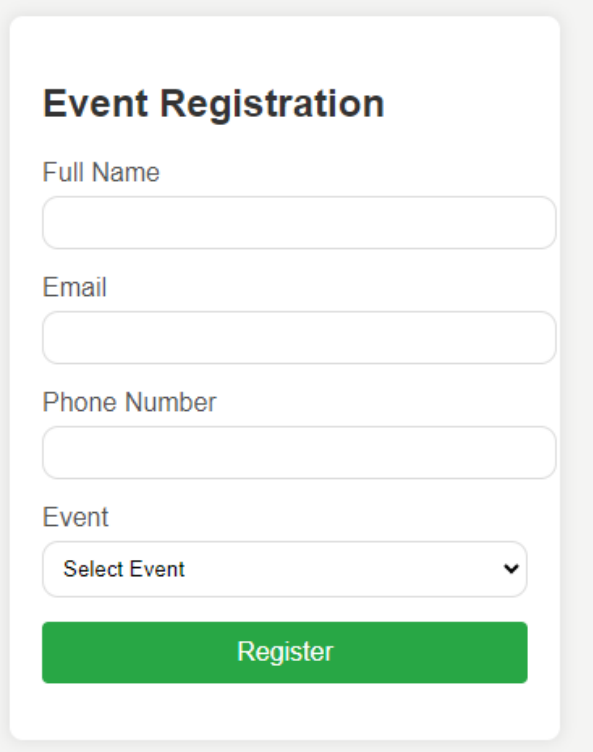
**1. Write code for a simple user registration form for an event.**

```
<!DOCTYPE html>
<html>
<head>
      <title>Event Registration Form</title>
   <style>
      body {
         font-family: Arial, sans-serif;
         background-color: #f4f4f3;
         display: flex;
         justify-content: center;
         align-items: center;
         height: 100vh;
         margin: 0;
      }
      .registration-form {
         background-color: #fff;
         padding: 20px;
         border-radius: 8px;
         box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);
         width: 300px;
      }
      .registration-form h2 {
         margin-bottom: 20px;
         font-size: 24px;
         color: #333;
      }
      .form-group {
         margin-bottom: 15px;
      }
      .form-group label {
         display: block;
         margin-bottom: 5px;
         color: #555;
      }
      .form-group input, .form-group select {
         width: 100%;
         padding: 8px;
         border: 1px solid #ddd;
         border-radius: 10px;
```

```
        }
        .form-group button {
            width: 100%;
            padding: 10px;
            background-color: #28a745;
            border: none;
            border-radius: 4px;
            color: #fff;
            font-size: 16px;
            cursor: pointer;
        }
            </style>
</head>
<body>
    <div class="registration-form">
        <h2>Event Registration</h2>
        <form action="/submit" method="post">
            <div class="form-group">
                <label input="name">Full Name</label>
                <input type="text" id="name" name="name" required>
            </div>
            <div class="form-group">
                <label input="email">Email</label>
                <input type="email" id="email" name="email" required>
            </div>
            <div class="form-group">
                <label input="phone">Phone Number</label>
                <input type="tel" id="phone" name="phone" required>
            </div>
            <div class="form-group">
                <label input="event">Event</label>
                <select id="event" name="event" required>
                    <option value="">Select Event</option>
                    <option value="event1">Paper Presentation</option>
                    <option value="event2">Poster Presentation</option>
                    <option value="event3">Project Expo</option>
                </select>
            </div>
            <div class="form-group">
                <button type="submit">Register</button>
```

```
        </div>
    </form>
  </div>
</body>
</html>
```

**Output:**



## 2. Explore Git and GitHub commands

**Version Control Systems / Source code management (SCM)/Software configuration management(SCM):**

**Version Control System:**

The process of monitoring and managing changes to software code is known as version control, also sometimes referred to as revision control or source control systems.

**Why Version Control System / Benefits of Version Control Systems:**

1) We can maintain different versions and we can choose any version based on client requirements.
2) With every version / commit we can maintain metadata like commit message
   who did changes
   when he did the change
   what changes he did
3) Developers can share the code to the peer developers in a very easy way.
4) Multiple developers can work in collaborative way
5) It enables Parallel development.
6) We can provide access control like who can read code or who can modify code
7) If something goes wrong, you can revert to previous stable versions of the code, ensuring that mistakes don't have lasting negative impacts

**Version Control System Terminology:**

**Working Directory:** Where developers are required to create/modify files. Here version control is not applicable. Here we won't use the word like version-1, version-2 etc

**Repository:** Where we have to store files and metadata. Here version control is applicable. Here we can talk about versions like version-1, version-2 etc

**Commit:** The process of sending files from the working directory to the repository.

**Checkout:** The process of sending files from repository to working directory.

**Types of Version Control Systems:**

**There are 2 types of Version Control Systems**

1) Centralized Version Control System
2) De Centralized / Distributed Version Control System

**Centralized Version Control System**

The name itself indicates that this type contains only one central repository and every developer should be connected to that repository. The total project code will be stored in the central repository. If 4 developers are there, still we have only one repository.
Example: CVS, SVN, Perforce, TFS, Clearcase etc

**Drawbacks:**

1) Central Repository is the only place where everything is stored, which causes a single point of failure. If something goes wrong to the central repository then recovery is very difficult.
2) All commit and checkout operations should be performed by connecting to the central repository via network. If network outage, then no version control to the developer. i.e in this type, developer work space and remote repository server should be connected always.
3) All commit and checkout operations should be performed by connecting to the central repository via network and hence these operations will become slow, which causes performance issues.
4) Organization of the central repository is very complex if the number of developers and files increases.

**Distributed Version Control Systems**

The name itself indicates that the repository is distributed and every developer's workspace contains a local copy of the repository. There is no question of a central repository.
1) The checkout and commit operations will be performed locally. Hence performance is more.
2) To perform checkout and commit operations, the network is not required. Hence if there is any network outage, still version control is applicable.
3) If something goes wrong in any repository there is a chance to recover. There is no question of a single point of failure.
4) To perform push and pull operations the network must be required, but these operations are not the most common operations and we are performing very rarely.

**Tools: Git, Mercurial, Fossil**

**Note:**

1) commit and checkout operations will be performed between workspace and repository.

work space —>commit—-->  Repository
Repository—>checkout  —->workspace

2) push and pull operations will be performed between repositories.

one repository ---->push  —other repository
one repository  <—pull----other repository

**Remote Repository Server is not Central Repository:**

1) Every developer has his own local copy of the repository. It is not centralized and it is distributed only.

2) commit and checkout operations won't be performed on remote repositories and these will be performed on local repositories only.

3) The main job of remote repositories is just to share our work to peer developers.

4) High availability, Speed and there is no single point of failure are the main reasons for the popularity of this model.

**Example Tools or server** :

Github, Gitlab, bitBucket, cloud platforms

**What is GIT:**

1. Git is a Distributed Version Control System Tool.
2. Git is not acronym and hence no expansion.But most of the people abbreviated as "Global Information Tracker".
3. GIT is developed by Linus(lai-nuhs taw vldz) Torvalds(Finnish software engineer), who also developed Linux Kernel.

**Features of GIT:**

**Distributed:**

1. Every Developer has his own local repository. All the operations can be performed locally.Hence local repo and remote repo need not be connected always.

2. All operations will be performed locally, and hence performance is high when compared with other VCSs. i.e it is very speed

3. Most of the operations are local. Hence we can work offline most of the time.

4. There is no single point failure as Every Developer has his own local repository.

5. It enables parallel development & automatic-backups.

**Staging Area:**

1 It is also known as the index area. There is a logical layer/virtual layer in git between working directory and local repository.

2. Working Directory Staging Area Local Repository

3. We cannot commit the files of the working directory directly. First we have to add to the staging area and then we have to commit.

4.This staging area is helpful to double check/cross-check our changes before commit.

5. This type of layer is not available in other Version Control System Tools like CVS, SVN etc

6. Git stores files in the repository in some hash form, which saves space. GIT will use an internal snapshot mechanism for this. All these conversions and taking snapshots of our data will happen in the staging area before commit.

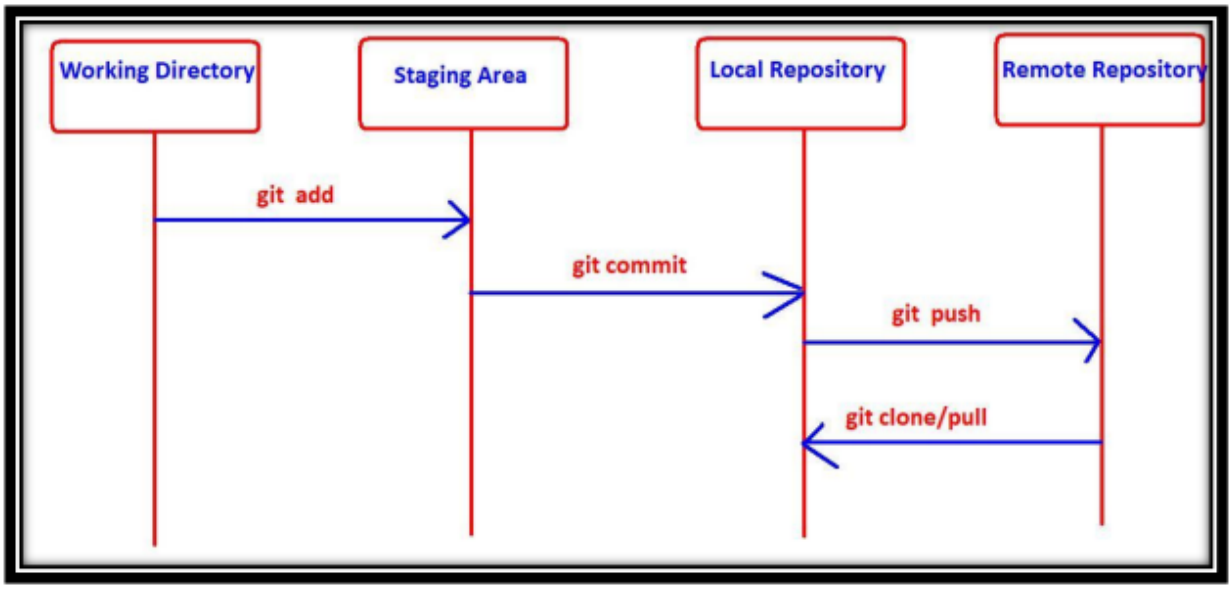
Eg: If a sample repository takes around 12 GB space in SVN where as in GIT it takes hardly 420 MB.

Hence Staging Area is the most important Strength of GIT.

**Branching and Merging:**

1. We can create and work on multiple branches simultaneously and all these branches are isolated from each other. It enables multiple workflows.
2. We can merge multiple branches into a single branch. We can commit branch wise also.
3. Moving files in GIT is very easy as GIT automatically tracks the moves. Whereas in other VCS we need to create a new file & then delete the old one.
4. Freeware and Open Source
5. It provides support for multiple platforms.

**GIT Architecture:**




GIt contains 2 types of repositories:

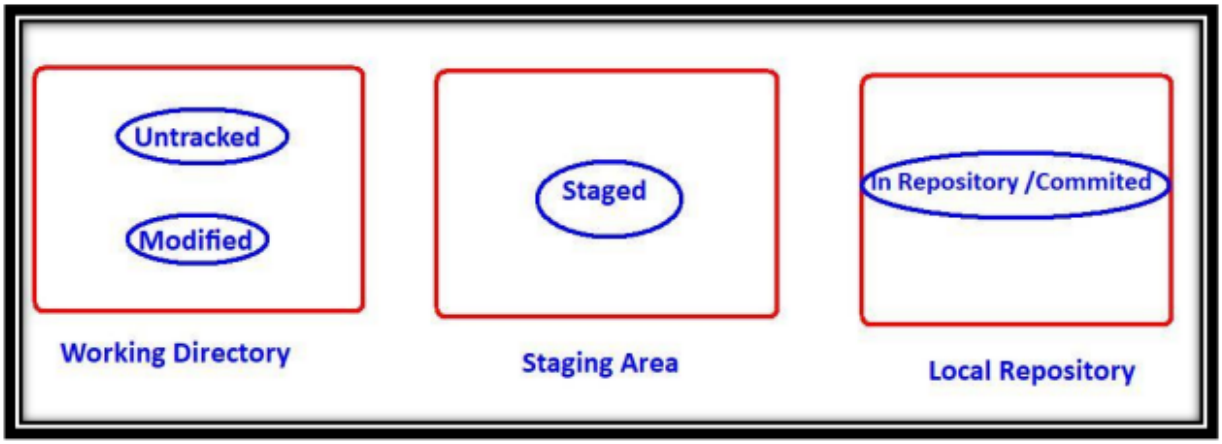
1) Local Repository

2) Remote Repository

- For every developer, a separate local repository is available. Developer can perform all checkout and commit operations wrt local repository only.
- To perform commit operation, first he has to add files to staging area by using git add command, and then he has to commit those changes to the local repository by using git commit command. Hence commit in GIT is a 2-step process.
- commit is applicable only for staging area files but not for working directory files.
- If the developer wants to share his work to the peer developers then he has to push his local repository to the remote repository by using **git push** command.
- Remote repository contains total project code, which can be accessible by all developers. New developer can get local repository by cloning remote repository.For this we have to use the **git clone** command.
- A developer can get updates from the remote repository to the local repository by using

**git pull** command.
**git add:** To add files from working directory to staging area.
**git commit:** To commit changes from staging area to local repository.
**git push:** To move files from local repository to remote repository.
**git clone:** To create a new local repository from the remote repository.
**git pull:** To get updated files from remote repository to local repository.

**Life cycle of File in GIT**
Every file in GIT is in one of the following states:
**1)Untracked:**
The files which are newly created in working directory and git does not aware of these files are said to be in an untracked state.
**2)Staged:**
The files which are added to the staging area are said to be in staged state.
These files are ready for commit.
**3)In Repository/ Committed:**
Any file which is committed is said to be In Repository/Committed State.
**4)Modified:**
Any file which is already tracked by git, but it is modified in working directory is said to be in a Modified State.




**Example-1 To Understand Working Directory, Staging Area and Local Repository**
1) Creating workspace
2) git initialization
3) Creating files with some content in the working directory
4) Adding these files to staging area
5) Git Configurations before first commit
6) Commit those changes to local repository

First create directories

```
C:\Users\kits>e:

E:\>mkdir kits

E:\>cd kits

E:\kits>mkdir cse

E:\kits>cd cse

E:\kits\cse>
```

- Now **cse directory** acts as a working directory. We have to request git, to provide version control for this directory. For this we have to use the **git init** command.

## git init

- Once we create a workspace, if we want version control, then we require a local repository. To create that local repository we have to use the git init command.
- .git is an empty repository, which is a hidden directory.

```
E:\kits\cse>git init
Initialized empty Git repository in E:/kits/cse/.git/
```

**Note:**

1) If our working directory contains any files, then these files won't be added to the local repository by default, we have to add explicitly.

2) If our working directory already contains local repository(.git), still if we call git init command, then there is no impact.

**Creating Files with some Content and adding to staging Area and then commit:**

## git status

It shows the current status of all files in each area, like which files are untracked, which are modified, which are staged etc**.**

We can get concise information by using -s option.

**Syntax:** git status -s

```
E:\kits\cse>git status
On branch master

No commits yet

nothing to commit (create/copy files and use "git add" to track)
```

Create a.txt and b.txt with some content

```
E:\kits\cse>cat > a.txt
First line in a.txt
```

*** After writing the content, press ctrl + D to get back to the command prompt.**

```
E:\kits\cse>cat > b.txt
First line in b.txt
```

```
E:\kits\cse>git status
On branch master

No commits yet

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

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

```
E:\kits\cse>ls
a.txt  b.txt

E:\kits\cse>git ls-files
```

## git add

To add files from the working directory to the staging area for tracking/commiting purposes, we have to use the git add command.

**To add all files present in current working directory**

git add **.**

**To add one or more specified files**

git add a.txt

git add a.txt b.txt

**Even we can use pattern also**

git add *.txt

git add *.java

```
E:\kits\cse>git add a.txt b.txt

E:\kits\cse>git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   a.txt
        new file:   b.txt
```

## git config

### Git Configurations before 1st Commit:

Before the first commit, we have to configure username and mail id, so that git can use this information in the commit records. We can perform these configurations with the following commands.

```
E:\kits\cse>git config --global user.email "deepak@gmail.com"

E:\kits\cse>git config --global user.name "Deepak"
```

global means these configurations are applicable for all repositories created by git. If we are not using global then it is applicable only for the current repository.
We can change user name and mail id with the same commands

## git commit

If we want to commit staged changes, then we have to use the **git commit** command. For every commit, a unique commit id will be generated. It is of 40-length hexadecimal string.
While using git commit command, commit message is mandatory.

```
E:\kits\cse>git commit -m "added two files a.txt and b.txt"
[master (root-commit) 0fdda3a] added two files a.txt and b.txt
 2 files changed, 2 insertions(+)
 create mode 100644 a.txt
 create mode 100644 b.txt
```

```
E:\kits\cse>git status
On branch master
nothing to commit, working tree clean
```

## git log

It shows the history of all commits.
It provides commit id, author name,maild , timestamp and commit message.

```
E:\kits\cse>git log
commit 0fdda3afeab58faa4a875f6582b1a32d496ed6ba (HEAD -> master)
Author: Deepak <deepak@gmail.com>
Date:   Sat Jul 27 17:29:39 2024 +0530

    added two files a.txt and b.txt
```

### git log file1.txt

Gives information of a Particular File.

### git log --oneline

If we want concise output then we should go for the --oneline option.

git log --author shows commits based on Author.
Example: git log –author=Deepak

**Suppose If we modify the File Content in working Directory:**

```
E:\kits\cse>cat >> a.txt
second line
```

```
E:\kits\cse>cat >> b.txt
second line
```

```
E:\kits\cse>git status
On branch master
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:   a.txt
        modified:   b.txt

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

**Adding these modified Files to the staging Area and then commit:**

```
E:\kits\cse>git add a.txt b.txt

E:\kits\cse>git commit -m "Both files modified"
[master 865addb] Both files modified
 2 files changed, 2 insertions(+), 2 deletions(-)
```

We can combine above two commands into a single command
git commit -a -m "Both files modified"

```
E:\kits\cse>git log
commit 865addb2a53bf8e9d7020e426ad305c6e13c213a (HEAD -> master)
Author: Deepak <deepak@gmail.com>
Date:   Sat Jul 27 17:59:45 2024 +0530

    Both files modified

commit 0fdda3afeab58faa4a875f6582b1a32d496ed6ba
Author: Deepak <deepak@gmail.com>
Date:   Sat Jul 27 17:29:39 2024 +0530

    added two files a.txt and b.txt
```

**git ls-files:**

This command will list out all files which are tracked by git.

```
E:\kits\cse>git ls-files
a.txt
b.txt
```

**ls:**

This command will list out all files present in workspace


## git diff command

It is a very common requirement to find differences between the content of a particular file or all files.

1) Between working directory and staging area
2) Between working directory and last commit
3) Between staged area and last commit
4) Between working directory and a particular commit
5) Between staged area and a particular commit
6) Between two specific commits

For this we are required to use the **git diff** command. diff means difference

**Example:**

Create file1.txt file2.txt and add some content.

Type **vim file.txt** in command type

After executing the above command,the Text editor will open.

To insert data, type **i** in the keyword .

Enter the data you want to.

Press **esc** and type **:wq(in editor itself)** to come out of the vim file

**file1.txt**

First line in file1.txt
Second line in file1.txt

**file2.txt**

First line in file2.txt
Second line in file2.txt

Save the above files , add to staging area and commit

Again open file1.txt, file2.txt and add 2 more lines .

**file1.txt**

First line in file1.txt
Second line in file1.txt
Third line in file1.txt
Fourth line in file1.txt

**file2.txt**
First line in file2.txt
Second line in file2.txt
Third line in file2.txt
Fourth line in file2.txt

Now commit both files i.e 2 files contain 4 lines of content.
Now add one line to the file1.txt

**file1.txt**
First line in file1.txt
Second line in file1.txt
Third line in file1.txt
Fourth line in file1.txt
Fifth line in file1.txt

We are adding file1.txt to staging area
git add file1.txt

Again we add a new line in file1.txt of working directory

**file1.txt**
First line in file1.txt
Second line in file1.txt
Third line in file1.txt
Fourth line in file1.txt
Fifth line in file1.txt
sixth line in file1.txt

1. **To see the difference in File Content between Working Directory and staging Area**

```
P c@DESKTOP-TEKQI08 MINGW64 /e/kits/cse (master)
$ git diff file1.txt
warning: in the working copy of 'file1.txt', LF will be replaced by CRLF the nex
t time Git touches it
diff --git a/file1.txt b/file1.txt
index 0e17c9d..78ed7b9 100644
--- a/file1.txt
+++ b/file1.txt
@@ -3,3 +3,4 @@ Second line in file1.txt
 Third line in file1.txt
 Fourth line in file1.txt
 Fifth line in file1.txt
+Sixth line in file1.txt
```

**diff --git a/file1.txt b/file1.txt**
a/file1.txt means source copy which means staging area
b/file1.txt means destination copy which means working directory copy

**index 0e17c9d..e3e329f 100644**
0e17c9d  hash of source file content
e3e329f  hash of destination file content
100644  git file mode
First 3 characters(100) represent the type of file. 100 means ASCII text file.
Next 3 characters represents the file permissions. 644  rw-r--r--

**--- a/file1.txt**
--- means missing lines in staged copy

**+++ b/file1.txt**
+++ means new lines added in working directory version

**@@ -3,3 +3,4 @@**
-3,3
- means source version from 3rd line onwards total 3 lines
+3,4
+ means destination version from 3rd line onwards total 4 lines

If any line prefixed with space means it is unchanged.
If any line prefixed with + means it is added in destination copy.
If any line prefixed with - means it is removed in destination copy.

