**Introduction:**

*We will learn*:

1. Fundamentals (what is GIT, How to install and configure it)
2. Creating snapshots.
3. Browsing your project history
4. Branching and merging.
5. Collaborating using GitHub.
6. Rewriting History

**How to take this course**:

🡪 Watch every lesson in order.

**What is GIT**?

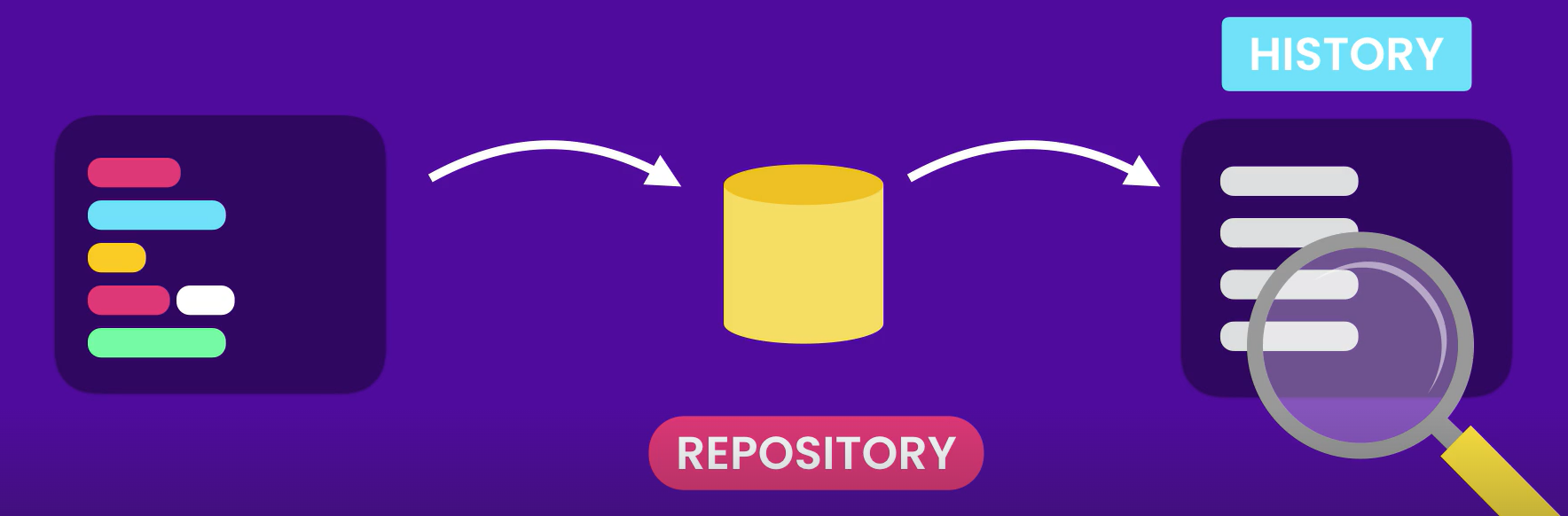
GIT is the most popular,

 In the world.

**“***A version control system records the changes made to our code over time in a special database called* ***repository*”**

We can *look at our project history and see who has made what changes, when and why*.

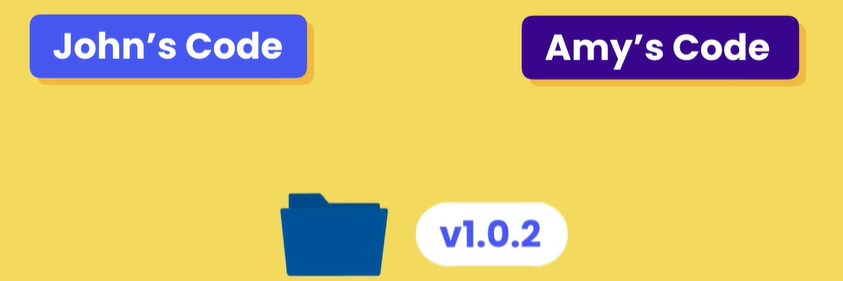
And if we screw something up we can *easily revert our project back to an earlier state*.



