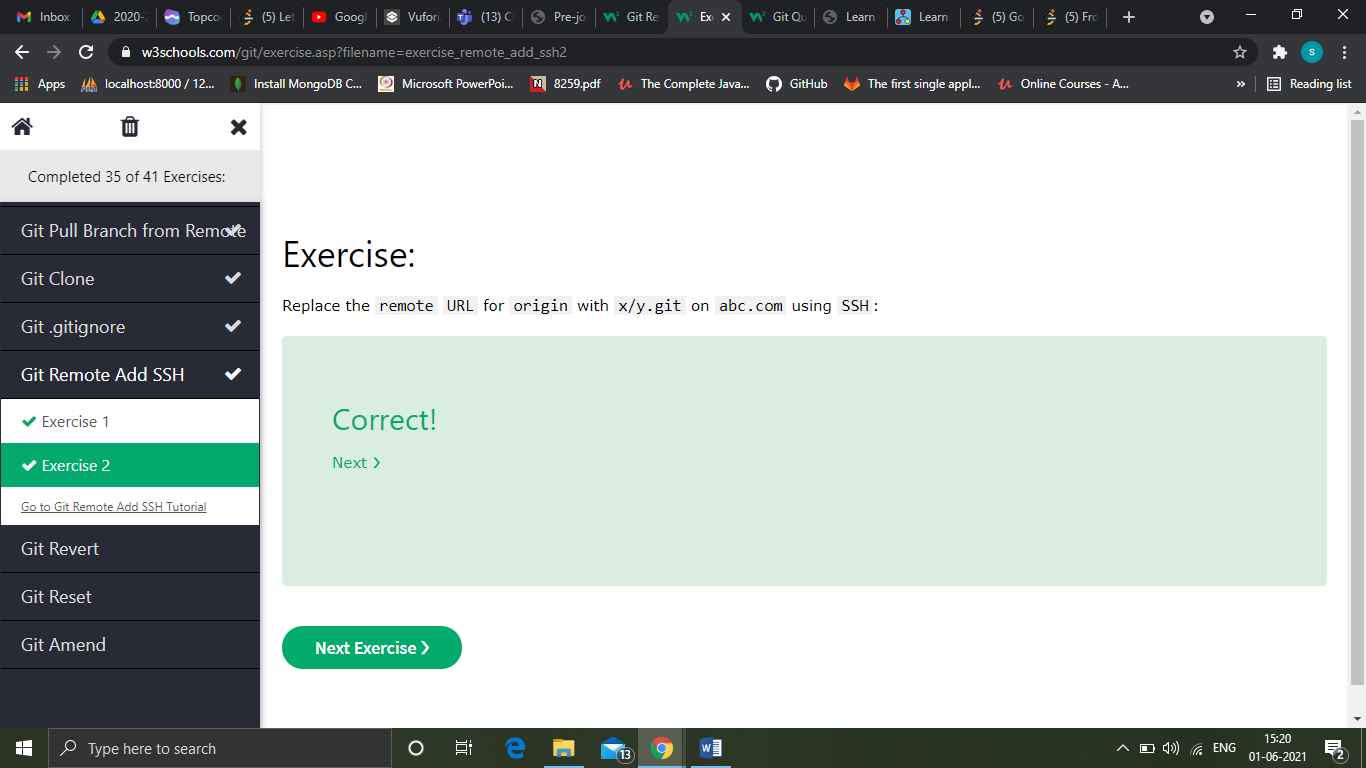
The "private" key is the one you keep for yourself in a secure place. Think of this as the key to the lock.

SSH keys are generated through a security algorithm. It is all very complicated, but it uses prime numbers, and large random numbers to make the public and private key.