@@ -3,3 +3,4 @@@
Second line in file1.txt
Third line in file1.txt
Fourth line in file1.txt
Fifth line in file1.txt
+sixth line in file1.txt
Clear indication that one line added in the working directory copy when compared with staged copy.
+sixth line in file1.txt

2. **To see the difference in File Content between Working Directory and Last Commit**
   The last commit can be referenced by HEAD.
   git diff HEAD file1.txt
   It shows the differences between working copy and last commit copy**.**

```
P c@DESKTOP-TEKQI08 MINGW64 /e/kits/cse (master)
$ git diff HEAD file1.txt
warning: in the working copy of 'file1.txt', LF will be replaced by CRLF the nex
t time Git touches it
diff --git a/file1.txt b/file1.txt
index cadd0e1..78ed7b9 100644
--- a/file1.txt
+++ b/file1.txt
@@ -2,3 +2,5 @@ First line in file1.txt
 Second line in file1.txt
 Third line in file1.txt
 Fourth line in file1.txt
+Fifth line in file1.txt
+Sixth line in file1.txt
```

3. **To see the difference in File Content between staged Copy and Last Commit**

We have to use --staged option or --cached option.
git diff --staged HEAD file1.txt
It shows the differences between staged copy and last commit copy.
Here HEAD is optional. Hence the following 2 commands will produce same output
git diff --staged HEAD file1.txt
git diff --staged file1.txt

```
P c@DESKTOP-TEKQI08 MINGW64 /e/kits/cse (master)
$ git diff --staged HEAD file1.txt
diff --git a/file1.txt b/file1.txt
index cadd0e1..0e17c9d 100644
--- a/file1.txt
+++ b/file1.txt
@@ -2,3 +2,4 @@ First line in file1.txt
 Second line in file1.txt
 Third line in file1.txt
 Fourth line in file1.txt
+Fifth line in file1.txt
```

4. **To see the difference in File Content between specific Commit and Working Directory Copy.**

git diff 7chracters_of_specified_commitid filename

$ git log --oneline

042b184 (HEAD -> master) 2 files added and each file contains 4 lines

9b64469 2 files added and each file contains 2 lines

```
P c@DESKTOP-TEKQI08 MINGW64 /e/kits/cse (master)
$ git diff 9b64469 file1.txt
warning: in the working copy of 'file1.txt', LF will be replaced by CRLF the nex
t time Git touches it
diff --git a/file1.txt b/file1.txt
index d4effe0..78ed7b9 100644
--- a/file1.txt
+++ b/file1.txt
@@ -1,2 +1,6 @@
 First line in file1.txt
 Second line in file1.txt
+Third line in file1.txt
+Fourth line in file1.txt
+Fifth line in file1.txt
+Sixth line in file1.txt
```

**Summary:**

**git diff**

Shows the differences in the content of working directory, staging area and local repository. we can use in the following ways

**1) git diff file1.txt** To compare working directory copy with staged copy

**2) git diff HEAD file1.txt** To compare working directory copy with last commit copy

3) **git diff --staged file1.txt**

**git diff --cached file1.txt**

**git diff --staged HEAD file1.txt git diff --cached HEAD file1.txt**

To compare staged copy with last commit copy

**4) git diff <commit id>file1.txt** To compare working directory copy with the specified commit copy.

**5) git diff --staged <commit id>file1.txt** To compare staged copy with the specified commit copy.

**6) git diff <source commit id> <destination commit id>file1.txt** To compare content in the file between two commits

**7) git diff HEAD HEAD~1 file1.txt** To compare content in the file between last commit and last but one commit.

**8) git diff <source commit id> <destination commit id> :** To compare content of all files between two commits.

**9) git diff master test** It shows all differences between master branch and test branch

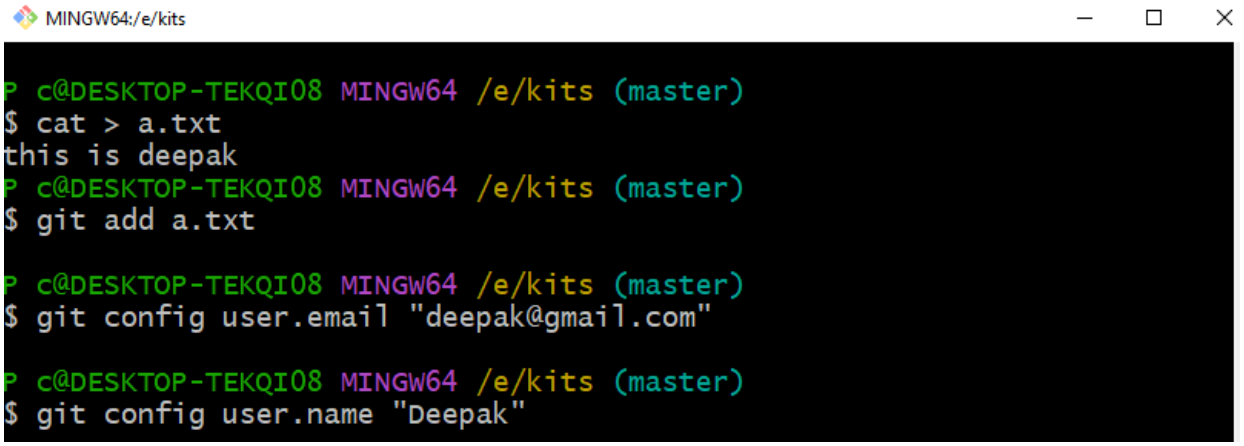
**10) git diff master origin/master** It shows all differences between master branch in local repository and master branch in remote repository.

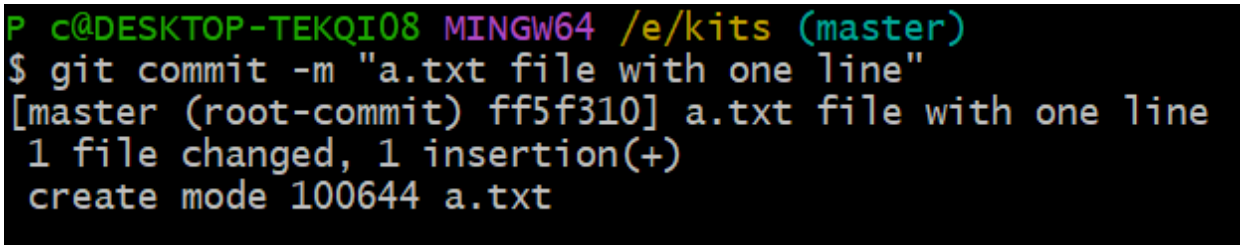
**git restore command:**

This command can be used to undo the effects of git add and unstage changes you have previously added to the Staging Area.

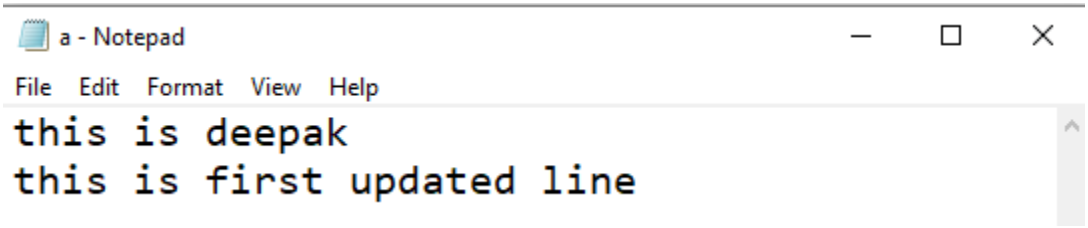
This restore command can also be used to discard local changes in a file, thereby restoring its last committed state.

**Example:**




**Open a.txt file and add one line**




**Open Git Bash , add the file to the staging area and see the status.**

```
P c@DESKTOP-TEKQI08 MINGW64 /e/kits (master)
$ git add a.txt
warning: in the working copy of 'a.txt', LF will be replaced by CRLF the
 next time Git touches it

P c@DESKTOP-TEKQI08 MINGW64 /e/kits (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   a.txt
```

**git restore –staged:**

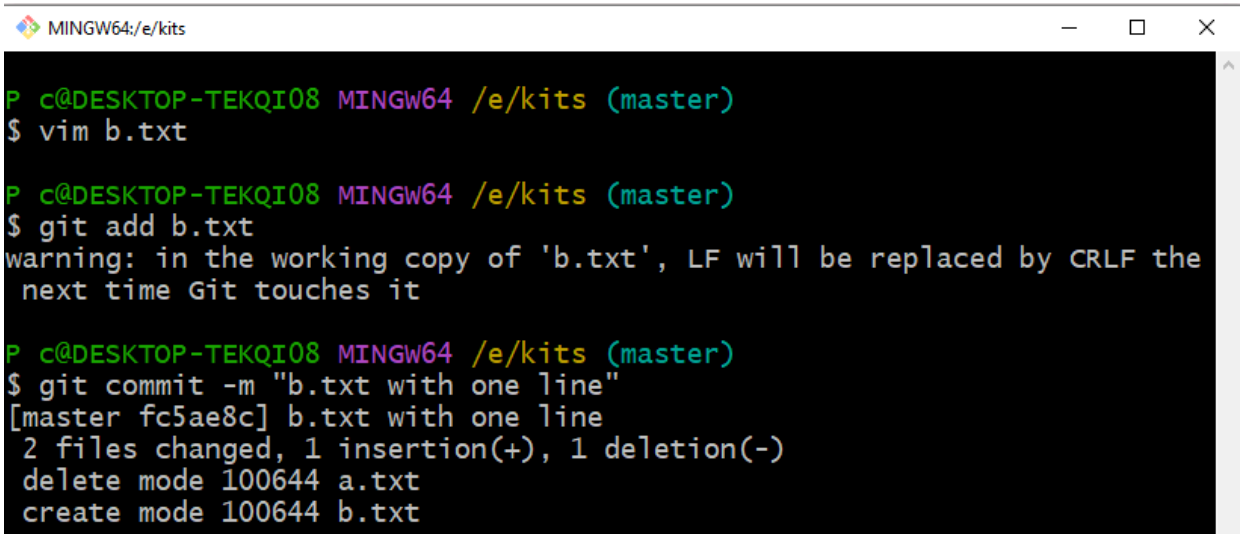
Removes the file from the Staging Area, but leaves its actual modifications untouched

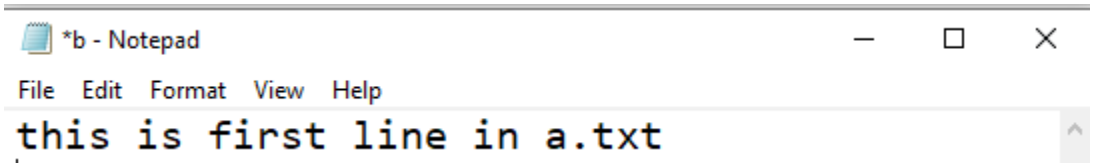
```
P c@DESKTOP-TEKQI08 MINGW64 /e/kits (master)
$ git restore --staged a.txt

P c@DESKTOP-TEKQI08 MINGW64 /e/kits (master)
$ git status
On branch master
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:   a.txt

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

**git restore:**

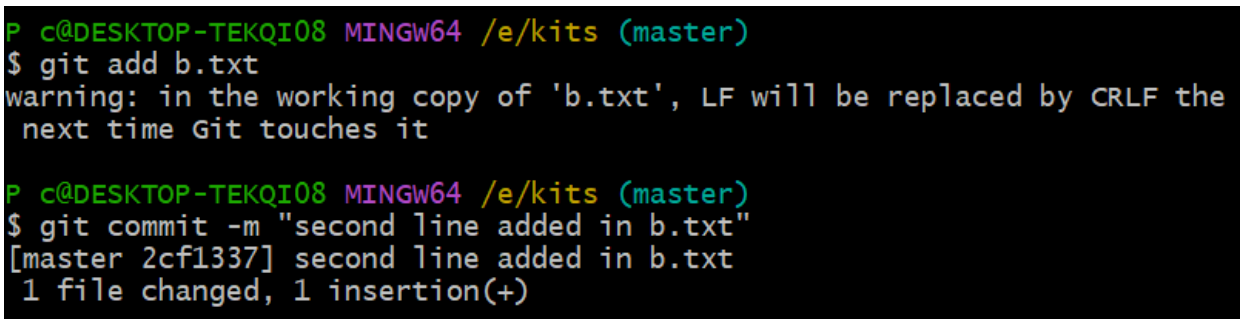
Suppose you made changes to a file named example.txt, but now you want to discard those changes.

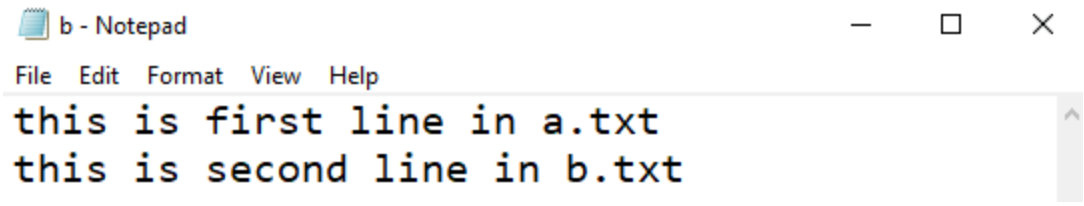
```
P c@DESKTOP-TEKQI08 MINGW64 /e/kits (master)
$ git restore a.txt
```


- *Open b.txt , add second line,add to the staging area and then commit*



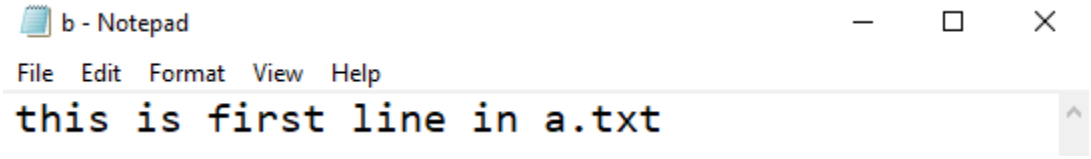



**git restore --source:**

This command is specifically designed to restore files in your working directory from a specific commit.

```
P c@DESKTOP-TEKQI08 MINGW64 /e/kits (master)
$ git log --oneline
2cf1337 (HEAD -> master) second line added in b.txt
fc5ae8c b.txt with one line
ff5f310 a.txt file with one line
```

```
P c@DESKTOP-TEKQI08 MINGW64 /e/kits (master)
$ git restore --source=fc5ae8c b.txt
```




**Difference between git restore and git checkout:**

**git restore --source:**

**Example:**

git restore --source=fc5ae8c a.txt

This command restores the example.txt file from commit fc5ae8c without affecting your current branch or the HEAD pointer.

Use git restore --source if we want to restore files from a specific commit or branch. It's part of the modern Git workflow, designed to make actions more intuitive.

**git checkout:**

**Example:**

**git checkout fc5ae8c -- a.txt**

This command does a similar thing by restoring example.txt from commit fc5ae8c, but the git checkout command can also be used to switch branches, which might cause confusion.

Use **git checkout** if you need to switch branches or restore files with an older Git version, but be cautious of its dual functionality.

## Removing Files by using git rm Command

It is a very common requirement to remove files from the working directory and staging area. For these removals we can use the following commands

1) To Remove Files from Working Directory and staging Area (git rm)

   **git rm file1.txt**

   file1.txt will be removed from staging area and from working directory
2) **git rm -r .** It will remove all files from the staging area and working directory.
3) To Remove Files Only from staging Area (git rm --cached)

   **git rm --cached file4.txt**

   file4.txt will be removed only from staging area but not from working directory
4) To Remove Files Only from Working Directory (rm Command)

   **rm file1.tx**t

**Undo Changes with *git Checkout* Command**

We can use the checkout command to discard unstaged changes in the tracked files of the working directory.

**Observe the 3 words:**

1) Only for working directory
2) To discard unstaged changes(The changes which are not added to staging area)
3) In the tracked files (The files which are already added to staging area/commit)

It is something like an undo operation. It will copy the contents of the file from the index area(staging area) to the working directory.

**Syntax:**

git checkout -- filename

**Example:**

Create a text file with name sample.txt and add some content.

**sample1.txt**

First line
Second line

Save the file , add to staging area (ignore warnings) and do commit

```
P c@DESKTOP-TEKQI08 MINGW64 /e/kits/cse (master)
$ git add sample1.txt
warning: in the working copy of 'sample1.txt', LF will be replaced by CRLF the n
ext time Git touches it

P c@DESKTOP-TEKQI08 MINGW64 /e/kits/cse (master)
$ git commit -m "sample1.txt contains 2 lines"
[master 3cb7476] sample1.txt contains 2 lines
 1 file changed, 2 insertions(+)
 create mode 100644 sample1.txt
```

Again open sample1.txt and add a third line.(Don't add to staging area)

**sample1.txt**
First line
Second line
Third line

Now sample1.txt in the working directory contains 3 lines and in staging area sample1.txt contains 2 lines.

```
P c@DESKTOP-TEKQI08 MINGW64 /e/kits/cse (master)
$ cat sample1.txt
First line
second line
third line

P c@DESKTOP-TEKQI08 MINGW64 /e/kits/cse (master)
$ git checkout -- sample1.txt

P c@DESKTOP-TEKQI08 MINGW64 /e/kits/cse (master)
$ cat sample1.txt
First line
second line
```

**Note:** git checkout is applicable only for the files which are already tracked by git. It is not applicable for new files.
git checkout If we are not passing any argument, then this command will show the list of eligible files for checkout
git checkout command can be used in branching also.

**Branching And Merging**

**Branching:**

Till now whatever files created and whatever commits we did, all these happened in the master branch.

master branch is the default branch/ main branch in git.

Generally the main source code will be placed in the master branch.

**Need of creating a New Branch:**

Assume we want to work on new requirements independently, then instead of working in the master branch, we can create a separate branch and we can work in that branch, related to that new requirement without affecting the main branch.

A Branch is an independent flow of development and by using branching concepts, we can create multiple workflows.

Based on our requirement, we can create any number of branches.

**Note:**

1. Once we create a branch, all files and commits will be inherited from the parent branch to the child branch. Branching is a logical way of duplicating files and commits. In the child branch we can create new files and we can perform new commits based on our requirements.
2. All branches are isolated from each other. The changes performed in the master branch are not visible to the new branch and the changes performed in the new branch are not visible to the master branch.
3. Once the work is completed in a new branch then we can merge that new branch to the main branch or we can push that branch directly to the remote repository.

**Commands in Branching:**

**To View Branches:**

To know all available branches in our local repository, we have to use the **git branch** command.

git branch

- ❖ It will show all branches in our local repository.
- ❖ By default we have only one branch: master

- ❖ master is the default name provided by GIT

**Example:**

```
P c@DESKTOP-TEKQI08 MINGW64 /e/kits/cse (master)
$ git branch
* master
```

* indicates that master is currently an active branch.

**Note:**

There is another way to check on which branch currently we are working, for this we have to use the **git status** command.

```
P c@DESKTOP-TEKQI08 MINGW64 /e/kits/cse (master)
$ git status
On branch master
nothing to commit, working tree clean
```

**How to Create a New Branch:**

We can create a new branch by using the **git branch** command.
**Syntax:** git branch branch_name
It will create a new branch: branch_name

**Example:**

```
P c@DESKTOP-TEKQI08 MINGW64 /e/kits/cse (master)
$ git branch child

P c@DESKTOP-TEKQI08 MINGW64 /e/kits/cse (master)
$ git branch
  child
* master
```

**How to Switch from one Branch to another Branch?**

We have to use the **git checkout** command.
**Syntax:** git checkout branch_name

```
P c@DESKTOP-TEKQI08 MINGW64 /e/kits/cse (master)
$ git checkout child
Switched to branch 'child'

P c@DESKTOP-TEKQI08 MINGW64 /e/kits/cse (child)
$ git branch
* child
  master
```

**Short-cut Way to Create a New Branch and switch to that Branch:**
We have to use the -b option with the checkout command.
**Syntax:**
git checkout -b new2branch

```
P c@DESKTOP-TEKQI08 MINGW64 /e/kits/cse (master)
$ git checkout -b child1
Switched to a new branch 'child1'

P c@DESKTOP-TEKQI08 MINGW64 /e/kits/cse (child1)
$ git branch
  child
* child1
  master
```

**Example:**

```
P c@DESKTOP-TEKQI08 MINGW64 /e/kits/cse (master)
$ touch a.txt b.txt c.txt

P c@DESKTOP-TEKQI08 MINGW64 /e/kits/cse (master)
$ git config user.email "deepak@gmail.com"

P c@DESKTOP-TEKQI08 MINGW64 /e/kits/cse (master)
$ git config user.name "Deepak"

P c@DESKTOP-TEKQI08 MINGW64 /e/kits/cse (master)
$ git add a.txt

P c@DESKTOP-TEKQI08 MINGW64 /e/kits/cse (master)
$ git commit -m "a.txt added"
[master (root-commit) 9b0de2b] a.txt added
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 a.txt
```

```
P c@DESKTOP-TEKQI08 MINGW64 /e/kits/cse (master)
$ git add b.txt

P c@DESKTOP-TEKQI08 MINGW64 /e/kits/cse (master)
$ git commit -m "b.txt added"
[master 5211a21] b.txt added
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 b.txt

P c@DESKTOP-TEKQI08 MINGW64 /e/kits/cse (master)
$ git add c.txt

P c@DESKTOP-TEKQI08 MINGW64 /e/kits/cse (master)
$ git commit -m "c.txt added"
[master d1b5c62] c.txt added
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 c.txt
```

```
P c@DESKTOP-TEKQI08 MINGW64 /e/kits/cse (master)
$ git log --oneline
d1b5c62 (HEAD -> master) c.txt added
5211a21 b.txt added
9b0de2b a.txt added

P c@DESKTOP-TEKQI08 MINGW64 /e/kits/cse (master)
$ ls
a.txt  b.txt  c.txt

P c@DESKTOP-TEKQI08 MINGW64 /e/kits/cse (master)
$ git branch child

P c@DESKTOP-TEKQI08 MINGW64 /e/kits/cse (master)
$ git branch
  child
* master
```

```
P c@DESKTOP-TEKQI08 MINGW64 /e/kits/cse (master)
$ git checkout child
Switched to branch 'child'

P c@DESKTOP-TEKQI08 MINGW64 /e/kits/cse (child)
$ git branch
* child
  master

P c@DESKTOP-TEKQI08 MINGW64 /e/kits/cse (child)
$ ls
a.txt  b.txt  c.txt

P c@DESKTOP-TEKQI08 MINGW64 /e/kits/cse (child)
$ git log --oneline
d1b5c62 (HEAD -> child, master) c.txt added
5211a21 b.txt added
9b0de2b a.txt added
```

```
P c@DESKTOP-TEKQI08 MINGW64 /e/kits/cse (child)
$ git checkout master
Switched to branch 'master'

P c@DESKTOP-TEKQI08 MINGW64 /e/kits/cse (master)
$ touch d.txt

P c@DESKTOP-TEKQI08 MINGW64 /e/kits/cse (master)
$ git add d.txt

P c@DESKTOP-TEKQI08 MINGW64 /e/kits/cse (master)
$ git commit -m "d.txt added in master branch"
[master 34a15df] d.txt added in master branch
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 d.txt

P c@DESKTOP-TEKQI08 MINGW64 /e/kits/cse (master)
$ ls
a.txt  b.txt  c.txt  d.txt
```

```
P c@DESKTOP-TEKQI08 MINGW64 /e/kits/cse (master)
$ git checkout child
Switched to branch 'child'

P c@DESKTOP-TEKQI08 MINGW64 /e/kits/cse (child)
$ ls
a.txt  b.txt  c.txt
```

**Important Conclusions:**

1. All branches are isolated from each other. The changes performed in master branch are not visible to the new branch and the changes performed in the new branch are not visible to the master branch
2. In GIT branching, logical duplication of files will be happened. For every branch, new directory won't be created.

**Advantages:**

1. We can enable Parallel development.
2. We can work on multiple flows in an isolated way.
3. We can organize source code in a clean way.
4. Implementing new features will become easy
5. Bug fixing will become easy.
6. Testing new ideas or new technologies will become easy

**Merging of a Branch**

We created a branch to implement some new features and we did some new changes in that branch. Once work is completed we have to merge that branch back to the parent branch.
We can perform merge operations by using the **git merge** command.
We have to execute this command from the parent branch.

**Example:**
**Open Git Bash and execute the following commands**

```
P c@DESKTOP-TEKQI08 MINGW64 /
$ cd e:

P c@DESKTOP-TEKQI08 MINGW64 /e
$ mkdir kits

P c@DESKTOP-TEKQI08 MINGW64 /e
$ cd kits

P c@DESKTOP-TEKQI08 MINGW64 /e/kits
$ mkdir project1

P c@DESKTOP-TEKQI08 MINGW64 /e/kits
$ cd project1

P c@DESKTOP-TEKQI08 MINGW64 /e/kits/project1
$ git init
Initialized empty Git repository in E:/kits/project1/.git/
```

```
P c@DESKTOP-TEKQI08 MINGW64 /e/kits/project1 (master)
$ git config user.email "deepak@gmail.com"

P c@DESKTOP-TEKQI08 MINGW64 /e/kits/project1 (master)
$ git config user.name "Deepak"

P c@DESKTOP-TEKQI08 MINGW64 /e/kits/project1 (master)
$ touch a.txt b.txt
```

```
P c@DESKTOP-TEKQI08 MINGW64 /e/kits/project1 (master)
$ git add a.txt b.txt

P c@DESKTOP-TEKQI08 MINGW64 /e/kits/project1 (master)
$ git commit -m "a.txt and b.txt added to master branch"
[master (root-commit) 8b0e467] a.txt and b.txt added to master branch
 2 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 a.txt
 create mode 100644 b.txt
```

```
P c@DESKTOP-TEKQI08 MINGW64 /e/kits/project1 (master)
$ git branch child1

P c@DESKTOP-TEKQI08 MINGW64 /e/kits/project1 (master)
$ git checkout child1
Switched to branch 'child1'

P c@DESKTOP-TEKQI08 MINGW64 /e/kits/project1 (child1)
$ git branch
* child1
  master

P c@DESKTOP-TEKQI08 MINGW64 /e/kits/project1 (child1)
$ touch c.txt

P c@DESKTOP-TEKQI08 MINGW64 /e/kits/project1 (child1)
$ git add c.txt;git commit -m "c.txt added to child1 branch"
[child1 51ac09f] c.txt added to child1 branch
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 c.txt
```

**What is Fast-forward Merge?**

After creating a child branch, if we are not doing any new commits in the parent branch, then git will perform a fast-forward merge. i.e updates(new commits) happened only in the child branch but not in the parent branch.

In the fast-forward merge, git simply moves the parent branch and points to the last commit of the child branch.