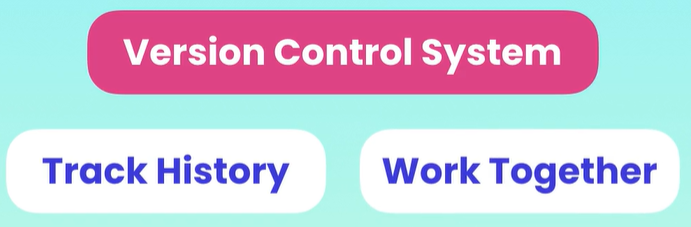
**Fig**. Code 🡪 Repo 🡪 Project History

Without a version control system, we would have to constantly store copies of entire project in various folders which is very slow and does not scale at all (*especially if multiple people are working on the same project*).

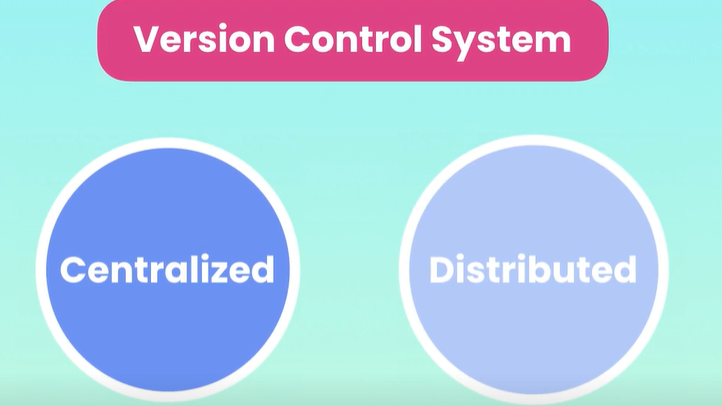
You would have to constantly toss around latest code by email or some other mechanism and then manually merge the changes.

 🡪 

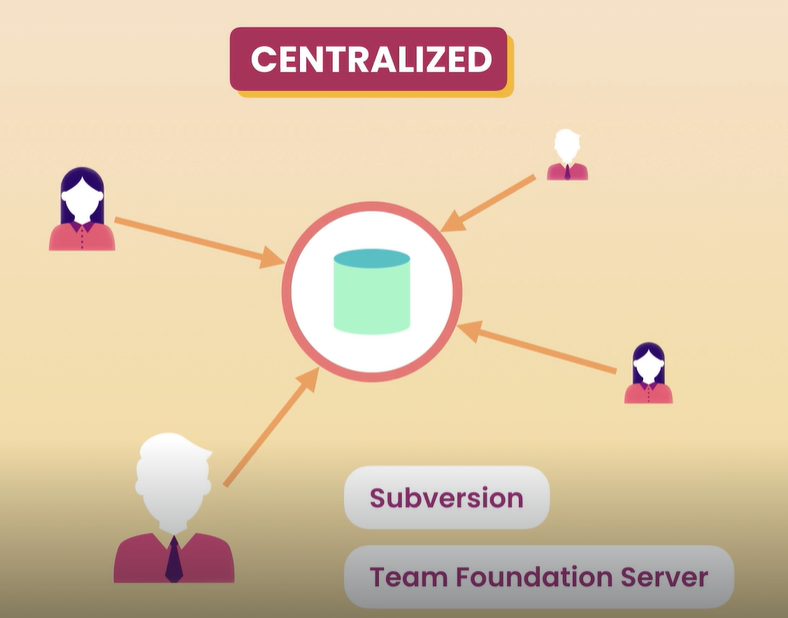
With the help of version control system, we can



Now version control system falls in to two categories,



In *Centralized* system, *all team members connect to a central server to get the latest copy of the code and to share the changes with others*.