It is created so that the public key can be derived from the private key, but not the other way around.

git remote add ssh-origin git@github.com:w3schools-test/hello-world.git

You can change a remote origin from HTTPS to SSH with the command: git remote set-url remote-name [git@github.com:username/repository.git](mailto:git@github.com:username/repository.git)

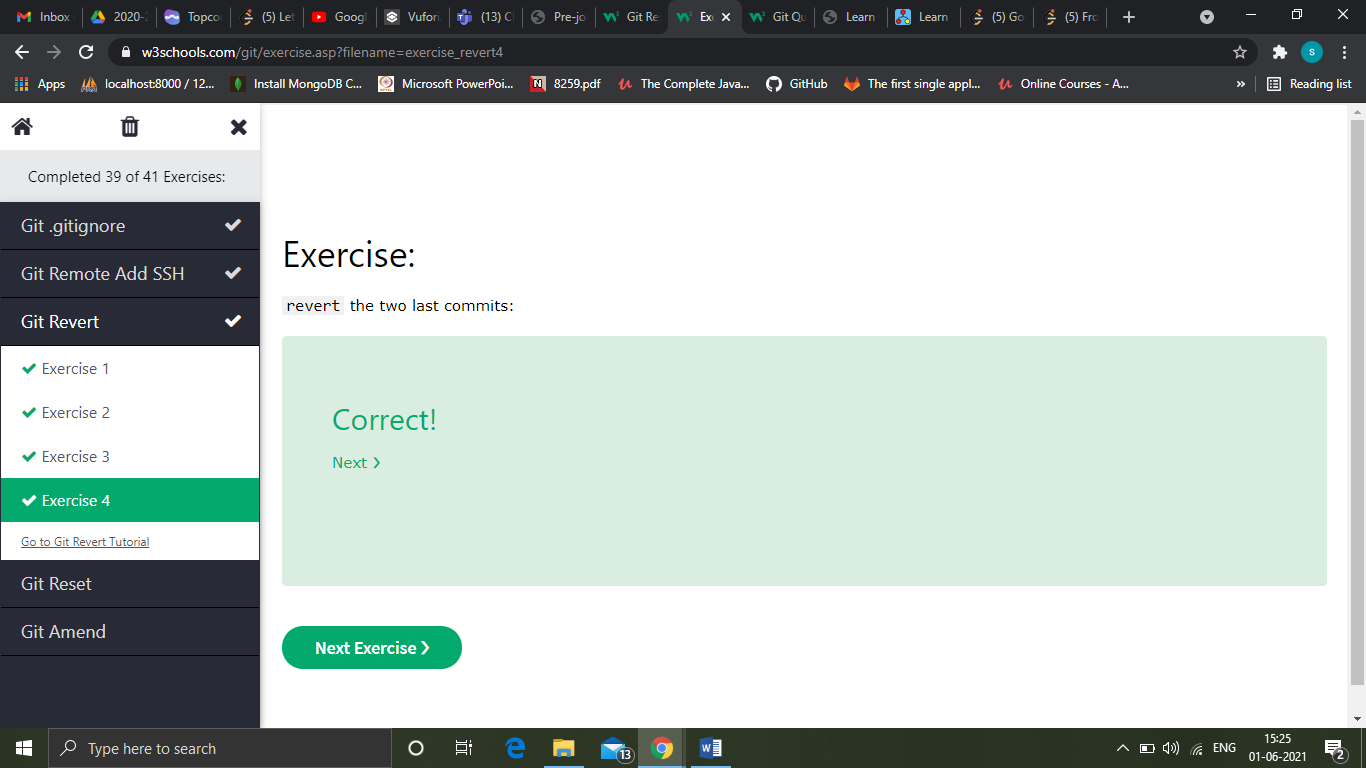


Git revert

revert is the command we use when we want to take a previous commit and add it as a new commit, keeping the log intact.

We revert the latest commit using git revert HEAD (revert the latest change,  and then commit), adding the option --no-edit to skip the commit message editor (getting the default revert message)

To revert to earlier commits, use git revert HEAD~x (*x* being a number. 1 going back one more, 2 going back two more, etc.)