```
P c@DESKTOP-TEKQI08 MINGW64 /e/kits/project1 (child1)
$ ls
a.txt  b.txt  c.txt

P c@DESKTOP-TEKQI08 MINGW64 /e/kits/project1 (child1)
$ git checkout master
Switched to branch 'master'

P c@DESKTOP-TEKQI08 MINGW64 /e/kits/project1 (master)
$ git merge child1
Updating 8b0e467..51ac09f
Fast-forward
 c.txt | 0
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 c.txt

P c@DESKTOP-TEKQI08 MINGW64 /e/kits/project1 (master)
$ ls
a.txt  b.txt  c.txt
```

**Working with Remote Repositories**

**Need of Remote Repositories:**

Upto this we created multiple files and we did several changes. We used multiple git commands. All these operations happened in the local repository.

If we are required to share our code to the peer developers/team members then the remote repository concept will come into the picture.

As GIT is a distributed version control system, every developer has his own local repository. Every developer can share his code to the peer developers/team members.

By using push operation, the developer can share his code with the peer developer. By using pull operation, developers can get the code of peer developers.

**The Problems if a Developer communicates directly with other Developers:**

1. On every developer's machine we have to install Git Server.
2. Every developer should be aware of the hostname and port number of all remaining developers, then only he can share the code.
3. If there is any change in port number of any developer's machine, intimating this for all developers every time is a very difficult task.

We can solve these problems with a common remote repository.

**The Advantages of Common Remote Repositories:**

1. On the developer's machine, we are not required to install a GIT server. GIT Server is available on a common remote repository.
2. Developer should only be aware of the url of the common remote repository and he is not required to be aware of all hostnames and port numbers of all remaining developer's machines.

**Note:**

1. Using a common remote repository is best practice in real time.
2. The most common service providers of remote repositories Github,Gitlab,Bitbucket etc
3. Some big companies may use their own common remote repository instead of using 3rd party repositories.
4. Sometimes in Real time we may be required to work with multiple remote repositories also.

## GitHub

What is GitHub?

GitHub is a Git repository hosting service. GitHub also facilitates with many of its features, such as access control and collaboration. It provides a Web-based graphical interface.
GitHub is an American company. It hosts source code of your project in the form of different programming languages and keeps track of the various changes made by programmers.
It offers both distributed version control and source code management (SCM) functionality of Git. It also facilitates with some collaboration features such as bug tracking, feature requests, task management for every project.

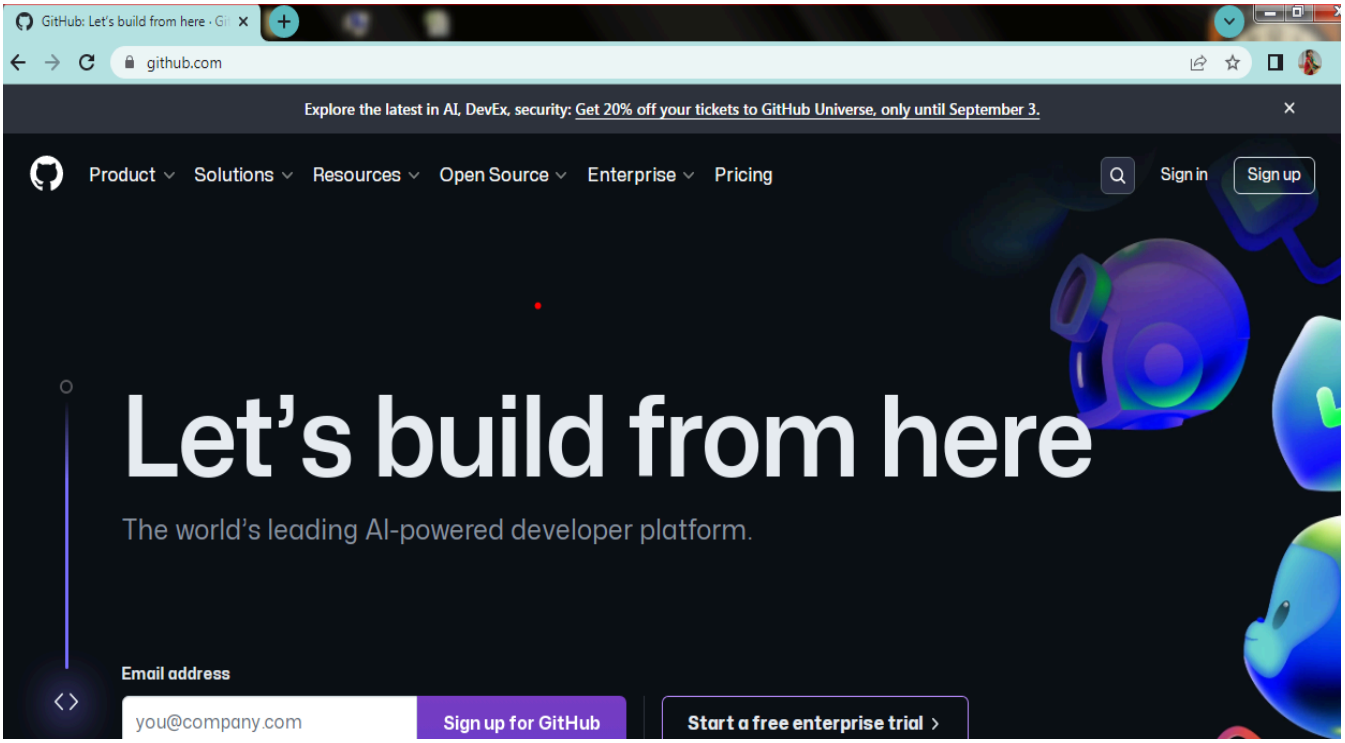
Benefits of GitHub

GitHub can be separated as the Git and the Hub. GitHub service includes access controls as well as collaboration features like task management, repository hosting, and team management.
The key benefits of GitHub are as follows.

- It is easy to contribute to open source projects via GitHub.
- It helps to create an excellent document.
- You can attract recruiter by showing off your work. If you have a profile on GitHub, you will have a higher chance of being recruited.
- It allows your work to get out there in front of the public.
- You can track changes in your code across versions.

**How to Create an Account in GitHub?.**

**Open the** https://github.com/ website




Select 'Sign Up' to create a new account. If you already have an account, click on 'Sign In'.

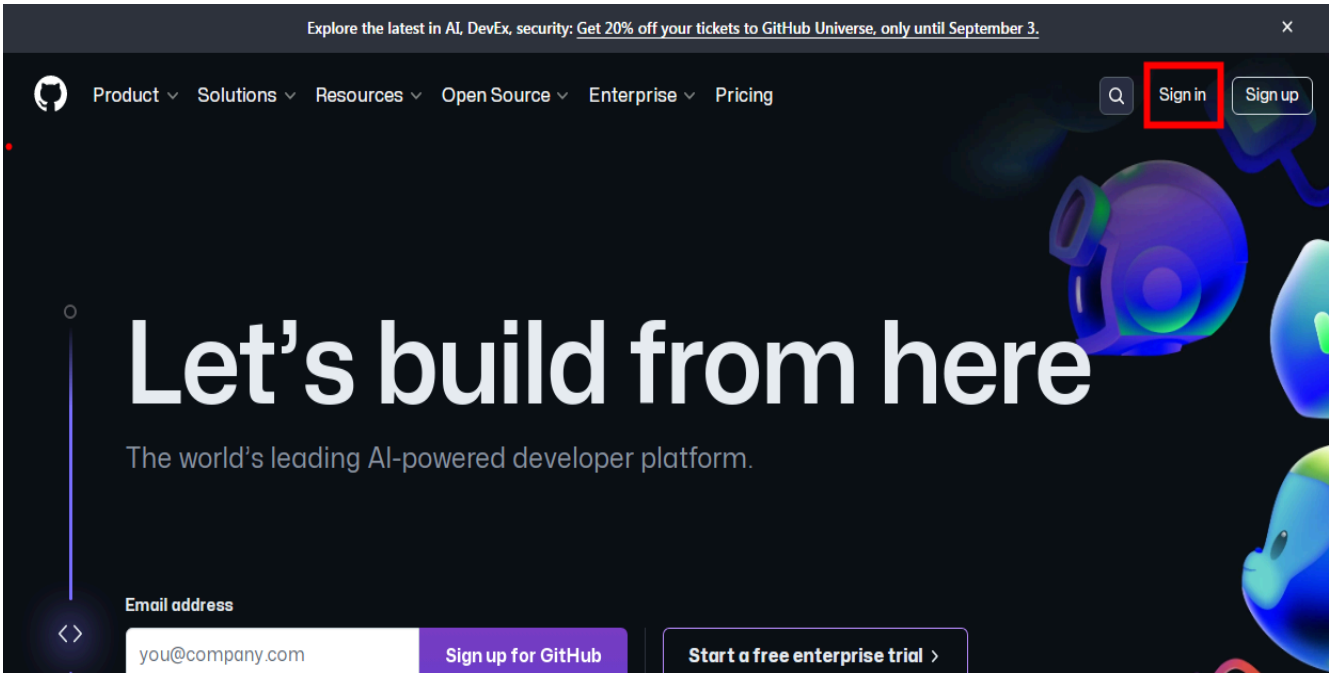
We should use a valid email ID because we need to verify it to create an account successfully. GitHub requires email verification.

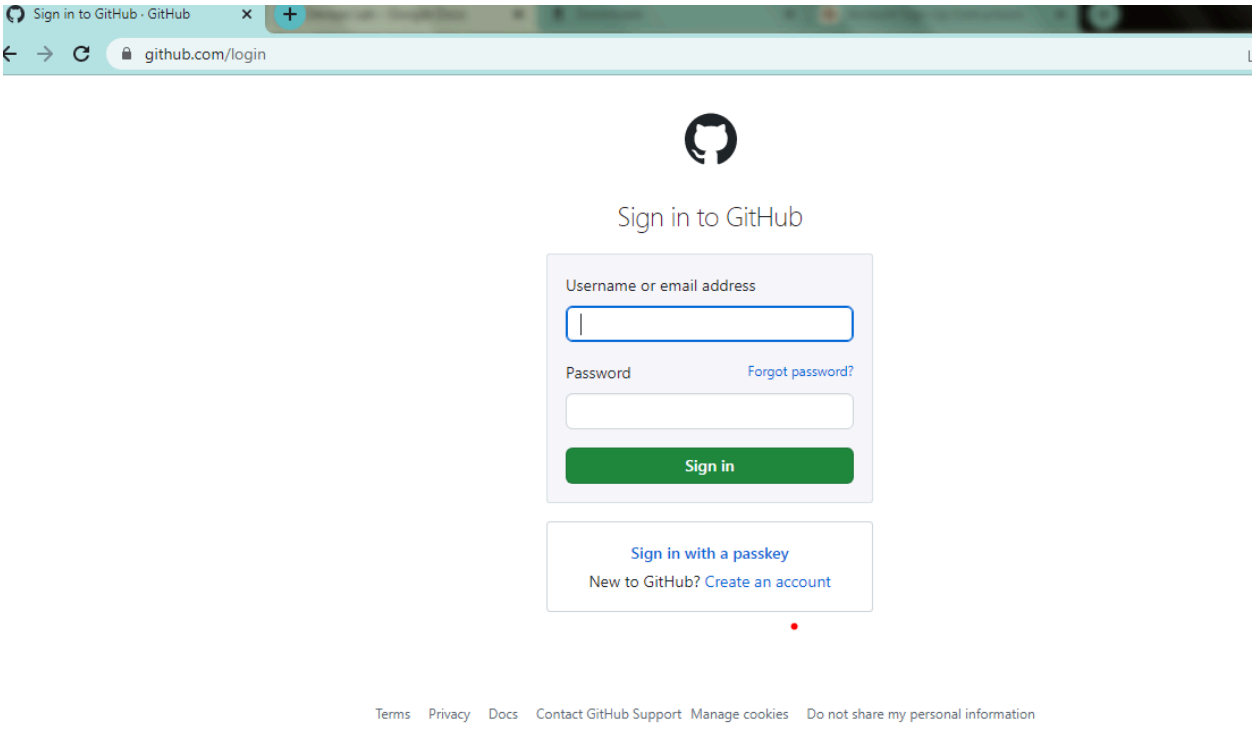
**Example:**

```
kits@kits-PC MINGW32 ~
$ cd e:

kits@kits-PC MINGW32 /e
$ mkdir projectkits

kits@kits-PC MINGW32 /e
$ cd projectkits

kits@kits-PC MINGW32 /e/projectkits
$ git init
Initialized empty Git repository in E:/projectkits/.git/

kits@kits-PC MINGW32 /e/projectkits (master)
$ cat > a.txt
First Line in a.txt
kits@kits-PC MINGW32 /e/projectkits (master)
$ git add a.txt

kits@kits-PC MINGW32 /e/projectkits (master)
$ git config user.email "gujjula@gmail.com"

kits@kits-PC MINGW32 /e/projectkits (master)
$ git config user.name "Deepak"
```

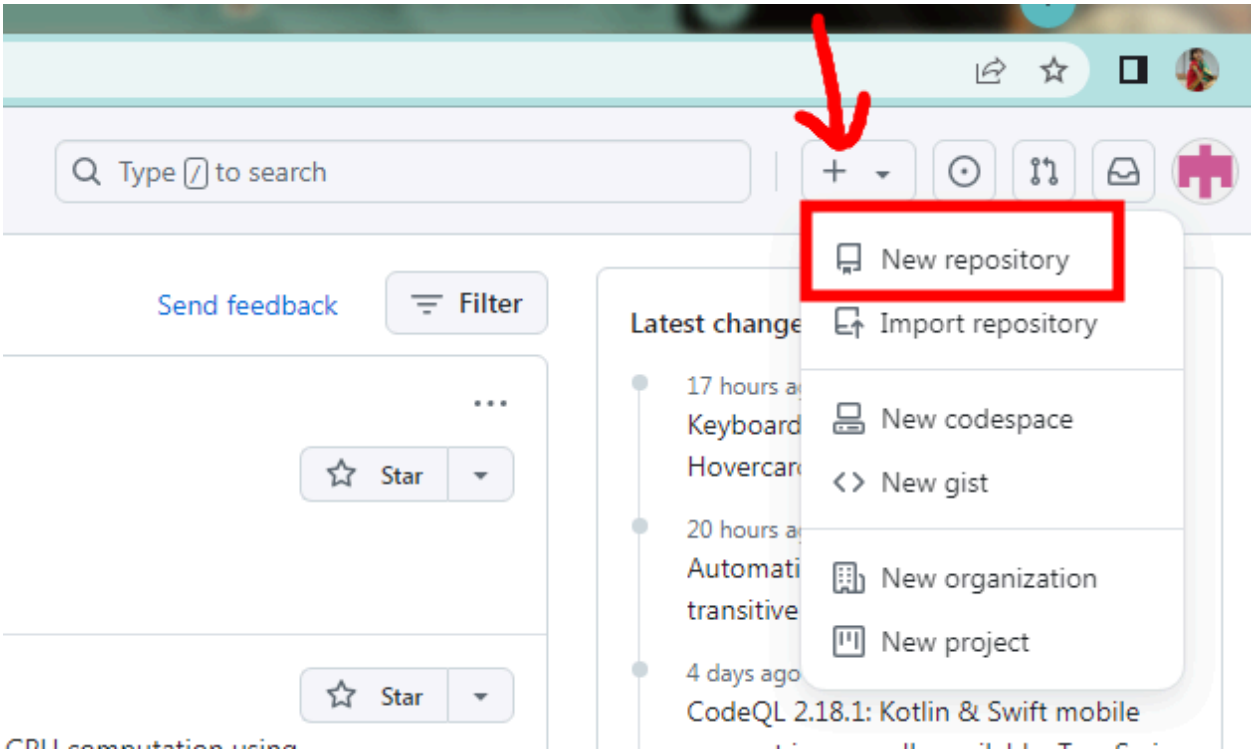
```
kits@kits-PC MINGW32 /e/projectkits (master)
$ git commit -m "a.txt added"
[master (root-commit) fa5fa1a] a.txt added
 1 file changed, 1 insertion(+)
 create mode 100644 a.txt

kits@kits-PC MINGW32 /e/projectkits (master)
$ cat > b.txt
Second line
kits@kits-PC MINGW32 /e/projectkits (master)
$ git add b.txt

kits@kits-PC MINGW32 /e/projectkits (master)
$ git commit -m "b.txt added"
[master 9c51b4a] b.txt added
 1 file changed, 1 insertion(+)
 create mode 100644 b.txt