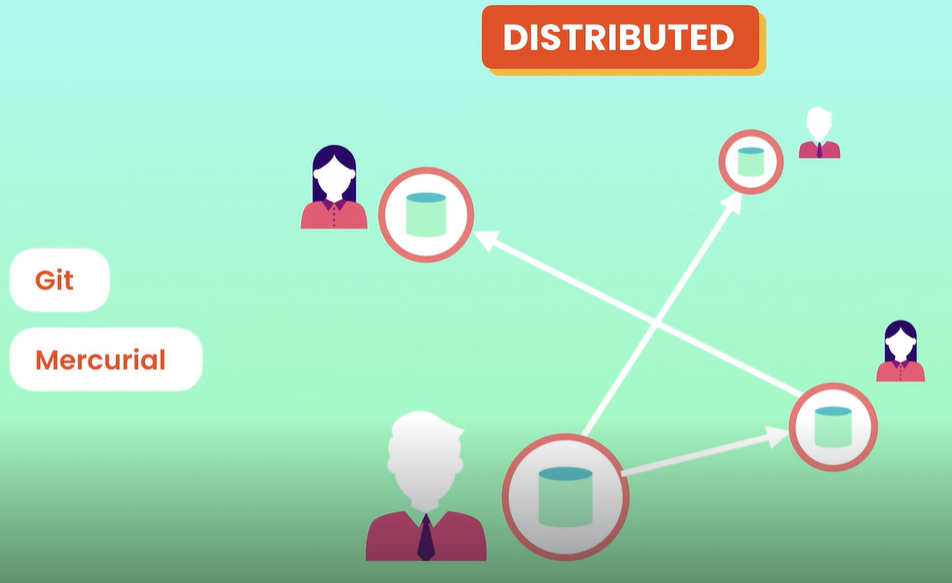


Subversion and Microsoft team Foundation server are examples of centralized version control systems.

The *problem* with centralized architecture is the *single point of failure*. *If the server goes offline, we cannot collaborate or save snapshots of our project*.

On the other hand in *distributed* systems, *every team member has a copy of the project with its history in the machine*, so we can *save snapshot of our project* ***locally*** *on our machine*.

Even if the central server gets offline, we can still synchronize our work directly with others. GIT and Mercurial are examples of distributed version control systems.



GIT is more famous and widely used because,



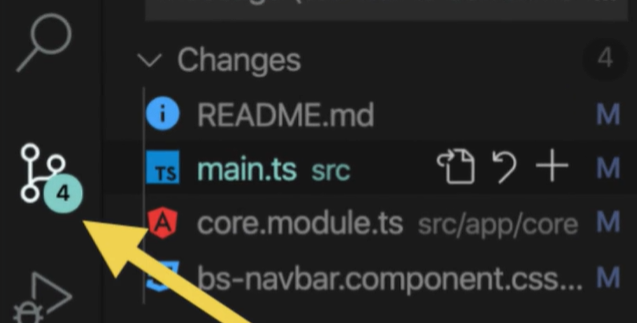
And Operations like *branching* and *merging* are slow and painful in other systems but are very fast in GIT.

**Using GIT**:

We can use GIT on,

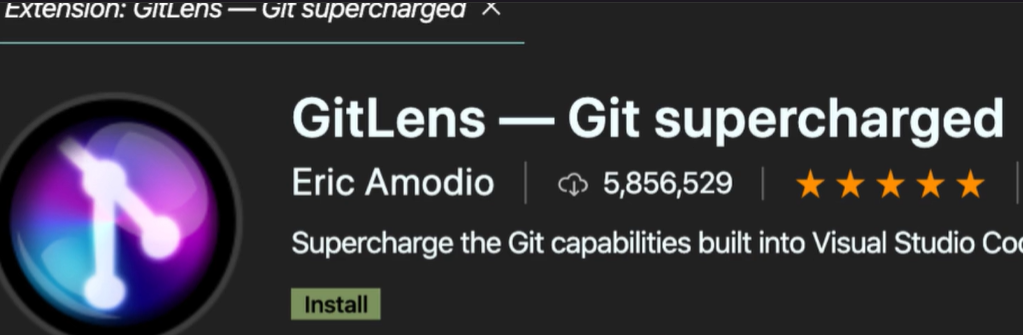
1. Command line (*Fastest and sometimes easiest way*).
2. Code Editors and IDEs

In VS code we have *source control panel* which gives us the essential GIT features.



There are some extensions available as well for bringing additional GIT features.

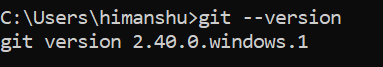
Most popular one is,



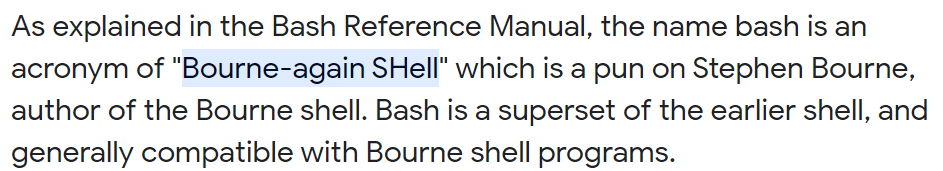
1. Graphical User interfaces. (Most popular ones are *GitKraken* and *Sourcetree*).