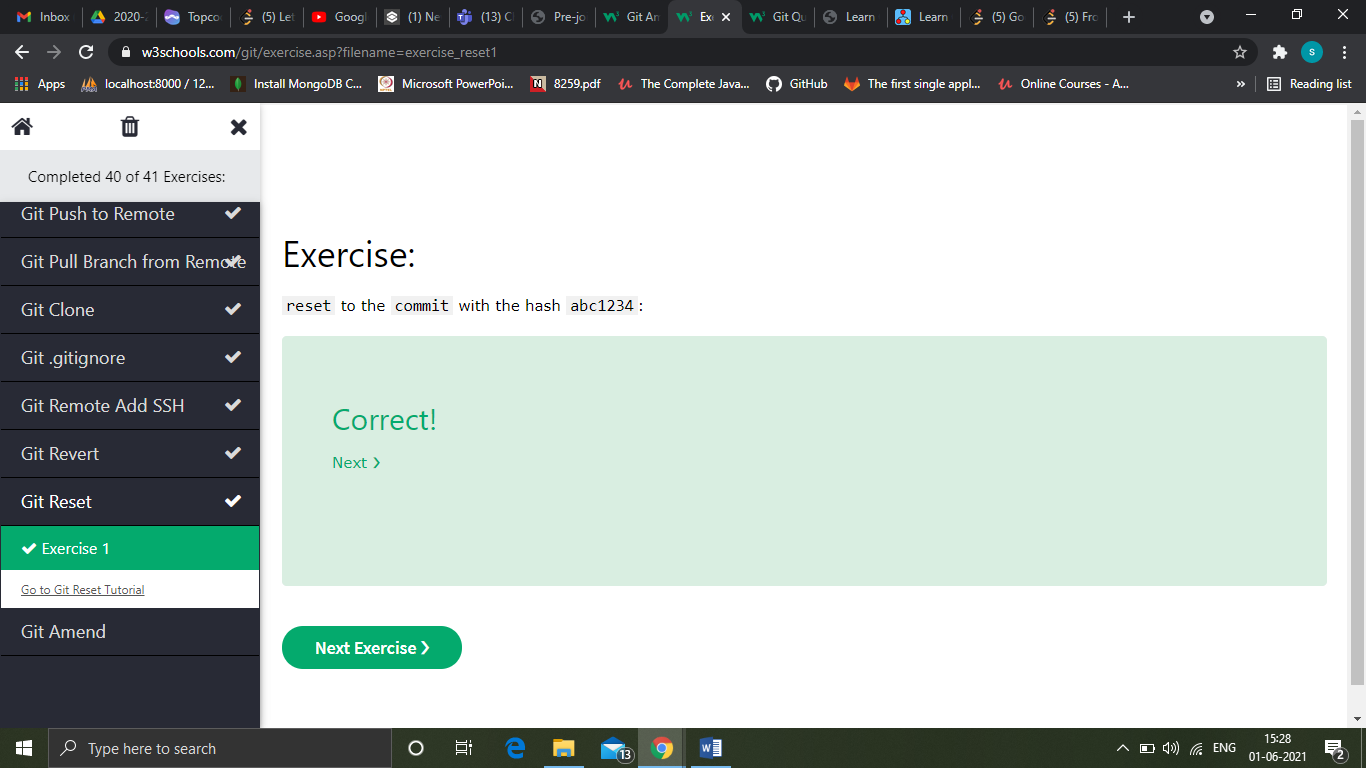


Git reset

reset is the command we use when we want to move the repository back to a previous commit, discarding any changes made after that commit

We reset our repository back to the specific commit using git reset commithash (commithash being the first 7 characters of the commit hash we found in the log)