kits@kits-PC MINGW32 /e/projectkits (master)
$ git log --oneline
9c51b4a (HEAD -> master) b.txt added
fa5fa1a a.txt added
```

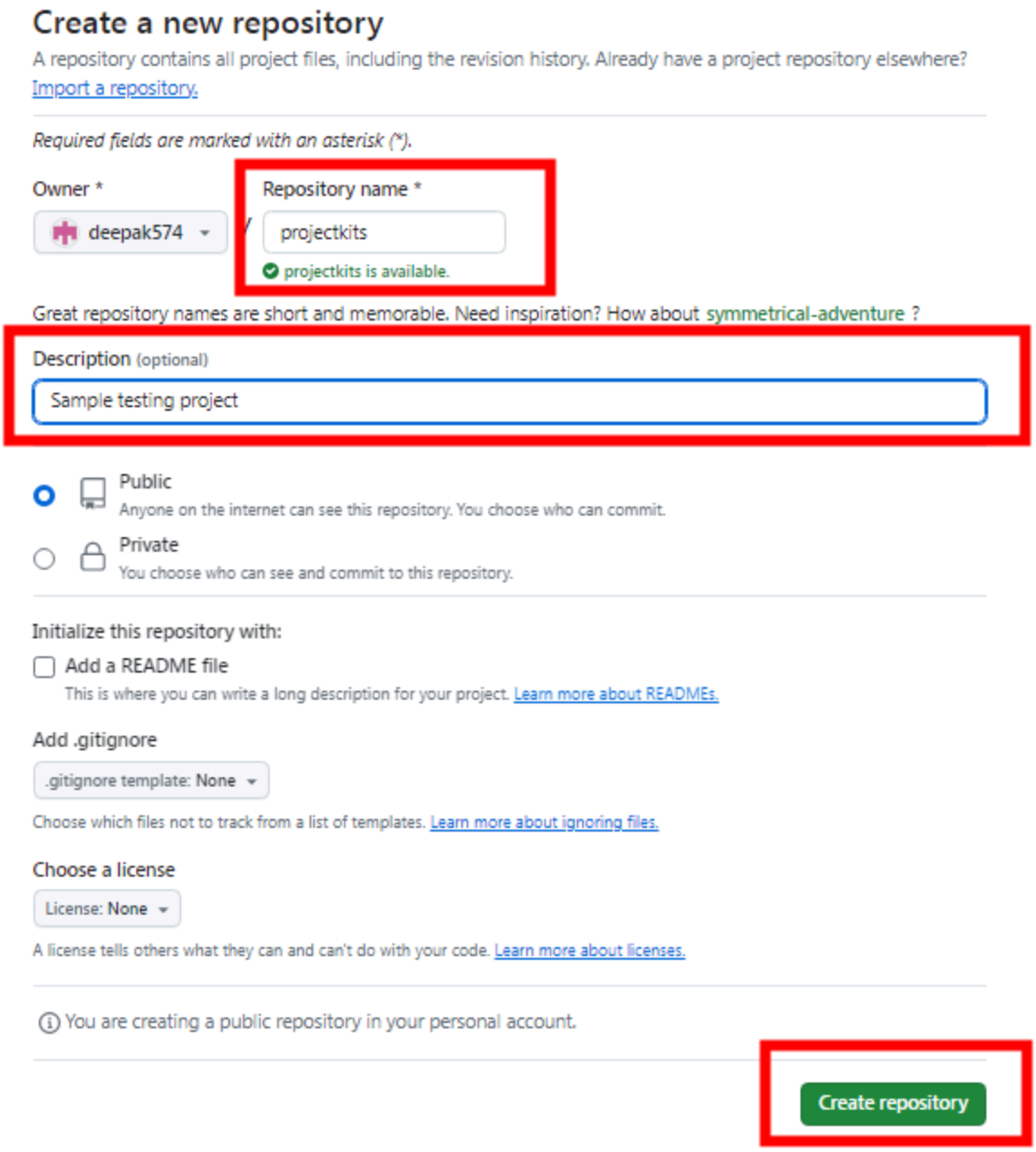
Being a Developer, I want to share this code to my peer developers. We have to send this code to the remote repository, so that it is available for them.
We have to configure the remote repository to our local repository. For this we have to use git remote command.
Open Github website and sign in

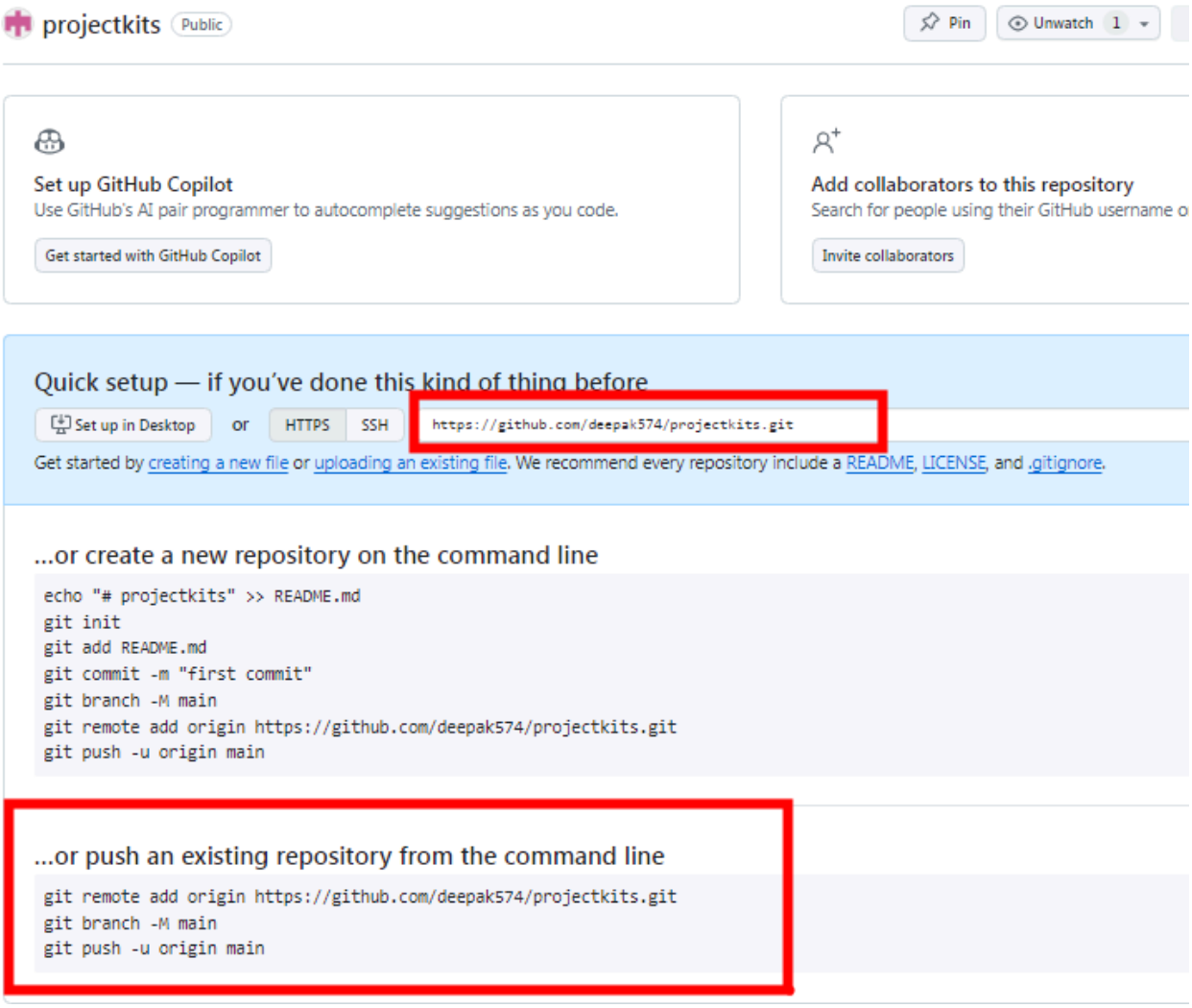


After successfully logged in click on + symbol and select New repository




Create a new repository
A repository contains all project files, including the revision history. Already have a project repository elsewhere?
Import a repository.
Required fields are marked with an asterisk (*).
Owner *
deepak574
Repository name *
projectkits
projectkits is available.
Great repository names are short and memorable. Need inspiration? How about symmetrical-adventure ?
Description (optional)
Sample testing project
Public
Anyone on the internet can see this repository. You choose who can commit.
Private
You choose who can see and commit to this repository.
Initialize this repository with:
Add a README file
This is where you can write a long description for your project. Learn more about READMEs.
Add .gitignore
.gitignore template: None
Choose which files not to track from a list of templates. Learn more about ignoring files.
Choose a license
License: None
A license tells others what they can and can't do with your code. Learn more about licenses.
You are creating a public repository in your personal account.
Create repository




This is the url of the remote repository
https://github.com/deepak574/projectkits.git
By using this url only we can communicate with remote repository.

**How to work with Remote Repository:**

To work with remote repository we have to use the following commands

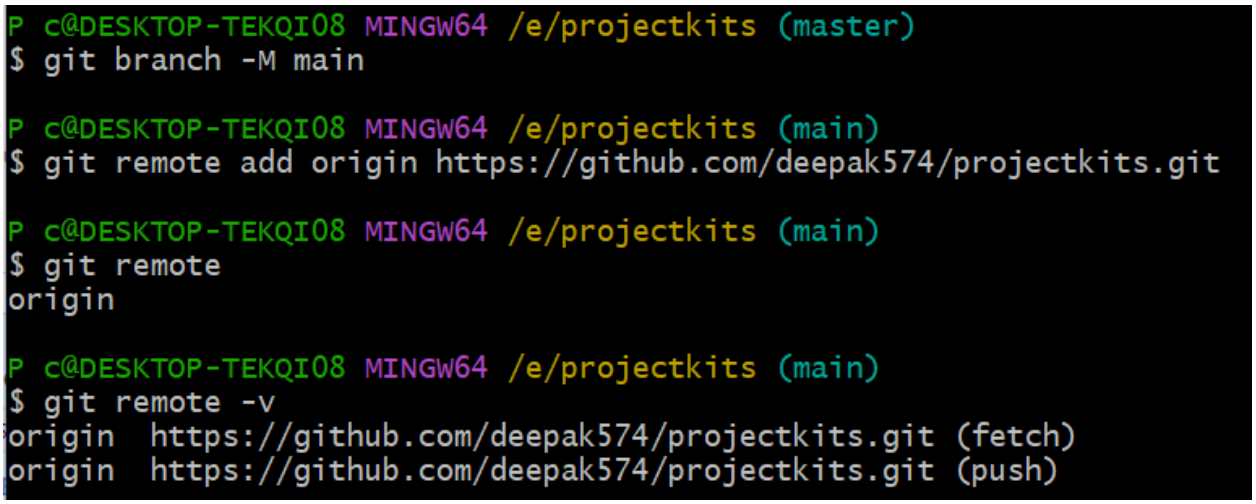
1. git remote
2. git push
3. git clone
4. git pull
5. git fetch

**git remote:**

We can use git remote command to configure remote repository to our local repository.

It just provides the alias names of remote repositories

$ git remote add <alias_name> remote_repository_url




**git push:**

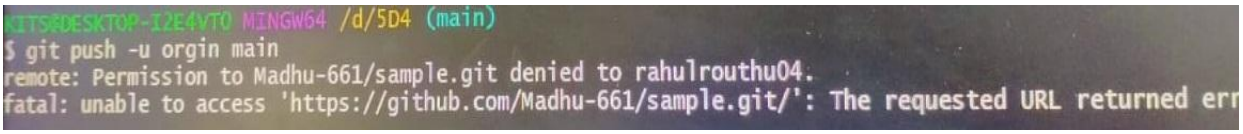
We can use the git push command to send our changes from local repository to remote repository. ie to push our changes from local repo to remote repo.

git push <remote_name> <branch_name>

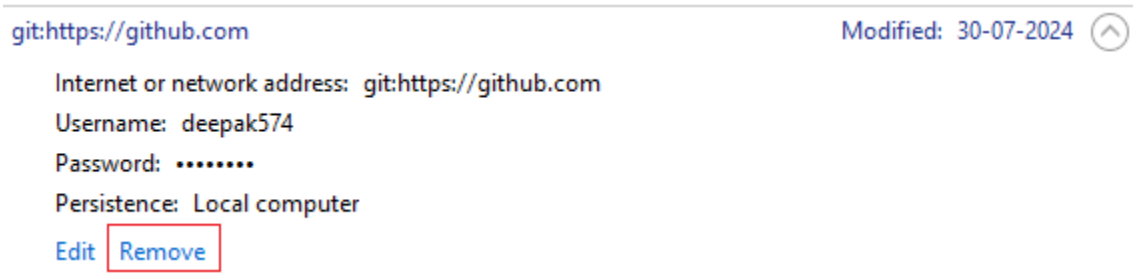
Eg: git push -u origin main

```
P c@DESKTOP-TEKQI08 MINGW64 /e/projectkits (main)
$ git push -u origin main
Enumerating objects: 6, done.
Counting objects: 100% (6/6), done.
Delta compression using up to 4 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (6/6), 465 bytes | 155.00 KiB/s, done.
Total 6 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
To https://github.com/deepak574/projectkits.git
 * [new branch]      main -> main
branch 'main' set up to track 'origin/main'.
```

**Note : If you get following error then do the following steps**

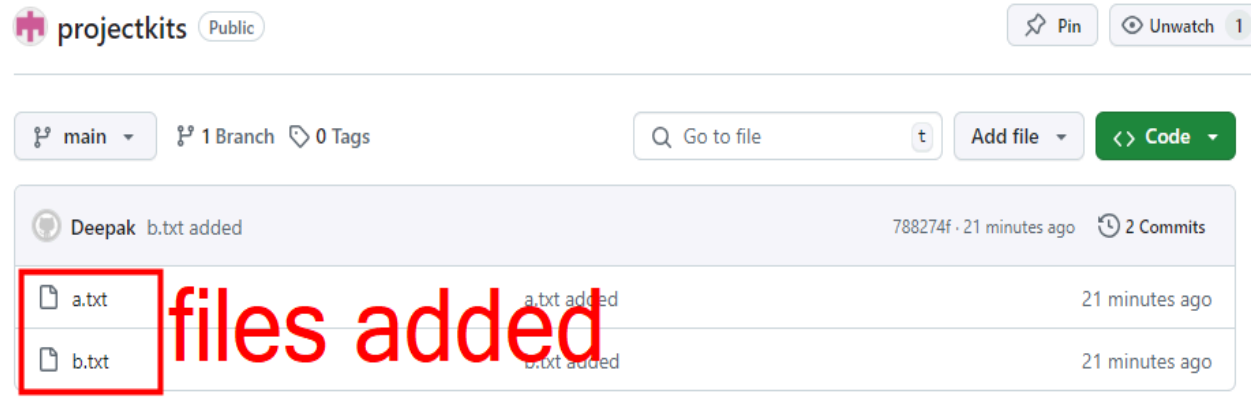



Windows OS : Control Panel → Credential Manager → Windows Credentials → Find and remove the GitHub-related credentials.




After removing we need to configure mail and name using the git config command.

Refresh the github page and will get the files



**Pull command:**

We can use git pull command to download and merge updates from remote repository into a local repository.

git pull = fetch+merge

Syntax:

$ git pull <remote> <branch>

$ git pull origin master


**git clone:**

Cloning means creating exactly duplicate copy.

As a new team member, you may need to set up your local development environment by copying the remote repository into your local system. This is done using the git clone command. The git clone command creates a local repository on your machine with all the files and commit history from the remote repository. This ensures that you have a complete copy of the project's current state and its entire history. To do this, simply navigate to your desired directory in the terminal and use the git clone command followed by the repository URL.

Syntax: git clone <remote_repo_url>

**Example**

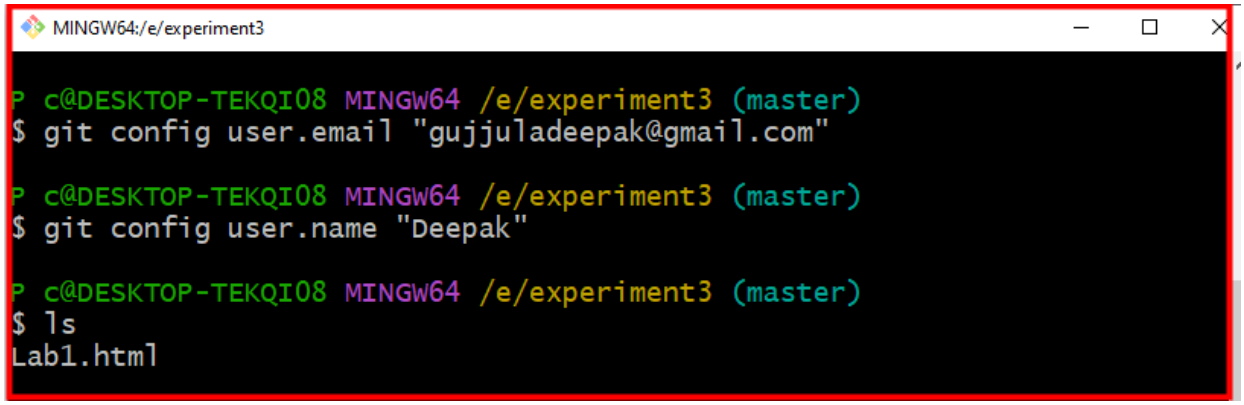
git clone https://github.com/deepak574/github_project.git

Now the project name will become the repository project name.

**Note:** Based on our requirement we can provide a new name for the project.
git clone <remote_repo_url> <new_project_name>
Eg: git clone https://github.com/deepak574/github_project.git my_project

**3. Practice Source code management on GitHub. Experiment with the source code in exercise 1.**

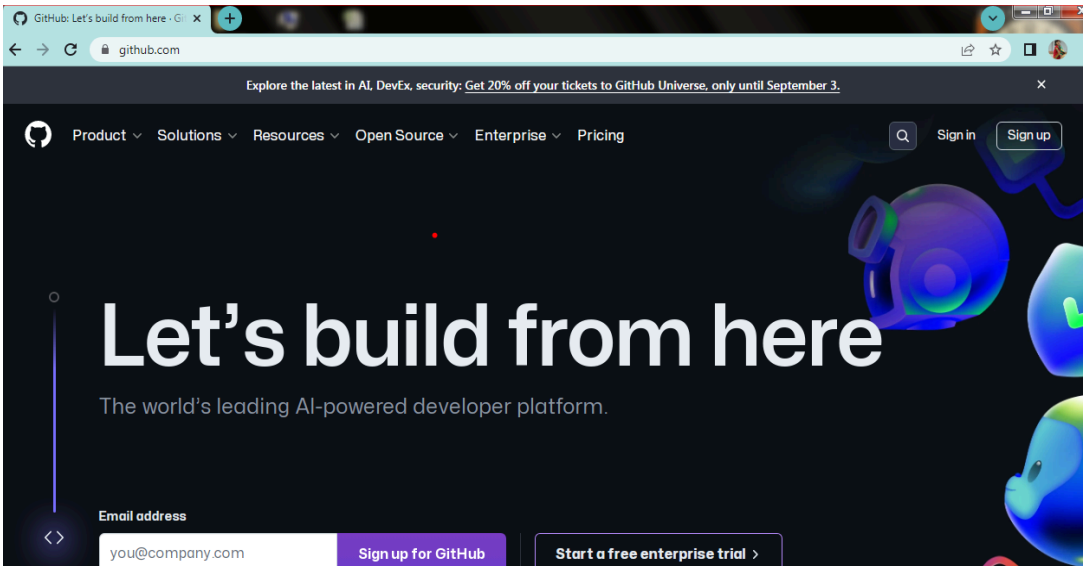
Create **experimnent3** folder in d drive.
Copy and paste the Lab1.html program in experimnent3 folder.

```
MINGW64:/e/experiment3
Lab1.html

P c@DESKTOP-TEKQI08 MINGW64 /e/experiment3 (master)
$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        Lab1.html

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

P c@DESKTOP-TEKQI08 MINGW64 /e/experiment3 (master)
$ git add .
```

```
P c@DESKTOP-TEKQI08 MINGW64 /e/experiment3 (master)
$ git commit -m "git and github commands on lab program1"
[master (root-commit) b92291e] git and github commands on lab program1
 1 file changed, 84 insertions(+)
 create mode 100644 Lab1.html

P c@DESKTOP-TEKQI08 MINGW64 /e/experiment3 (master)
$ git branch -M main
```

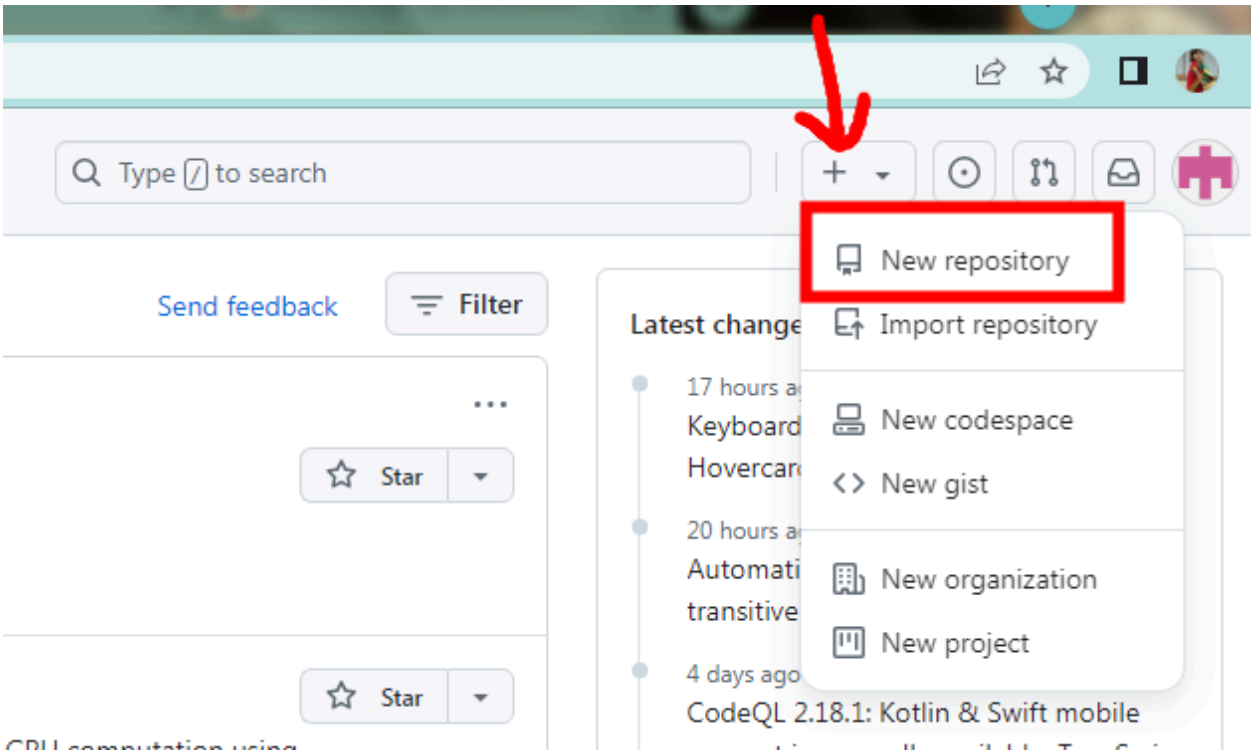
**Open the** https://github.com/ website

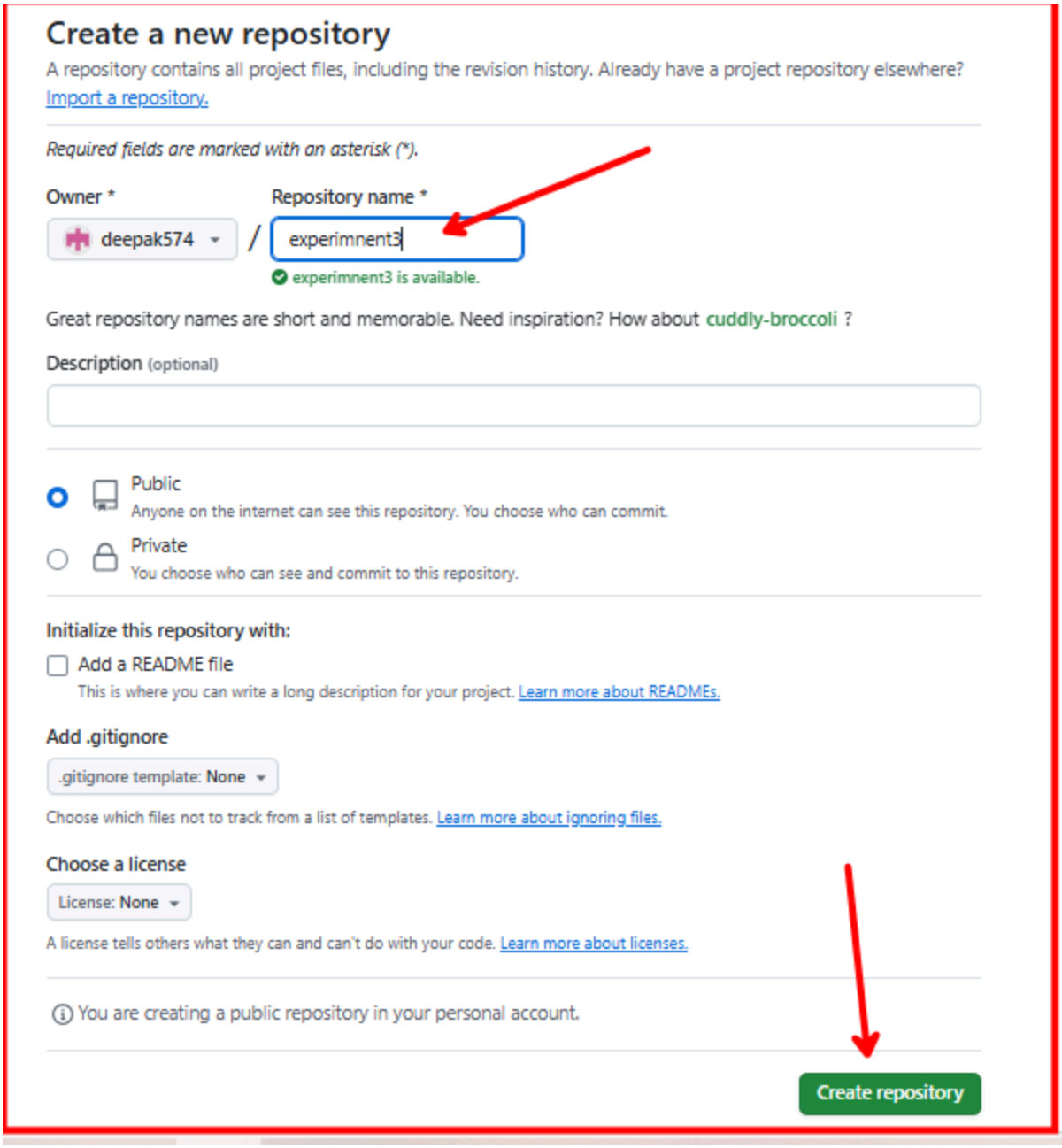


Select 'Sign Up' to create a new account. If you already have an account, click on 'Sign In'.

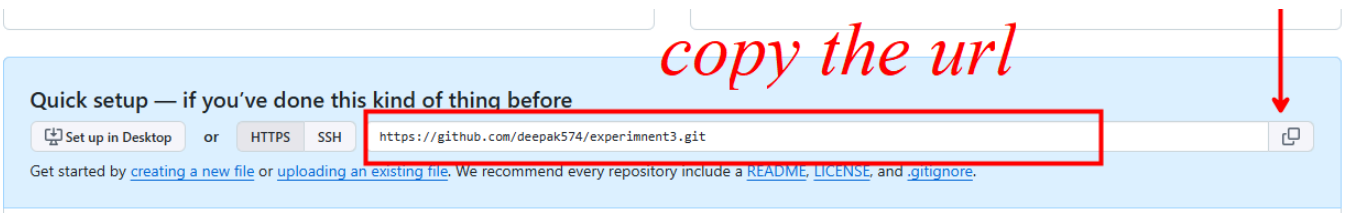
We should use a valid email ID because we need to verify it to create an account successfully. GitHub requires email verification.

After successfully logged in click on + symbol and select New repository



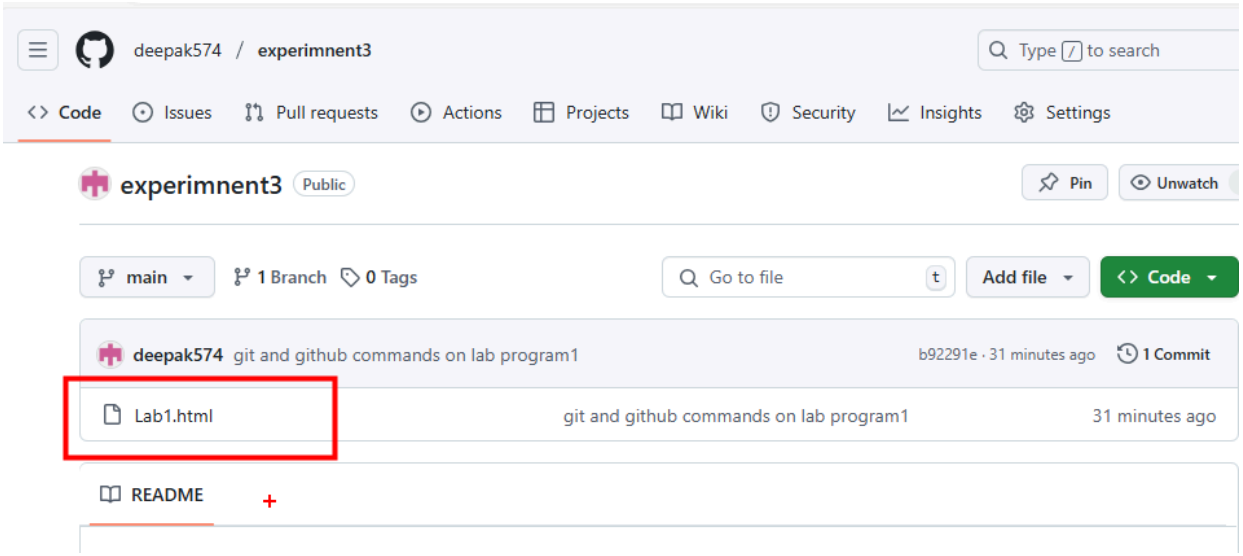



Give repository name as **experimnent3.**
And click on **create repository**