We will use command line for most in this course, because GUI tools have limitations (*not everything can be done from GUI tools only basic commands*) also they may not always be available (*permission issues*)

**Installing GIT**:



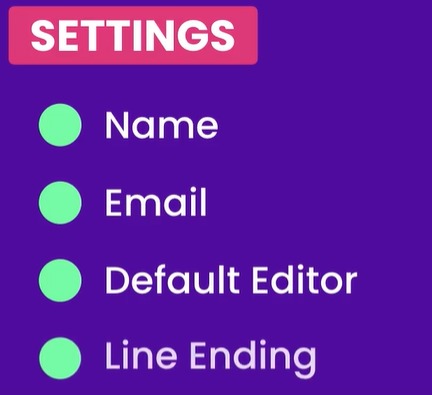
Once we install GIT on windows, we get **GIT BASH**.



It emulates Linux environment.

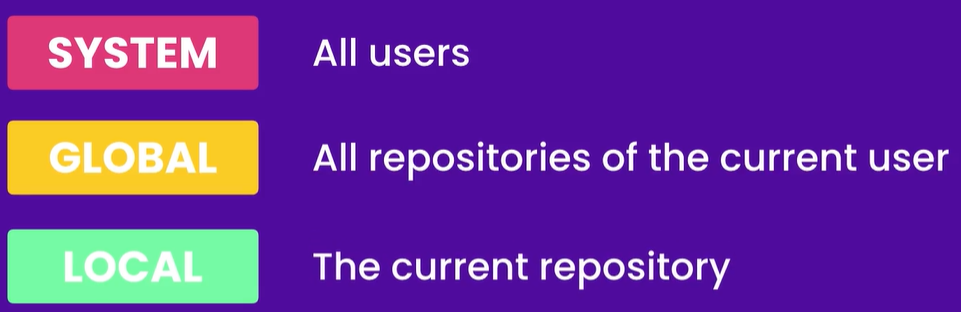
**Configuring GIT**:

First time we use GIT, we have to specify a few configuration settings,



We can configure these settings at 3 different levels,

**Levels**  **Applies on**



🡪 To set the *user name* *setting* at *global* level,

$ git config --global user.name "Himanshu Pandey"

Note: we use double quotes “ ” while specifying value here because of space in the name.

🡪 To set the *user email* *setting* at *global* level,

$ git config --global user.email himanshu.pandey@rsystems.com

🡪 To set the *default editor* *setting* at *global* level to **VS code**,

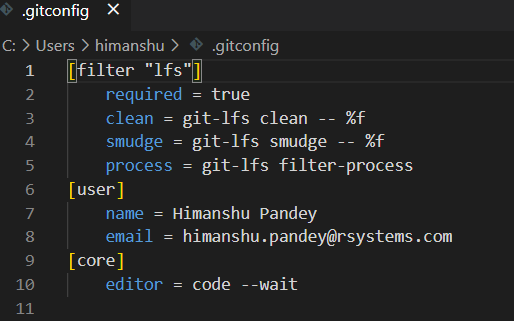
$ git config --global core.editor "code --wait"

With the --*wait* flag we can tell the terminal window to wait until we close the new VS code instance.

Note: All these configuration settings are stored in a txt file, which we can edit using our default editor (*VSC in our case*).

To open our config settings in our default editor,

$ git config --global -e



*How GIT should handle end of lines*,

This is a very important setting that a lot of developers miss,

🡪 In Windows end of lines is marked with two special keywords, **\r** (*carriage return*) and **\n** (*line feed*).