git reset 9a9add8



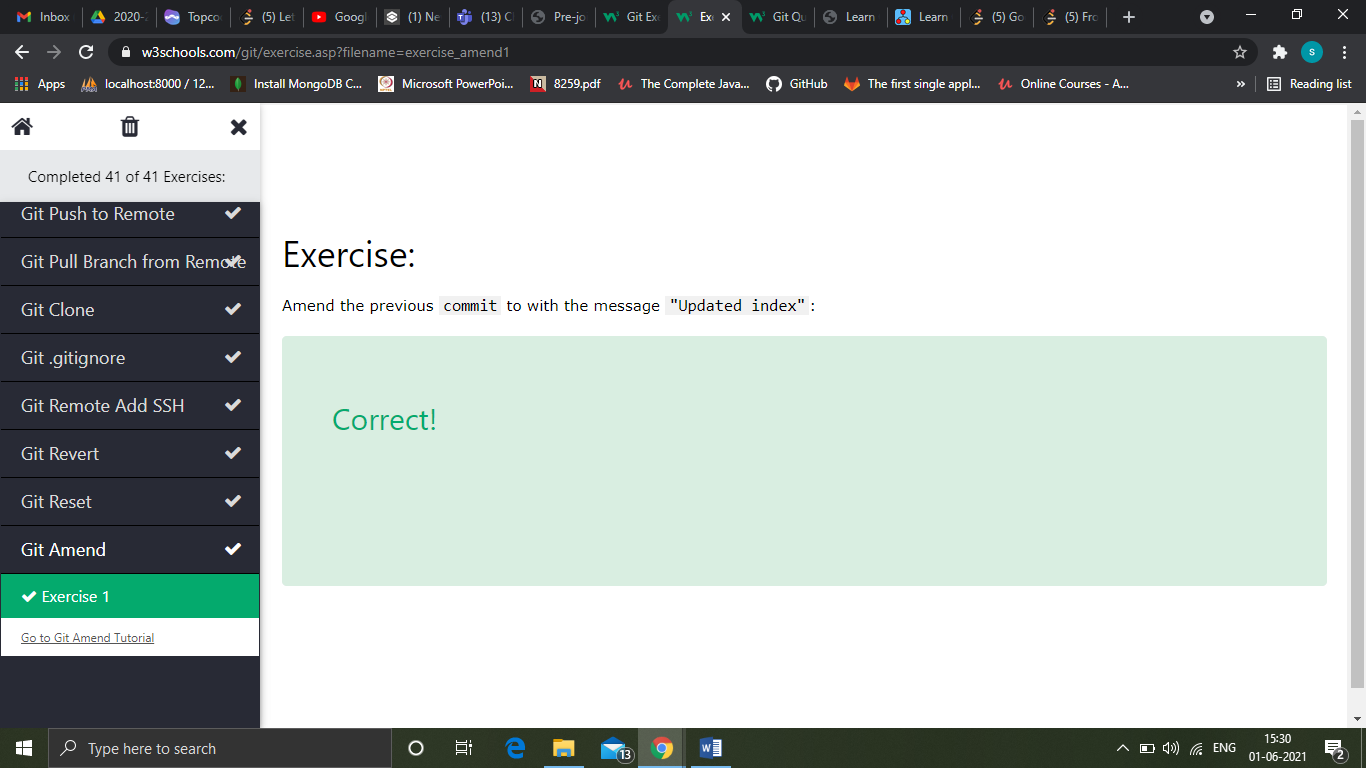
Git amend

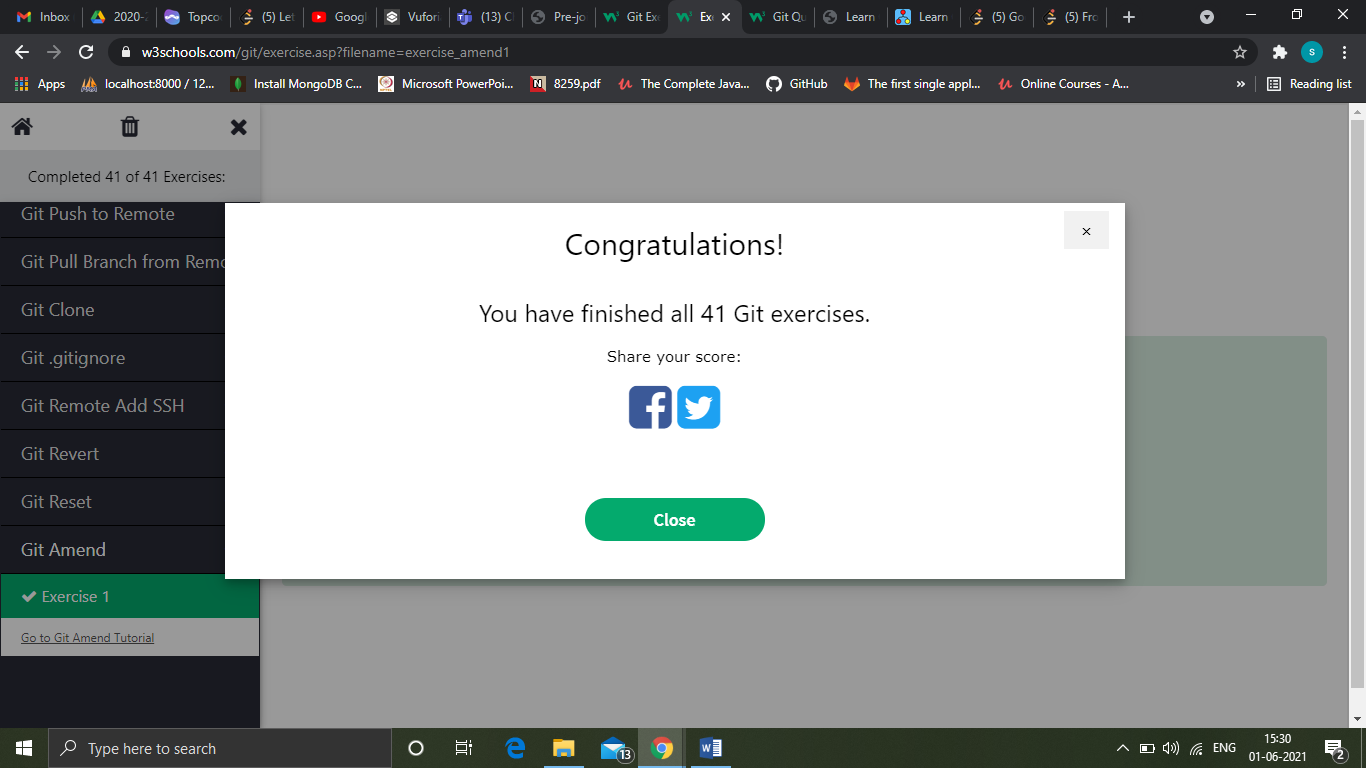
commit --amend is used to modify the most recent commit.