```
P c@DESKTOP-TEKQI08 MINGW64 /e/experiment3 (main)
$ git remote add origin https://github.com/deepak574/experimnent3.git

P c@DESKTOP-TEKQI08 MINGW64 /e/experiment3 (main)
$ git push -u origin main
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Delta compression using up to 4 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 958 bytes | 479.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
To https://github.com/deepak574/experimnent3.git
 * [new branch]      main -> main
branch 'main' set up to track 'origin/main'.
```

**Refresh the github page .**



**4. Jenkins installation and setup, explore the environment.**

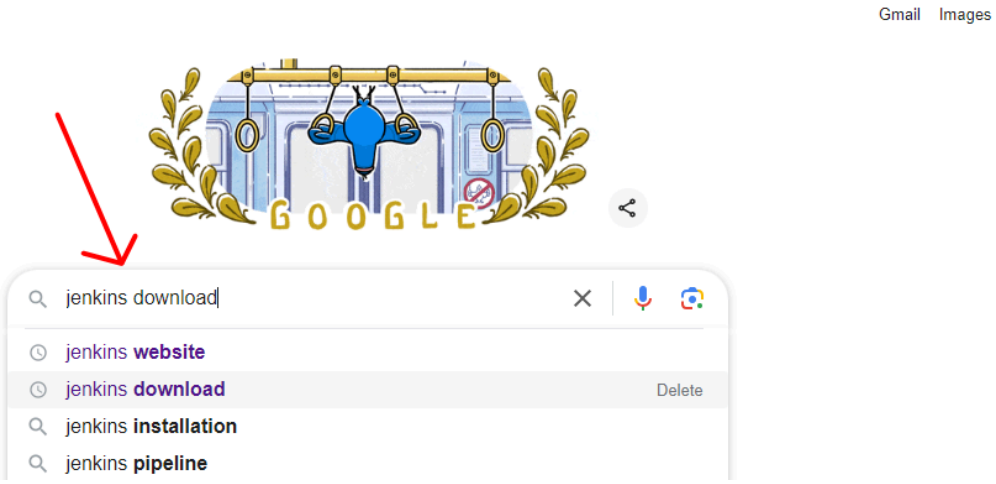
Detailed Guide to Install and Set Up Jenkins on Windows

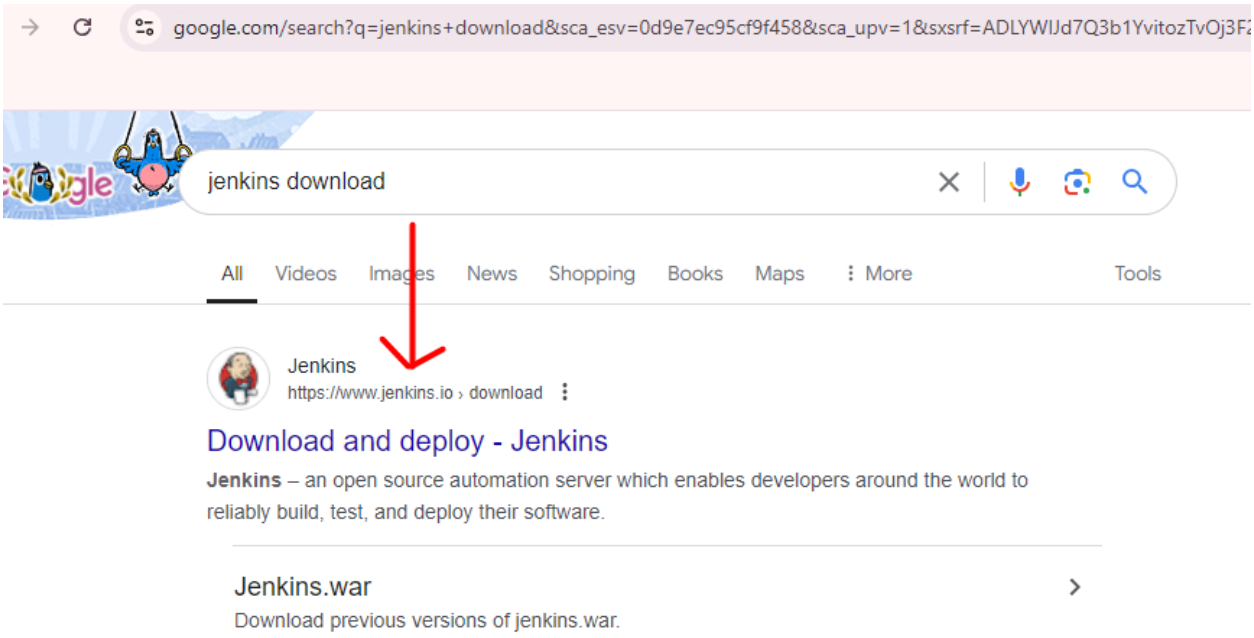
**Prerequisites**

- ❖ Ensure you have Java installed. Jenkins requires Java 8 or Java 11.
  - ➢ Download and install Java from the Oracle website.
  - ➢ Set the JAVA_HOME environment variable:
    - Right-click on "This PC" or "My Computer" and select "Properties".
    - Click on "Advanced system settings".
    - Click on "Environment Variables".
    - Under "System variables", click "New" and add JAVA_HOME with the path to your Java installation (e.g., C:\Program Files\Java\jdk-11).

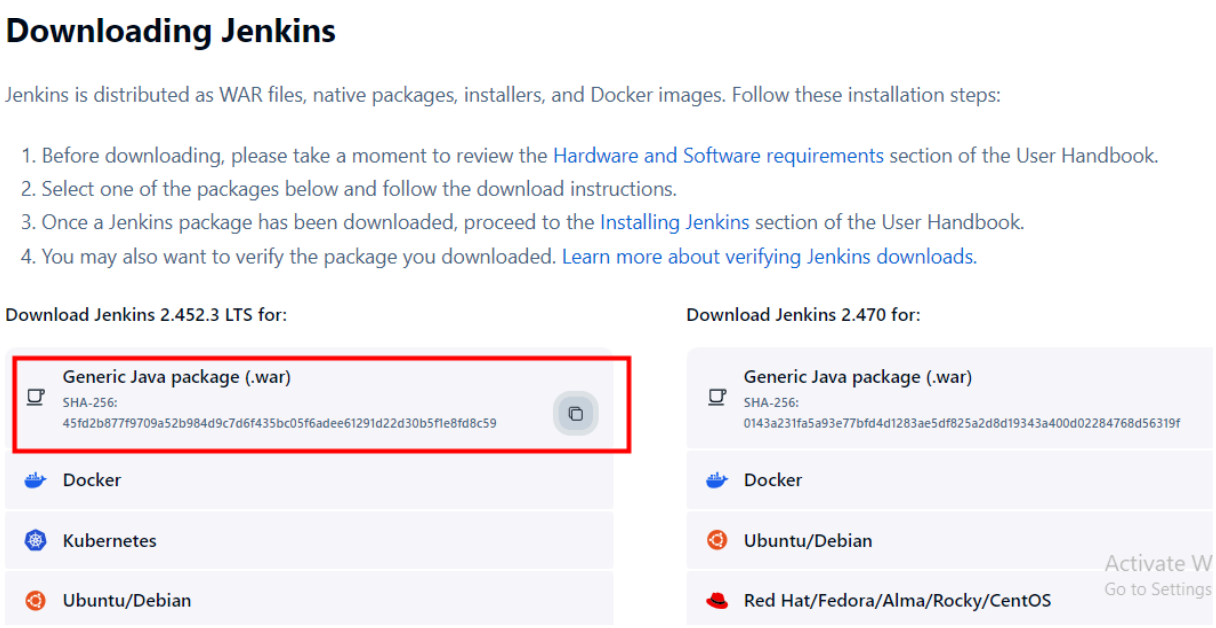
2. **Download Jenkins**

**Visit Jenkins website**

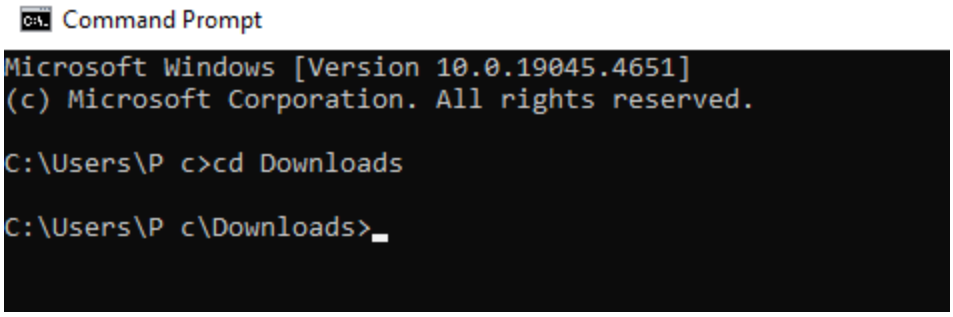




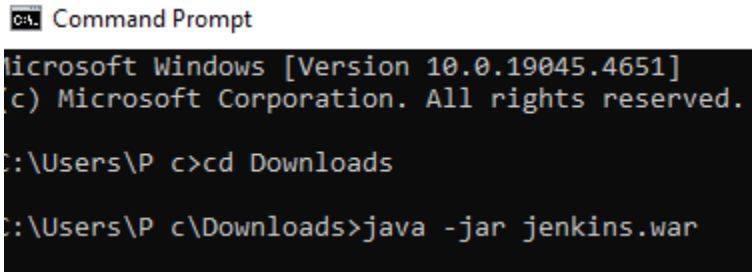

Go to the Jenkins website **( https://www.jenkins.io/ )**
The Jenkins project produces two release lines: Stable (LTS) and weekly. We use Stable (LTS)



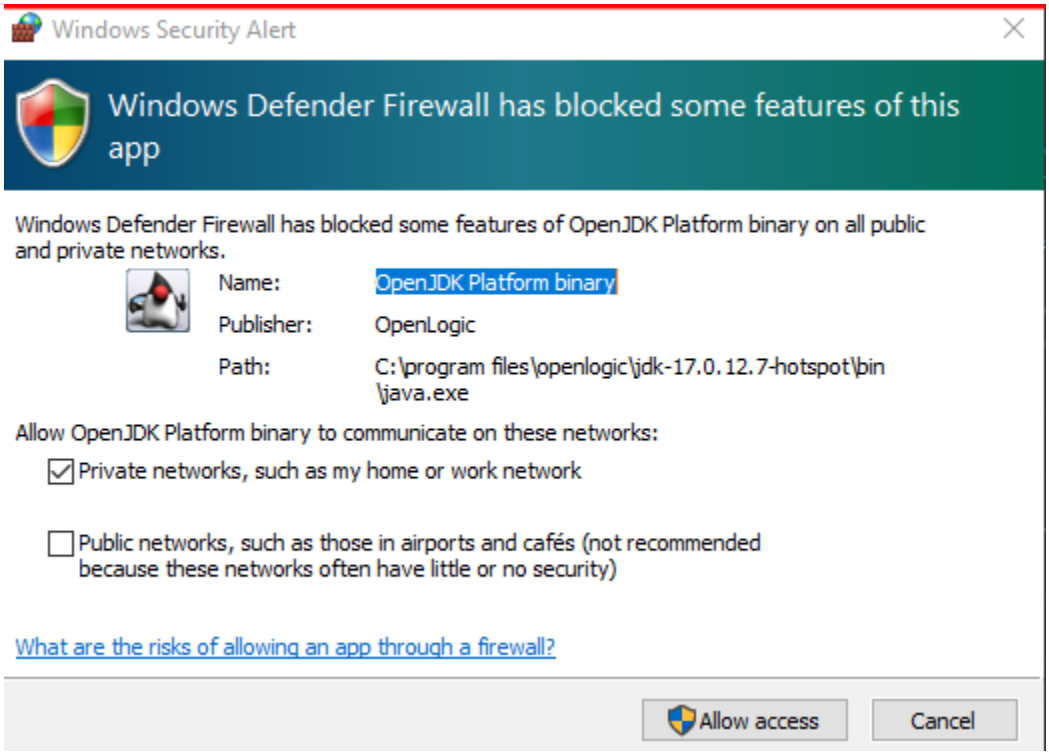

Click on Generic Java package
jenkins.war file will downloaded and saved in downloads folder
Once the jenkins.war is downloaded
Open command prompt and go the downloads folder



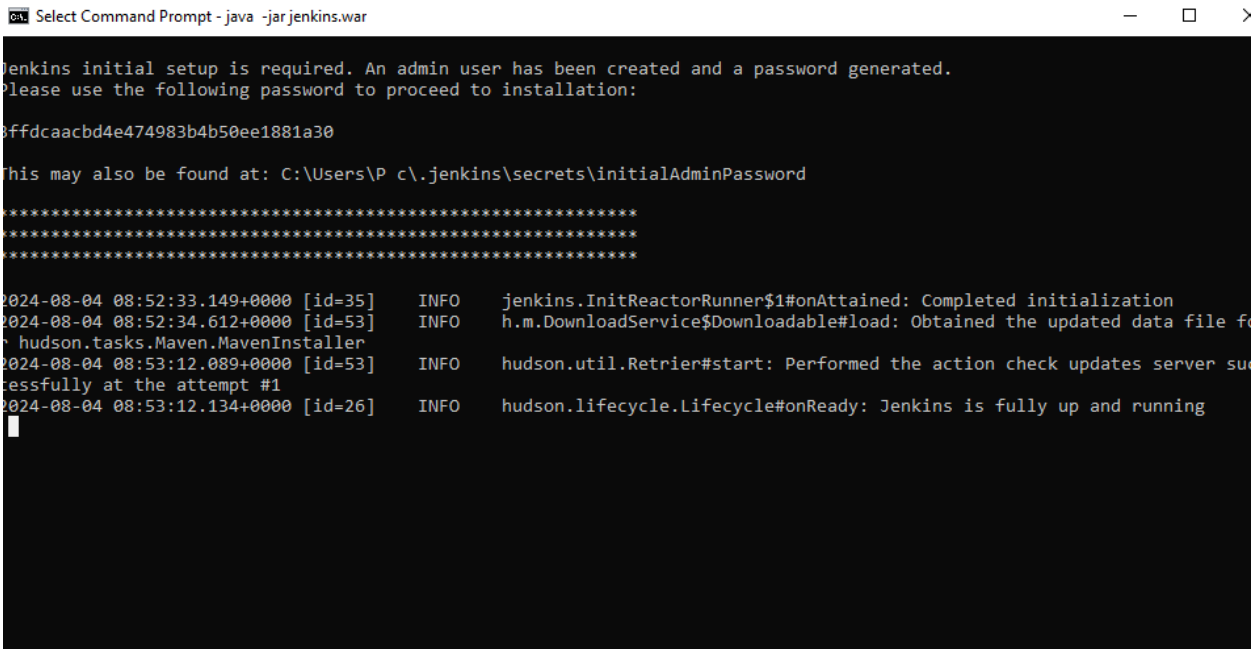

Type the following command
***java -jar jenkins.war*** and hit enter



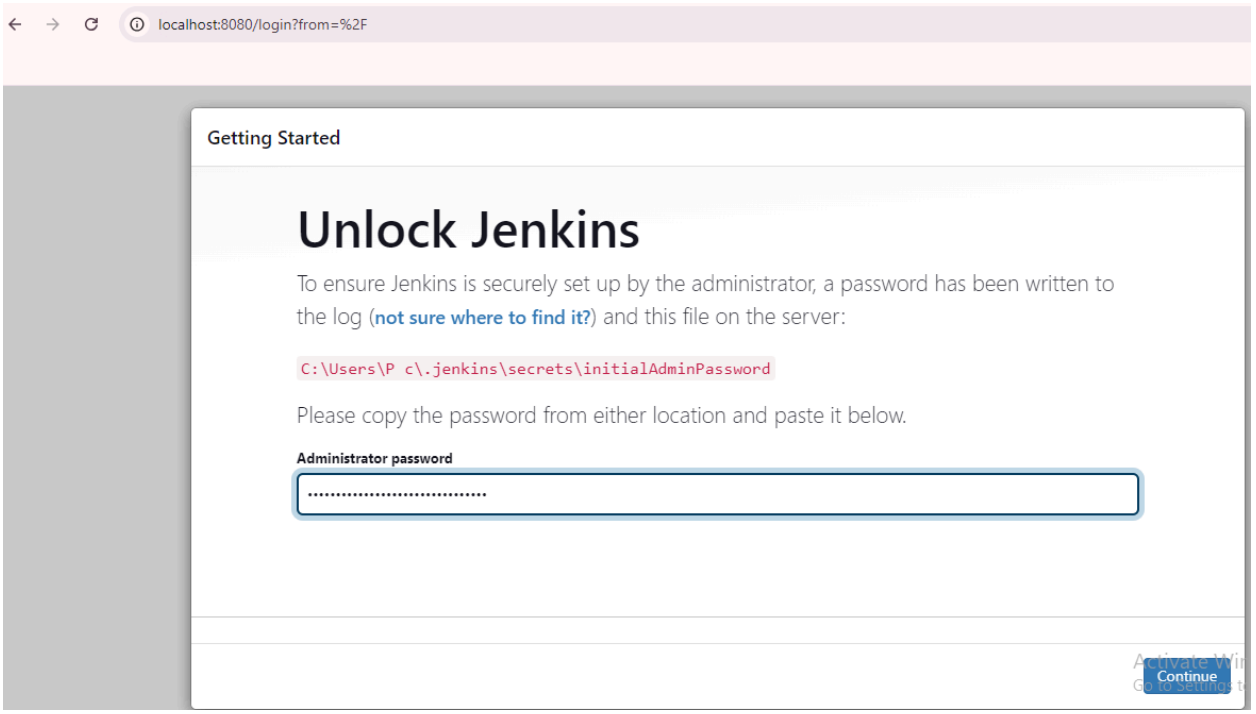

If you get below images then click on **Allow access.**



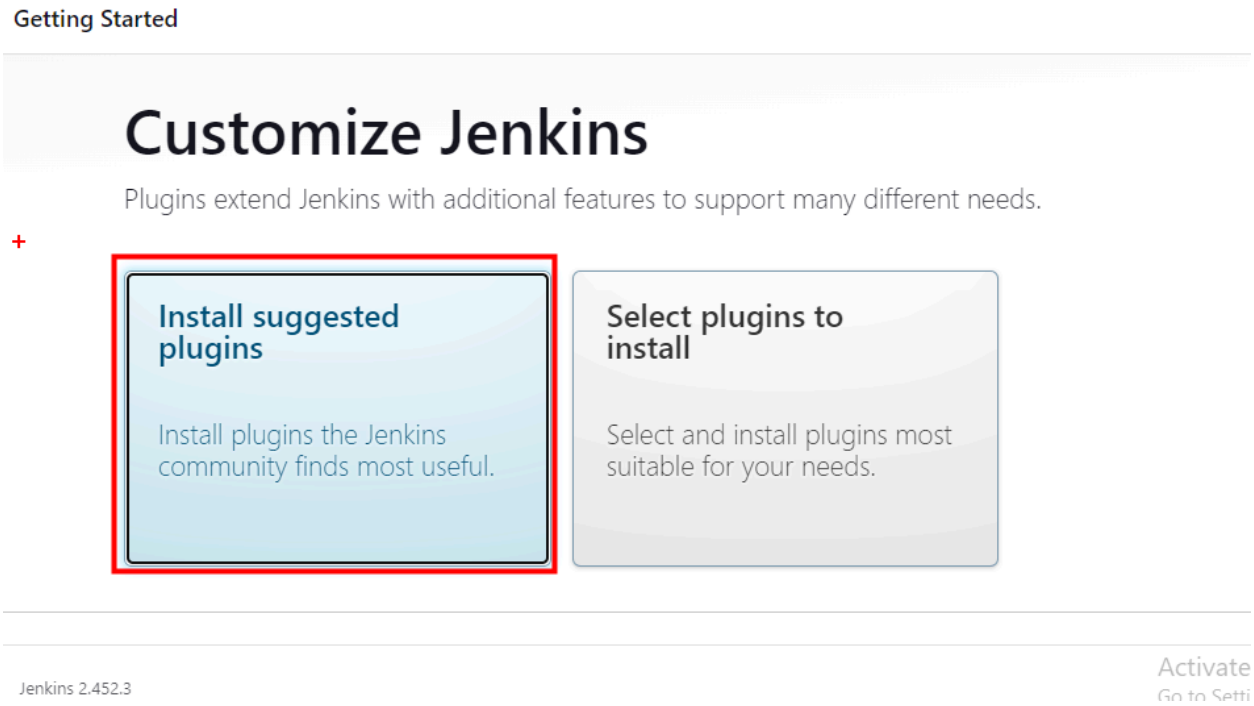
In the below image we get a message saying ***jenkins up and running,*** it generates one password in the ***C:\Users\P c\.jenkins\secrets\initialAdminPassword*** file.
Minimize the command prompt.
Open the initialPassword file and copy the password.



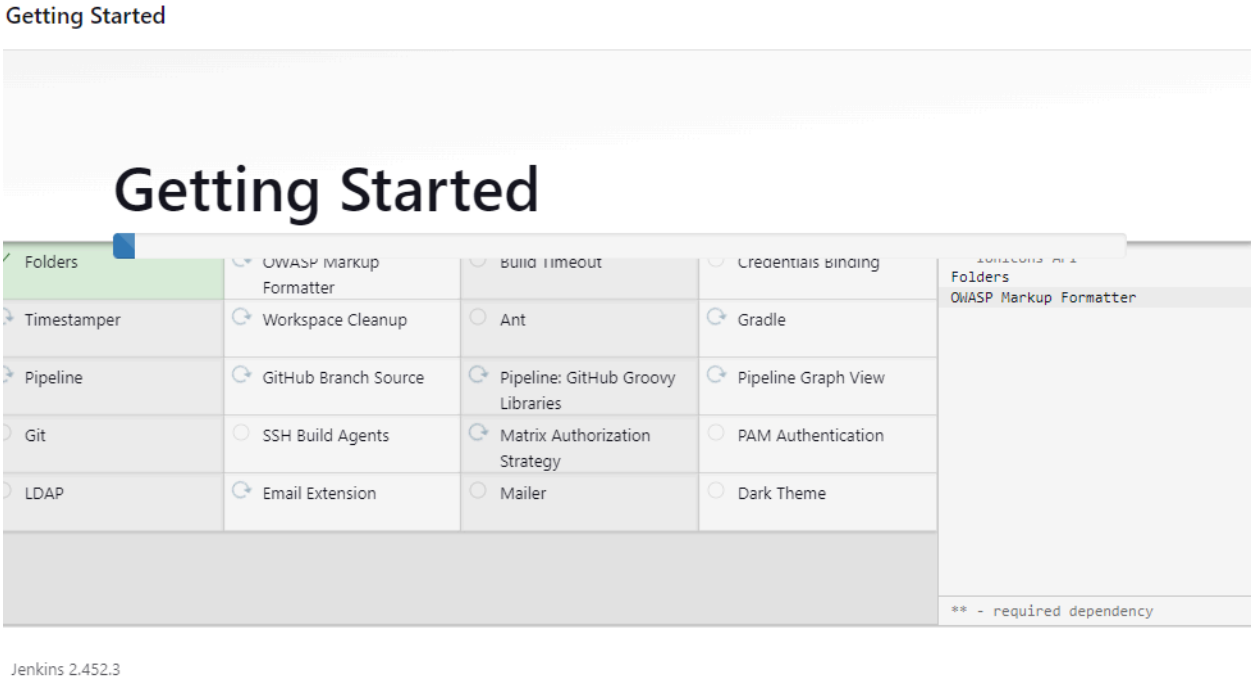

Paste the password in the Administrator password textfield and hit continue.



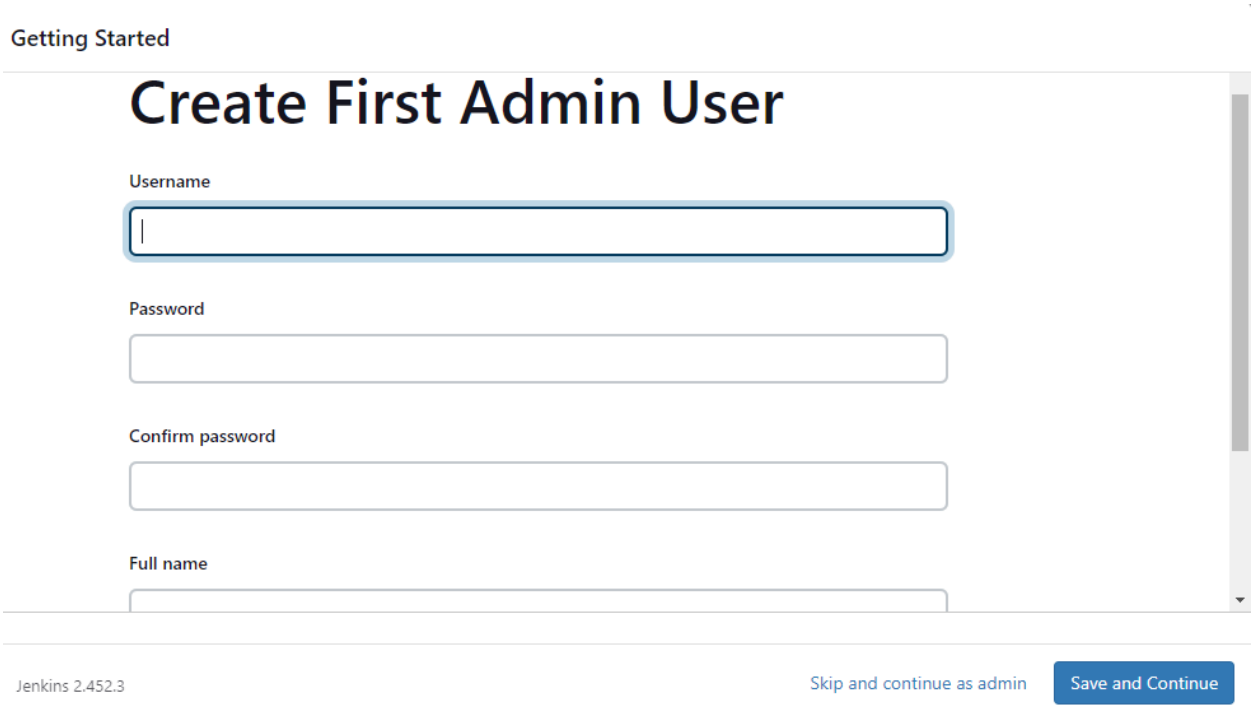
In the below image select and press Install suggested plugins option.

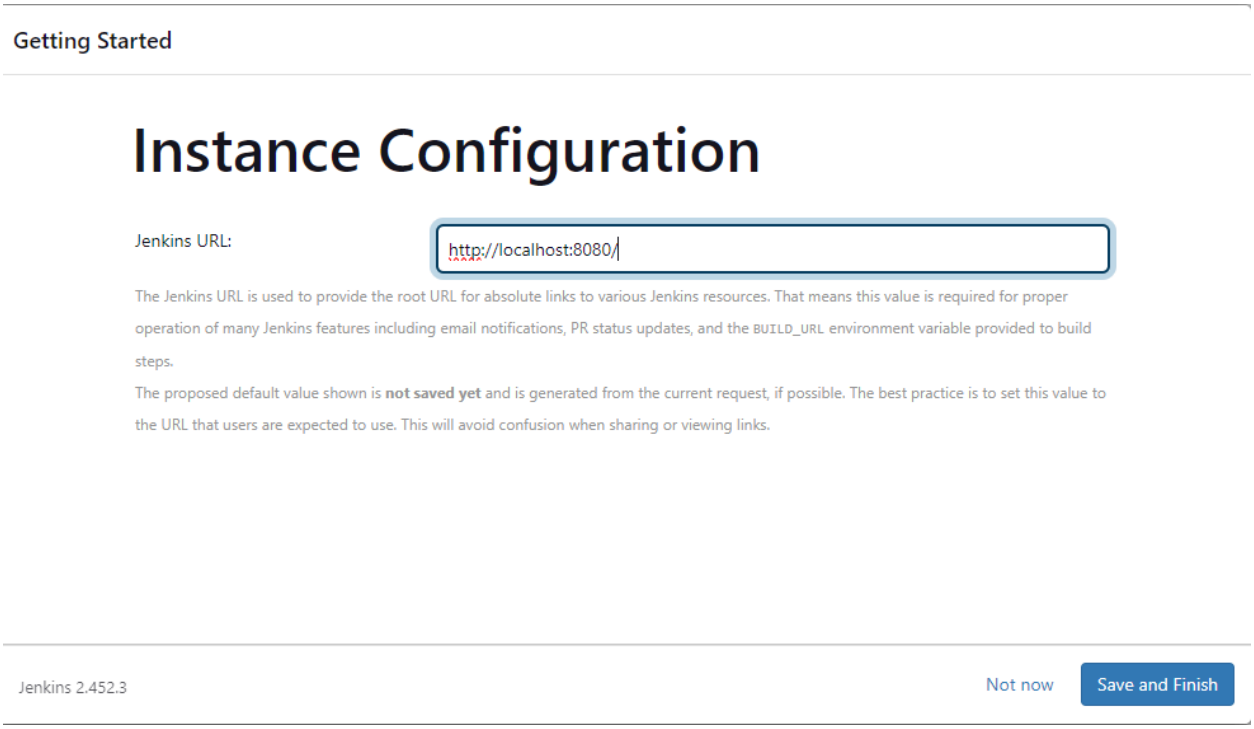



**Wait until all the plugins are installed.**

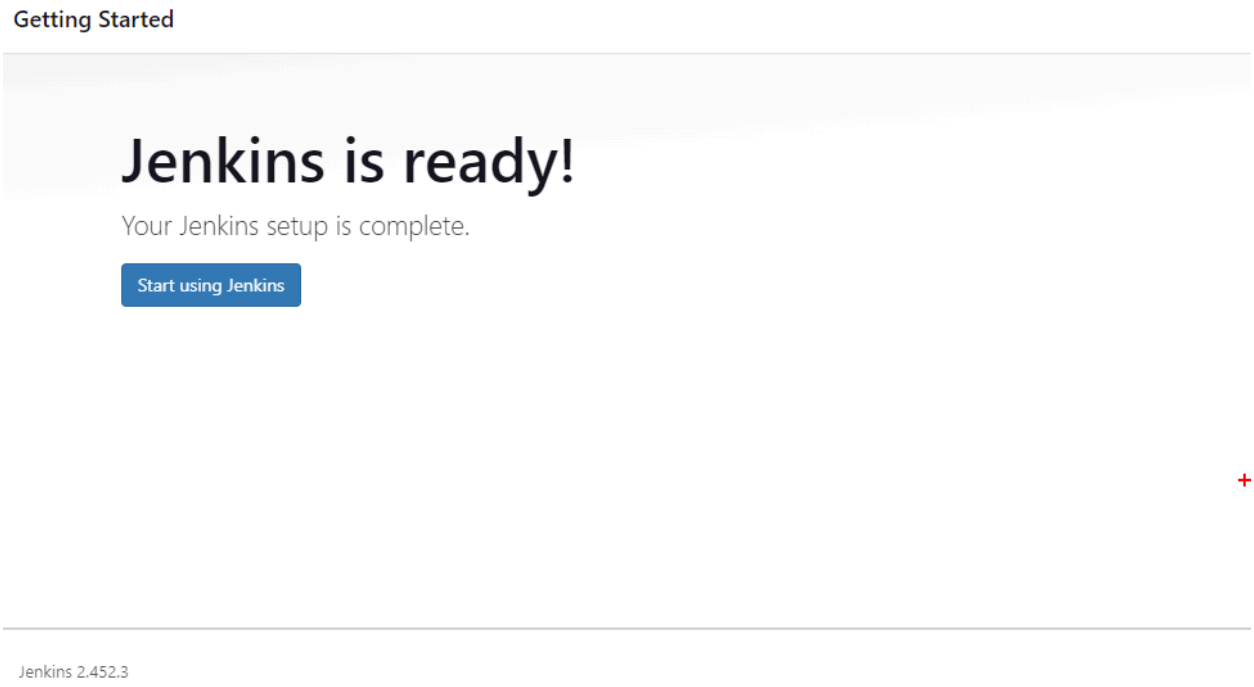



Create the Username , password and all necessary fields

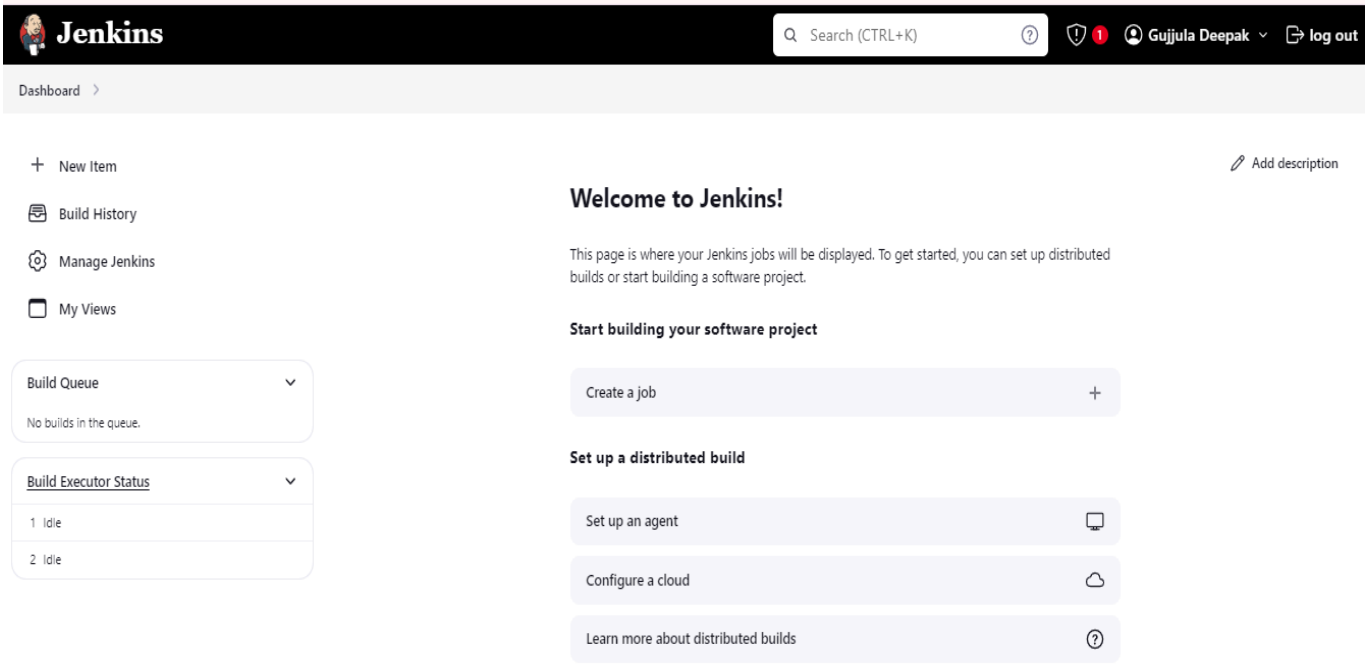



Save the Finish the process