🡪 In Mac and Linux machines, end of line is indicated with **\n** (*line feed*)

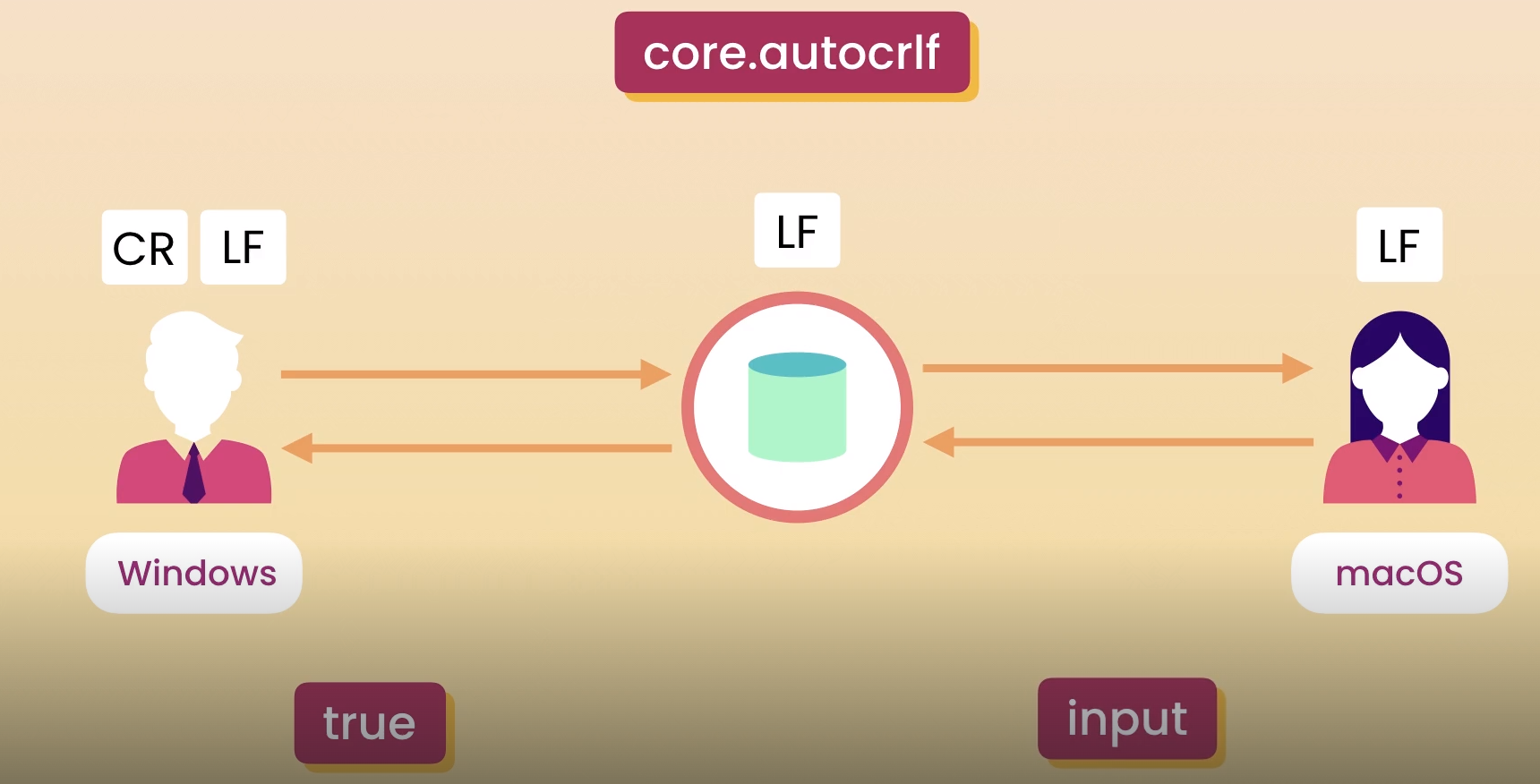


If we do not handle end of lines properly, we can run into some weird issues down the road.

To prevent this we have to configure a property called ***core.autocrlf***, which is short for auto carriage return line feed.

Suppose we have two people John and Julie working with the same repo, John has a windows machine & Julie uses mac.

So when john wants to check in his code into repo, GIT should remove the carriage return character from end of lines. Similarly when he checks out his code from repo, GIT should update the end of lines and add the carriage return character.



To achieve this behavior we should *set the value of this* ***core.autocrlf*** *property to* ***True*** *in case of* ***windows*** *machine*.

*For* ***mac****, set* ***core.autocrlf*** *property to* ***input***.