It combines changes in the staging environment with the latest commit, and creates a new commit.

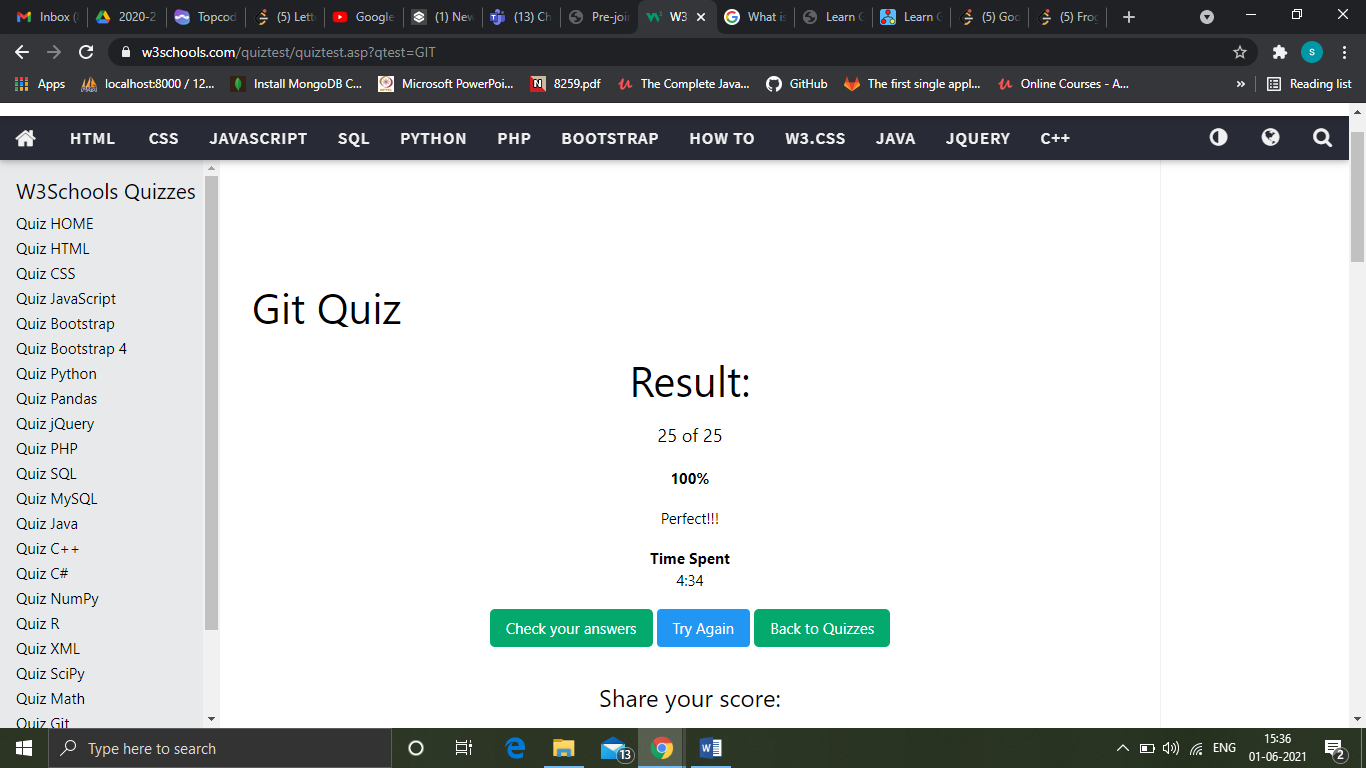
This new commit replaces the latest commit entirely.