Jenkins are successfully installed.When you hit start using jenkins button you will get the below dashboard page



**Exploring the Jenkins Environment in Detail**

**Dashboard**

**New Item:**

Create new projects in Jenkins

**Types of Projects:**

1. **Freestyle Project:** A flexible and easy-to-setup project type. You can define simple jobs and chains of jobs.
2. **Pipeline:** Allows you to define your build process as code. You write a script in Groovy language to automate your workflow.
3. **Multi-configuration Project:** Used for running the same job with different configurations (e.g., testing on multiple platforms).
4. **Folder:** Organize multiple jobs into a single folder.
5. **GitHub Organization:** Automatically create Pipeline jobs for each repository in a GitHub organization.
6. **Multibranch Pipeline:** Automatically create a pipeline for each branch in a repository.

**Example:**

- To create a new Freestyle project:
  1. Click on "New Item".
  2. Enter the name of the project.
  3. Select "Freestyle project" and click "OK".
  4. Configure the project (e.g., source code management, build triggers, build steps).
  5. Save the project.

**Build History:**

Track and display the status of recent builds.

Information Available:

- ➢ Build number.
- ➢ Project name.
- ➢ Build status (success, failure, unstable).
- ➢ Timestamp of the build.

**Example:**

On the main dashboard, you can see a list of recent builds under "Build History". Clicking on any build number will provide detailed information about that build.

**Manage Jenkins:**

Access to administration and configuration options for Jenkins.

**Options Available:**

- Configure System: Set global configurations.
- Global Tool Configuration: Manage tools (e.g., JDK, Git, Maven).
- Manage Plugins: Install, update, or remove plugins.
- Security: Configure global security settings.
- System Information: View detailed information about the Jenkins environment.
- Manage Nodes: Manage slave nodes for distributed builds.
- Script Console: Run Groovy scripts for advanced configuration and automation.

**Example:**

Navigate to "Manage Jenkins" to see all the available administration options.

**5. Demonstrate continuous integration and development using Jenkins.**

Open Git Bash

MINGW64:/e/jenkinsproject

```
P c@DESKTOP-TEKQI08 MINGW64 ~ (master)
$ cd e:

P c@DESKTOP-TEKQI08 MINGW64 /e
$ mkdir jenkinsproject

P c@DESKTOP-TEKQI08 MINGW64 /e
$ cd jenkinsproject

P c@DESKTOP-TEKQI08 MINGW64 /e/jenkinsproject
$ git init
Initialized empty Git repository in E:/jenkinsproject/.git/

P c@DESKTOP-TEKQI08 MINGW64 /e/jenkinsproject (master)
$ git config user.email "gujjuladeepak@gmail.com"

P c@DESKTOP-TEKQI08 MINGW64 /e/jenkinsproject (master)
$ git config user.name "Deepak"
```

Write simple java program

Use the command ***vim Sample.java***

MINGW64:/e/jenkinsproject