$ git config --global core.autocrlf true

**Getting Help**:

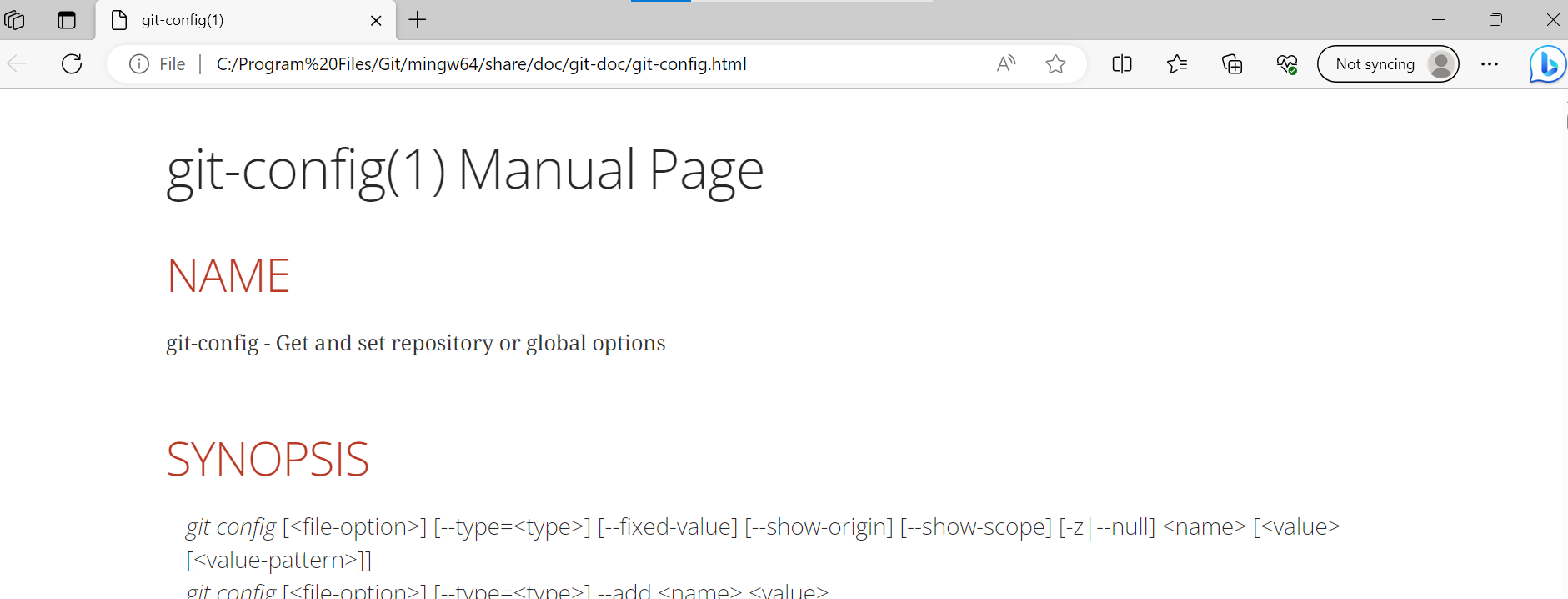
If you want to learn more about config command

<https://git-scm.com/docs/git-config>

We can access the same page by typing

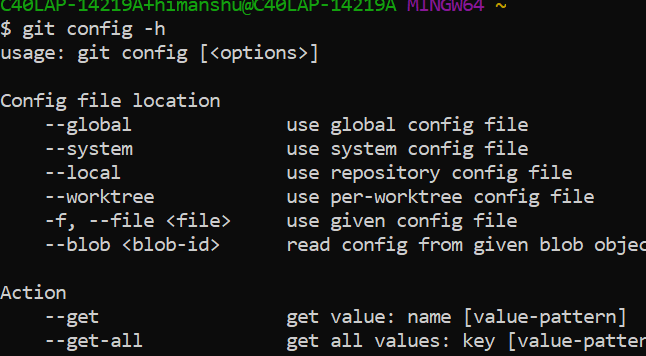
$ git config --help

In opens a HTML document on my machine.



If you do not want complete document just a refresher,

$ git config –h



It gives a short summary of these commands and how to use them.