One of the simplest things you can do with --amend is to change a commit message.





QUIZ



Course

While git log tells you the commit author and message, to view the changes made in the commit you need to use the the command git show

Like with other commands, by default it will show the changes in the HEAD commit. Use git show <commit-hash> to view older changes.

When working with Git, a common scenario is to undo changes in your working directory. The command git checkout will replace everything in the working directory to the last committed version.

If you want to replace all files then use a dot (*.*) to indicate the current directory, otherwise a list the directories/files separated by spaces.

The merge commit messages can be useful to indicate synchronisation points but they can also produce a lot of noise. For example if you're working against local branches and haven't pushed then this additional information is meaningless, and confusing, to other developers looking at the repository.

To solve this you can use git rebase instead of git merge. A rebase will unwind the changes you've made and replay the changes in the branch, applying your changes as if they happened all on the same branch. The result is a clean history and graph for the merge.

*Important* As rebase will replay the changes instead of merging, each commit will have a new hash id. If you, or other developers, have pushed/pulled the repository then changing the history can git to lose commits. As such you shouldn't rebase commits that have been made public, for example pushing commits then rebasing in older commits from a different branch. The result will be previously public commits having different hash ids. More details can be found at [The Perils of Rebasing](https://git-scm.com/book/ch3-6.html#The-Perils-of-Rebasing).

GIT BISECT

The git bisect commands allows you to do a binary search of the repository looking for which commit introduced the problem and the regression. In this step we'll find the commit which forgot HTML tags in list.html.

Git bisect takes a number of steps, execute the steps in order to see the results.

**Steps**

1. To enter into bisect mode you use the command git bisect start.
2. Once in bisect mode you define your current checkout as bad using git bisect bad. This indicates that it contains the problem
3. your searching to see when it was introduced.
4. We've defined where a bad commit happened, we now need to define when the last known good commit was using git bisect good HEAD~5. In this case it was five commits ago.
5. Step 3 will checkout the commit in-between bad and good commits. You can then check the commit, run tests etc to see if the bug exists. In this example you can check the contents using cat list.html
6. This commit looks good as everything has correct HTML tags. We tell Git we're happy using git bisect good. This will automatically check out the commit in the middle of the last known good commit, as defined in step 5 and our bad commit.
7. As we did before we need to check to see if the commit is good or bad. cat list.html
8. This commit has missing HTML tags. Using git bisect bad will end the search and output the the related commit id.

The result is that instead of searching five commits, we only searched two. On a much larger timescale bisect can save you signifant time.

Git blame

While having a "blame" culture isn't desirable, it can be useful to know who worked on certain sections of the file to help with improvements in future. This is where git blame can help.

git blame <file> shows the revision and author who last modified each line of a file.

#### Example

Running blame on a file will output who last touched each line.

git blame list.html

If we know the lines which we're concerned with then we can use the -L parameter to provide a range of lines to output.

git blame -L 6,8 list.html