```
P c@DESKTOP-TEKQI08 MINGW64 /e/jenkinsproject (master)
$ vim Sample.java
```

Text editor will be opened

Press ***i*** to write the code

MINGW64:/e/jenkinsproject

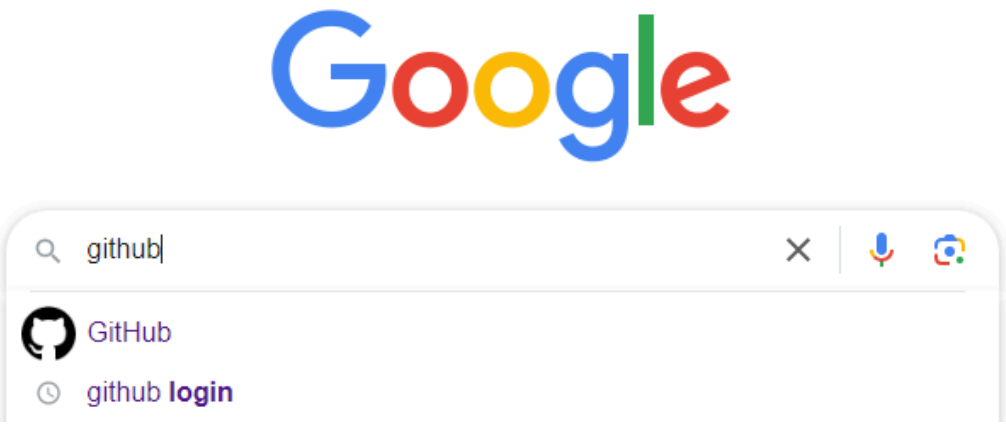
```
public class Sample
{
        public static void main(Strings args[])
        {
                System.out.print("Gujjula Deepak");
        }
}
```

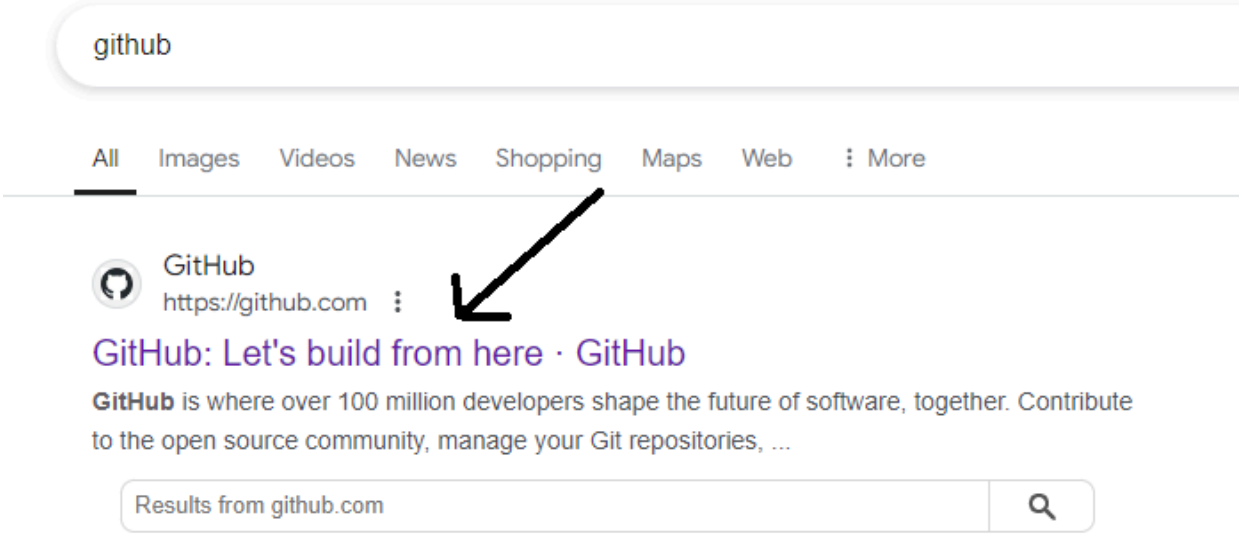
After writing the code press ***esc*** and type ***:wq*** and press enter

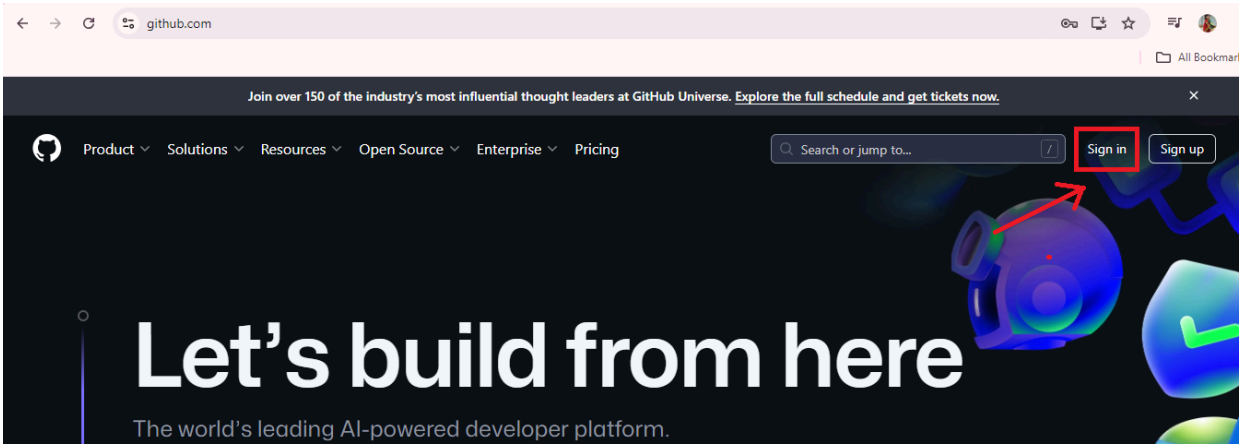
MINGW64:/e/jenkinsproject

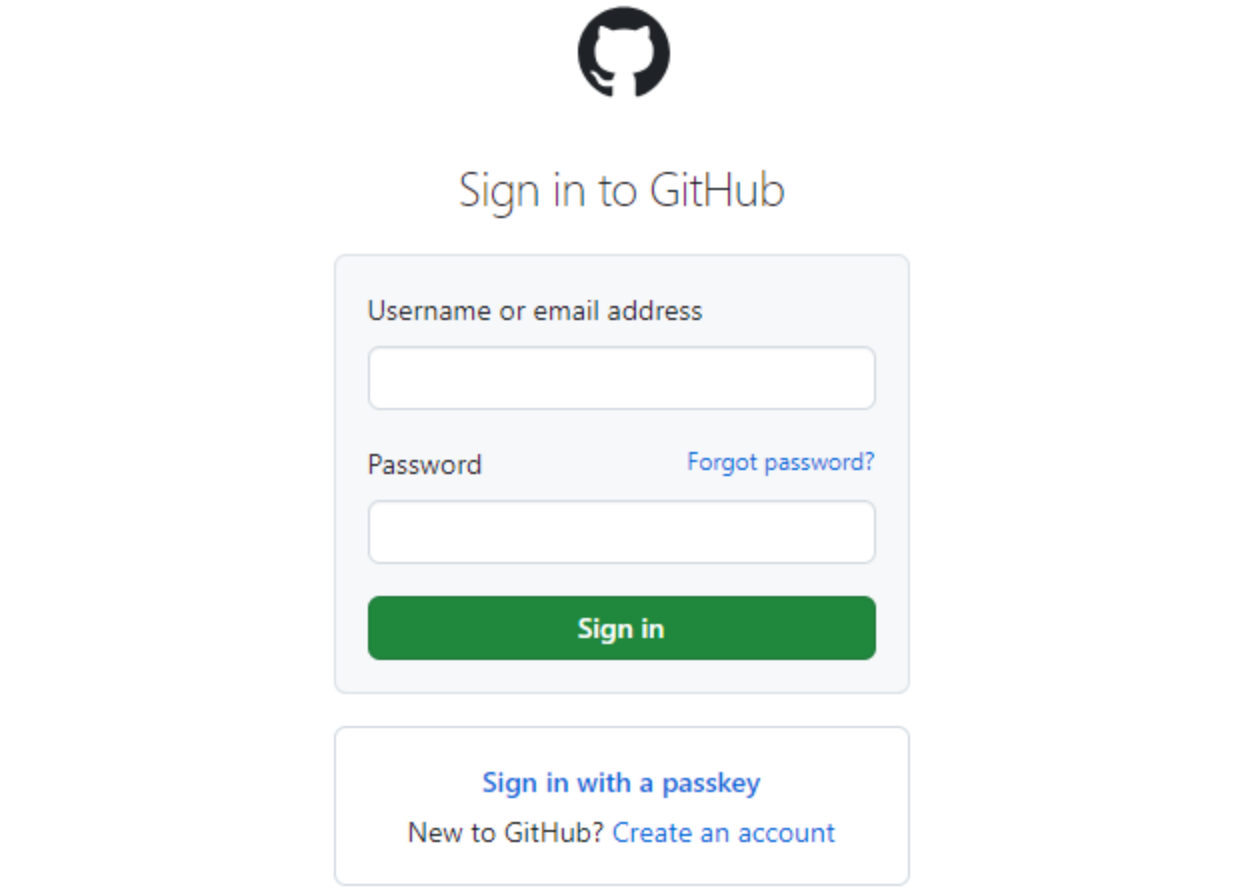
```
P c@DESKTOP-TEKQI08 MINGW64 /e/jenkinsproject (master)
$ git add .
warning: in the working copy of 'Sample.java', LF will be replaced

P c@DESKTOP-TEKQI08 MINGW64 /e/jenkinsproject (master)
$ git commit -m "Sample.java with content Gujjula Deepak"
[master (root-commit) 402d3ea] Sample.java with content Gujjula De
 1 file changed, 7 insertions(+)
 create mode 100644 Sample.java
```

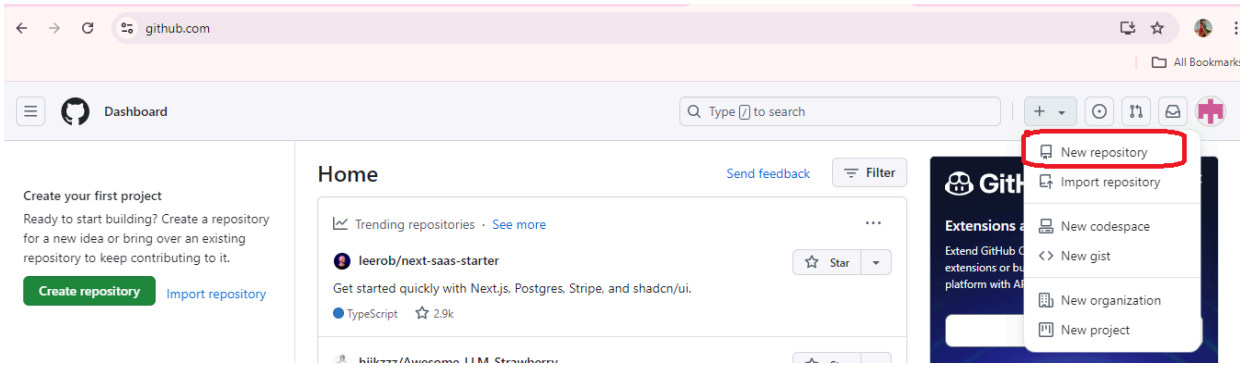
**Open any browser and Open github website**

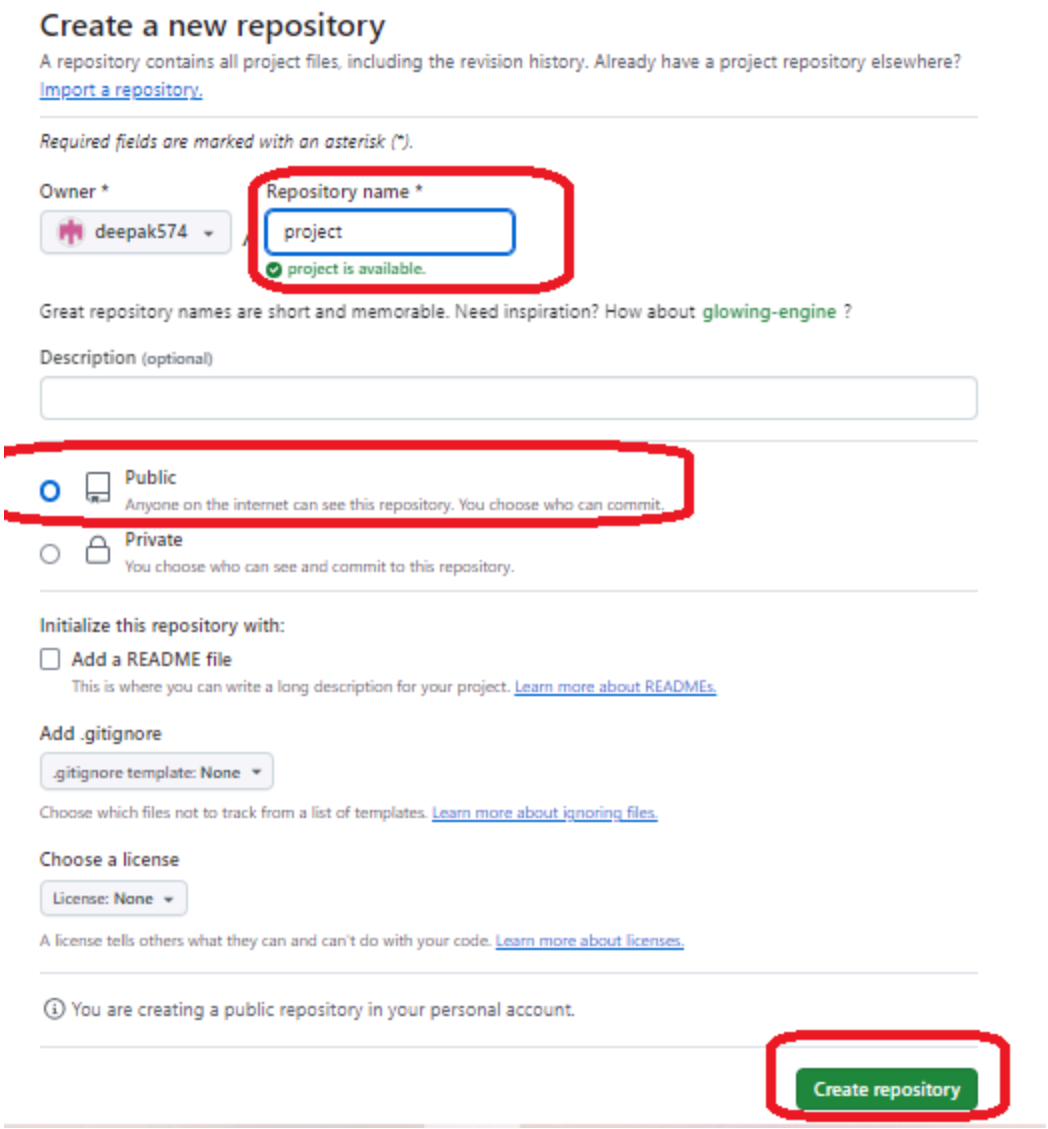


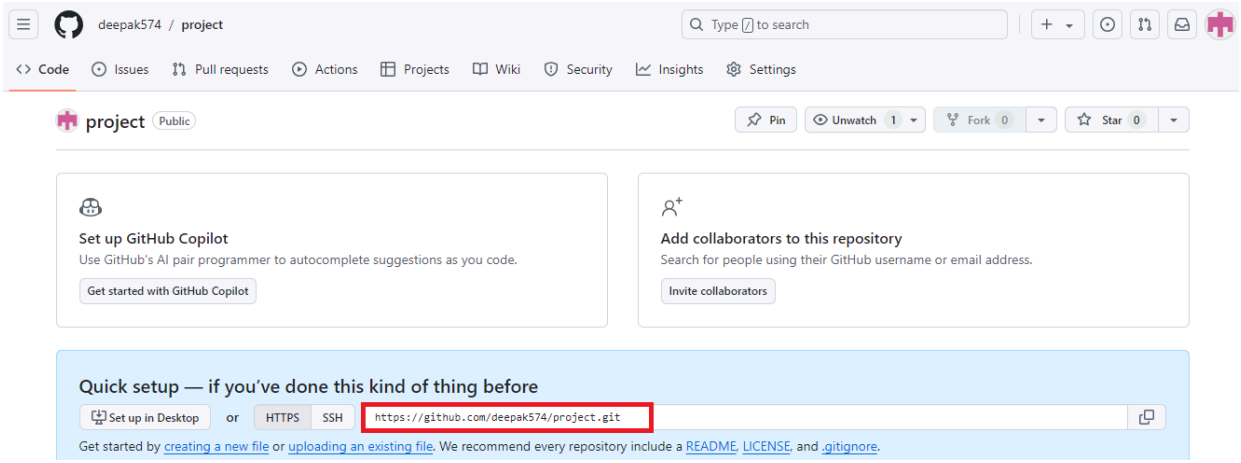

github
All
Images
Videos
News
Shopping
Maps
Web
More
GitHub
https://github.com
GitHub: Let's build from here · GitHub
GitHub is where over 100 million developers shape the future of software, together. Contribute to the open source community, manage your Git repositories, ...
Results from github.com


github.com
Join over 150 of the industry's most influential thought leaders at GitHub Universe. Explore the full schedule and get tickets now.
Product
Solutions
Resources
Open Source
Enterprise
Pricing
Search or jump to...
Sign in
Sign up
Let's build from here
The world's leading AI-powered developer platform.


Sign in to GitHub
Username or email address
Password
Forgot password?
Sign in
Sign in with a passkey
New to GitHub? Create an account

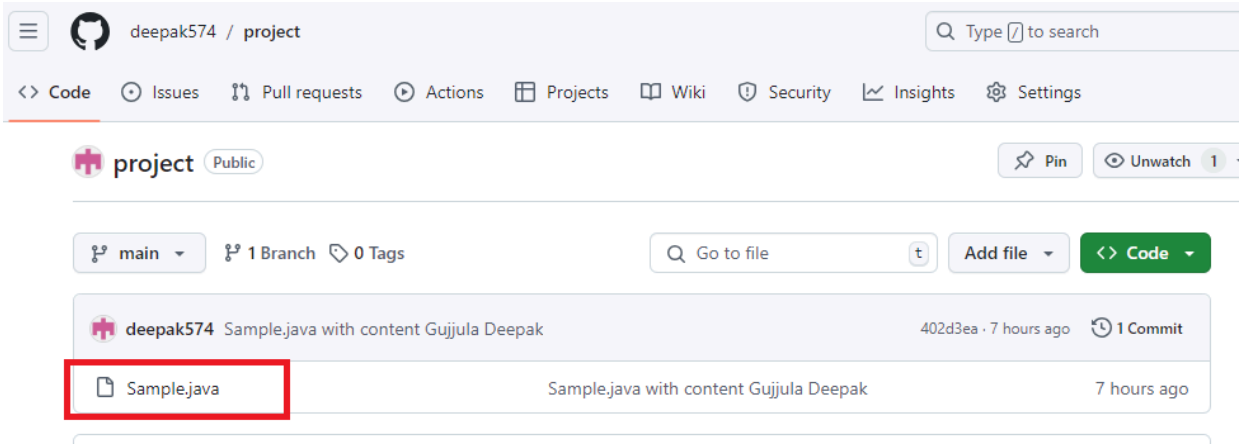

github.com
Dashboard
Type / to search
Create your first project
Ready to start building? Create a repository for a new idea or bring over an existing repository to keep contributing to it.
Create repository
Import repository
Home
Send feedback
Filter
Trending repositories · See more
leerob/next-saas-starter
Star
Get started quickly with Next.js, Postgres, Stripe, and shadcn/ui.
TypeScript
2.9k
New repository
Import repository
New codespace
New gist
New organization
New project


Create a new repository
A repository contains all project files, including the revision history. Already have a project repository elsewhere?
Import a repository.
Required fields are marked with an asterisk (*).
Owner *
deepak574
Repository name *
project
project is available.
Great repository names are short and memorable. Need inspiration? How about glowing-engine ?
Description (optional)
Public
Anyone on the internet can see this repository. You choose who can commit.
Private
You choose who can see and commit to this repository.
Initialize this repository with:
Add a README file
This is where you can write a long description for your project. Learn more about READMEs.
Add .gitignore
.gitignore template: None
Choose which files not to track from a list of templates. Learn more about ignoring files.
Choose a license
License: None
A license tells others what they can and can't do with your code. Learn more about licenses.
You are creating a public repository in your personal account.
Create repository

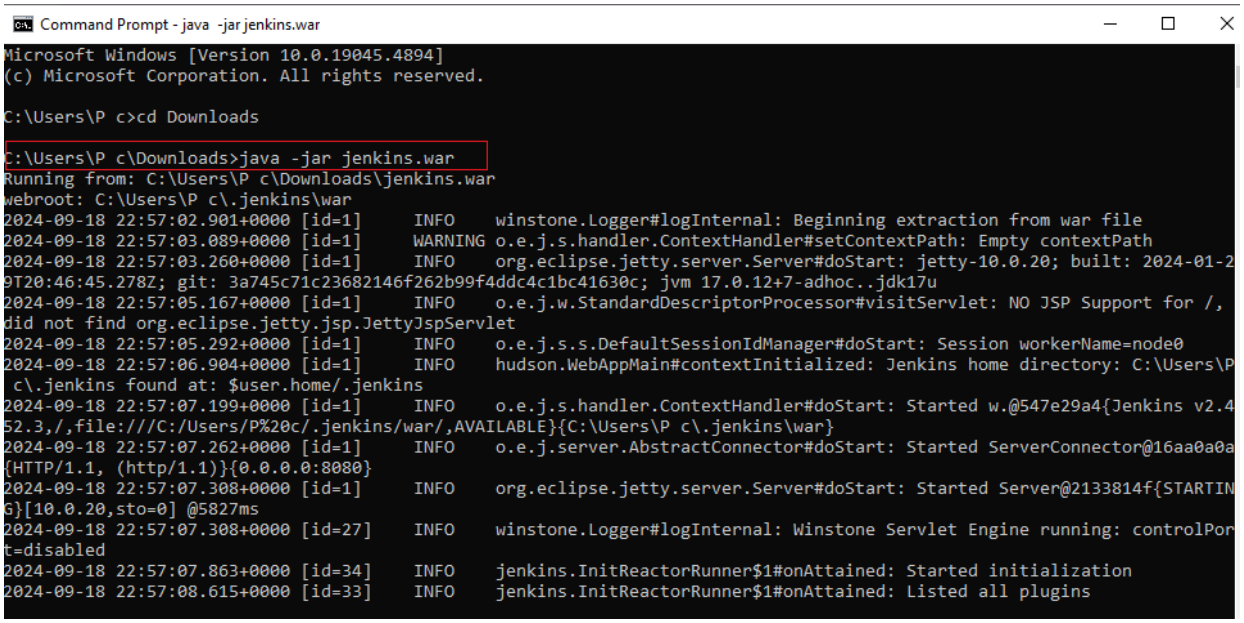



```
P c@DESKTOP-TEKQI08 MINGW64 /e/jenkinsproject (master)
$ git branch -m main

P c@DESKTOP-TEKQI08 MINGW64 /e/jenkinsproject (main)
$ git remote add origin https://github.com/deepak574/project.git

P c@DESKTOP-TEKQI08 MINGW64 /e/jenkinsproject (main)
$ git push -u origin main
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Delta compression using up to 4 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 333 bytes | 166.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
To https://github.com/deepak574/project.git
 * [new branch]      main -> main
branch 'main' set up to track 'origin/main'.
```

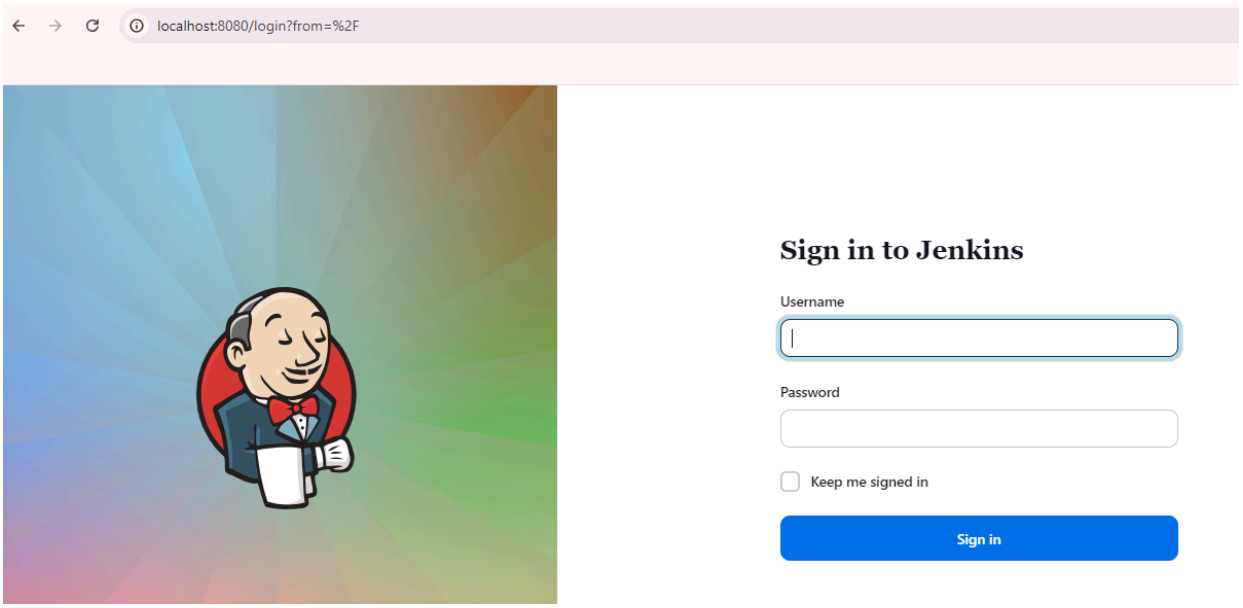
**Refresh the github website**



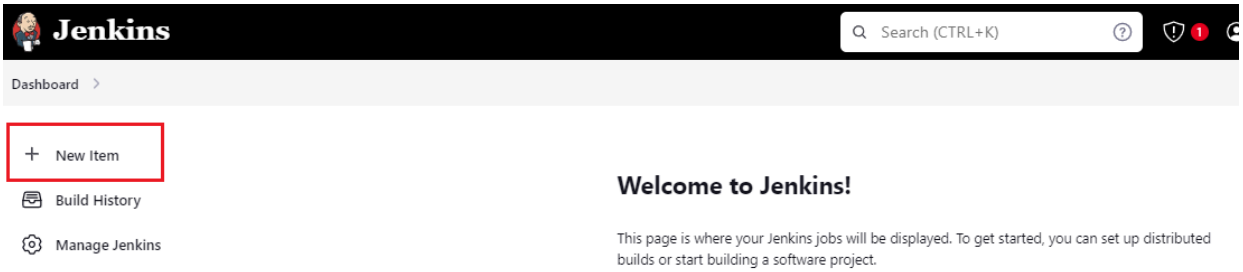

**Now we have to open jenkins**
**Open command prompt, go to the path where jenkins installed**



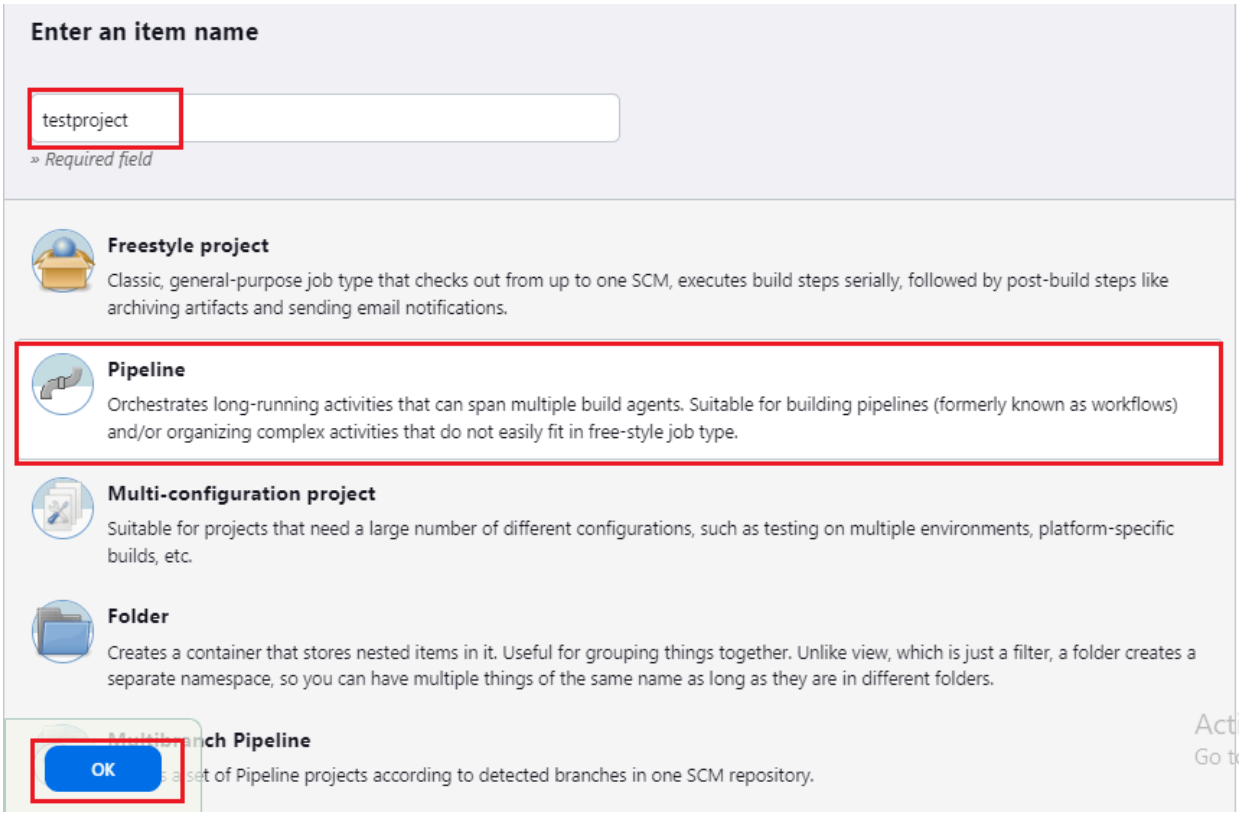

**Minimize this cmd prompt and open any browser**
**In the search bar type localhost:8080**



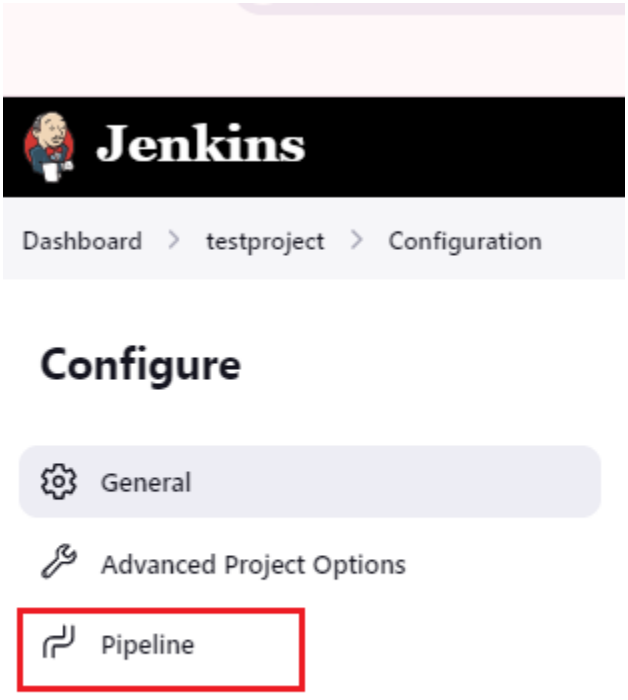

**Give the username , password and click sign in button**




**Click on New Item**



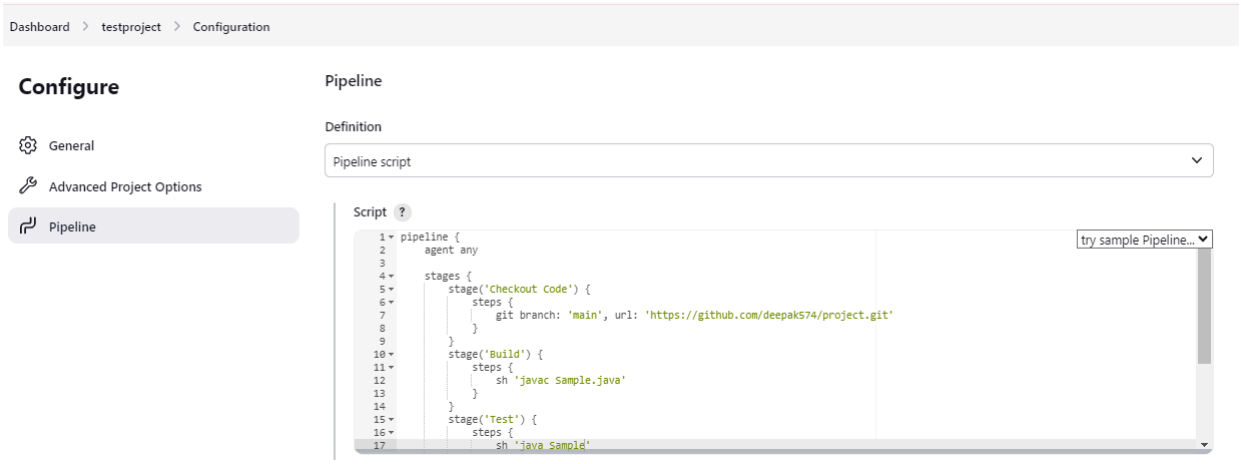

**Give item name as *testproject,* select *Pipeline* and click on *ok* button**



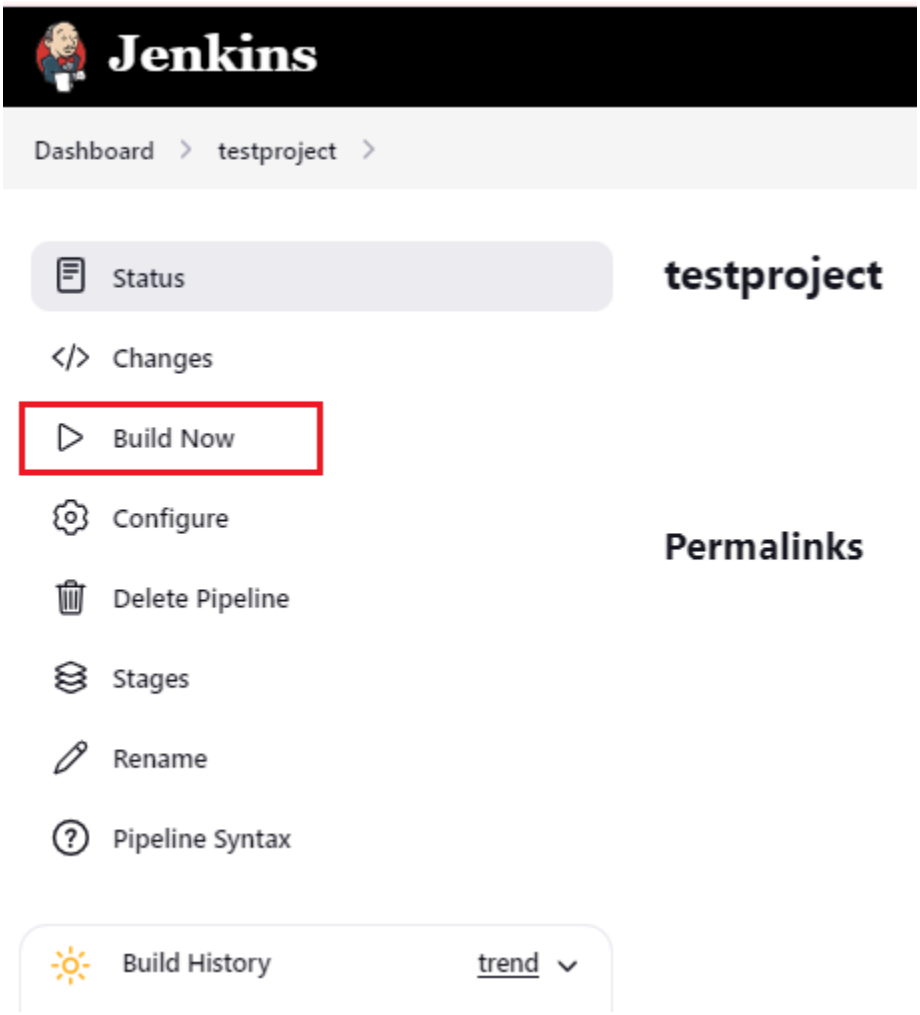
**Click on Pipeline and in the script box type the following code**

```
pipeline {
    agent any

    stages {
        stage('Checkout Code') {
            steps {
                git branch: 'main', url: 'https://github.com/deepak574/project.git'
            }
        }
        stage('Build') {
            steps {

                bat 'javac Sample.java'
            }
        }
        stage('Test') {
            steps {
                        bat 'java Sample'
            }
        }
        stage('Deploy') {
            steps {
                echo 'Deploying application...'
                // Add your deployment commands here (like copying files to a server)
            }
        }
post
{
success
{
mail to:'gujjuladeepak@gmail.com',
subject: "Build success",
Body: 'Your build was successfull'
}
}
    }
}
```

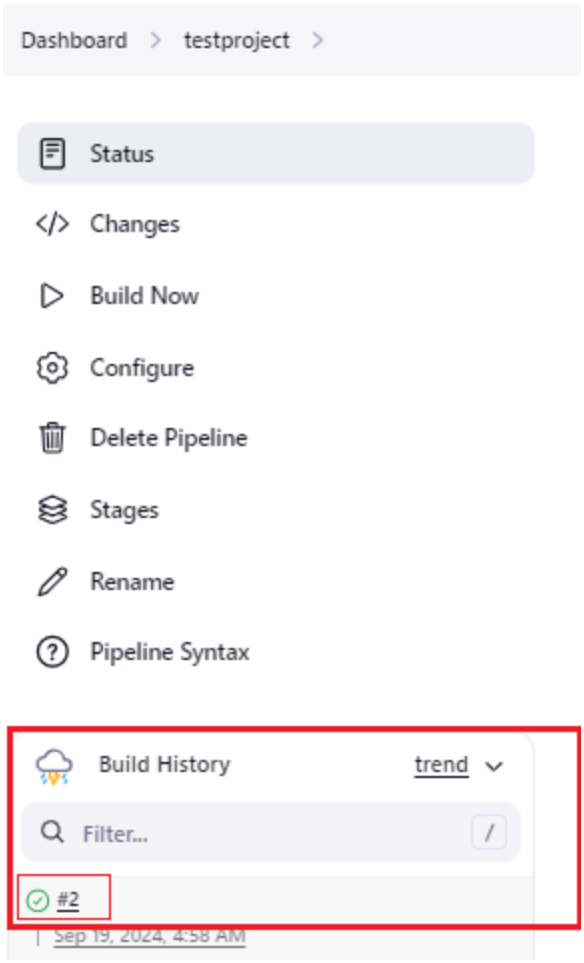
**Note : here url should be ours github url.**



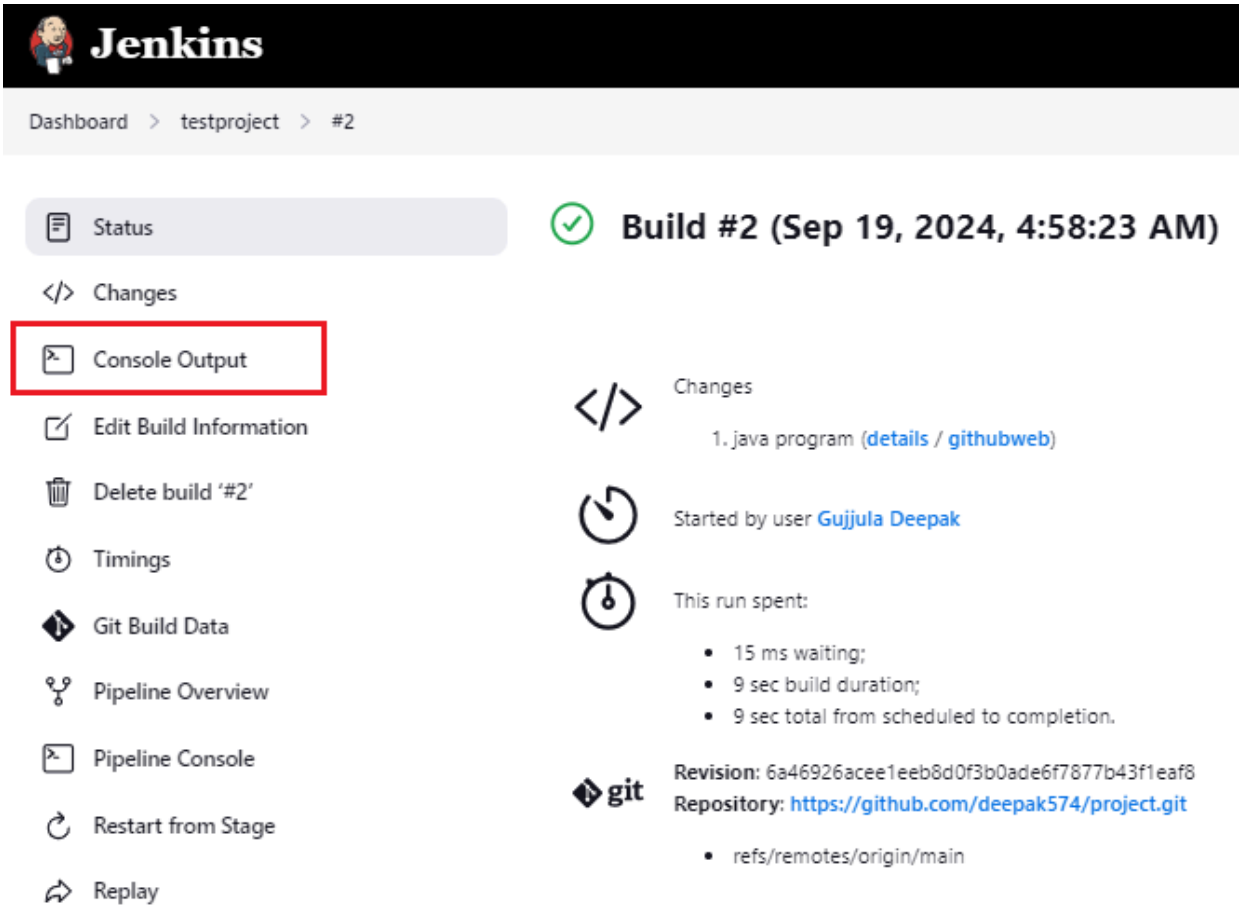

**After typing the code ,click on save button**




**Click on Build now button**




**After clicking on Build Now , the script will be executed and in the Build History we can see it. Now after successful execution click on #2**




**Now click on Console Output**

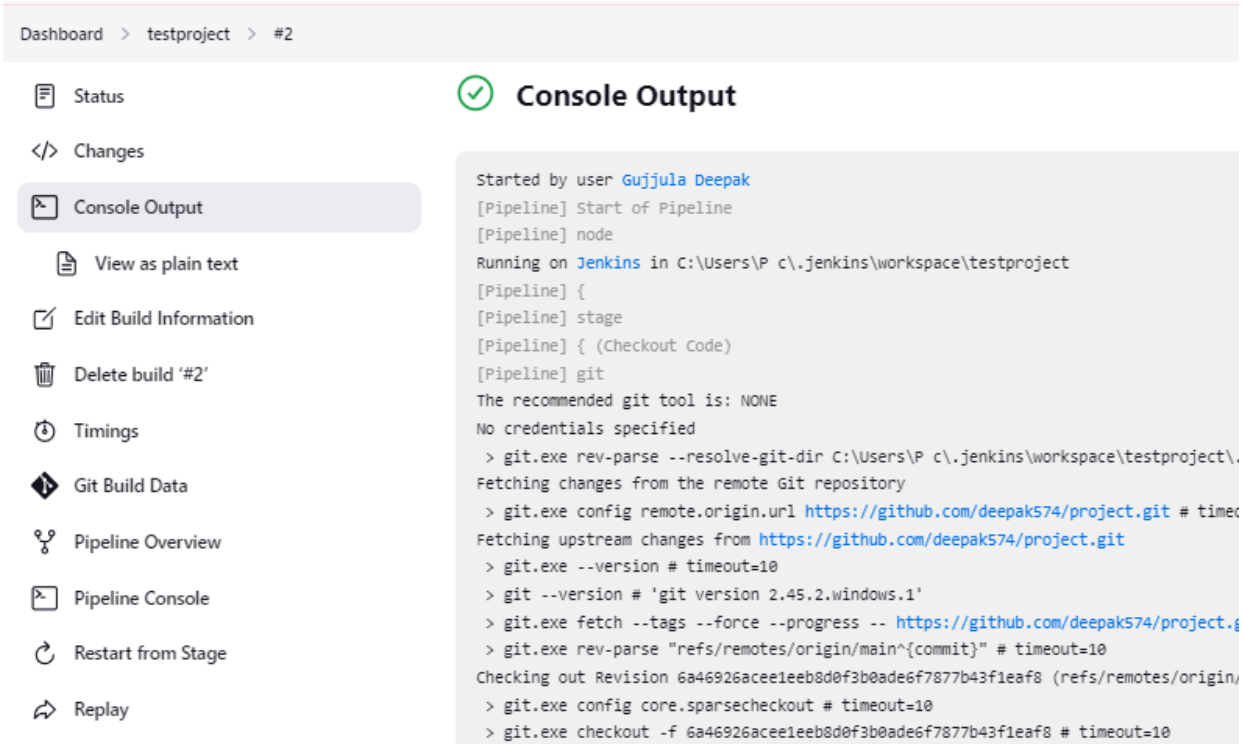
**Output:**

**Started by user Gujjula Deepak**

**[Pipeline] Start of Pipeline**
**[Pipeline] node**
**Running on Jenkins**
**in C:\Users\P c\.jenkins\workspace\testproject**
**[Pipeline] {**
**[Pipeline] stage**
**[Pipeline] { (Checkout Code)**
**[Pipeline] git**
**The recommended git tool is: NONE**
**No credentials specified**

> git.exe rev-parse --resolve-git-dir C:\Users\P c\.jenkins\workspace\testproject\.git # timeout=10
Fetching changes from the remote Git repository
> git.exe config remote.origin.url https://github.com/deepak574/project.git # timeout=10
Fetching upstream changes from https://github.com/deepak574/project.git
> git.exe --version # timeout=10
> git --version # 'git version 2.45.2.windows.1'
> git.exe fetch --tags --force --progress -- https://github.com/deepak574/project.git +refs/heads/*:refs/remotes/origin/* # timeout=10
> git.exe rev-parse "refs/remotes/origin/main^{commit}" # timeout=10
Checking out Revision 6a46926acee1eeb8d0f3b0ade6f7877b43f1eaf8 (refs/remotes/origin/main)
> git.exe config core.sparsecheckout # timeout=10
> git.exe checkout -f 6a46926acee1eeb8d0f3b0ade6f7877b43f1eaf8 # timeout=10
> git.exe branch -a -v --no-abbrev # timeout=10
> git.exe branch -D main # timeout=10
> git.exe checkout -b main 6a46926acee1eeb8d0f3b0ade6f7877b43f1eaf8 # timeout=10
Commit message: "java program"
> git.exe rev-list --no-walk 402d3ea25873de9dad6dd7f4705bcc23ec4cab6c # timeout=10
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Build)
[Pipeline] sh
+ javac Sample.java
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Test)
[Pipeline] sh
+ java Sample
Gujjula Deepak
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Deploy)
[Pipeline] echo
Deploying application...
[Pipeline] }
[Pipeline] // stage
[Pipeline] }

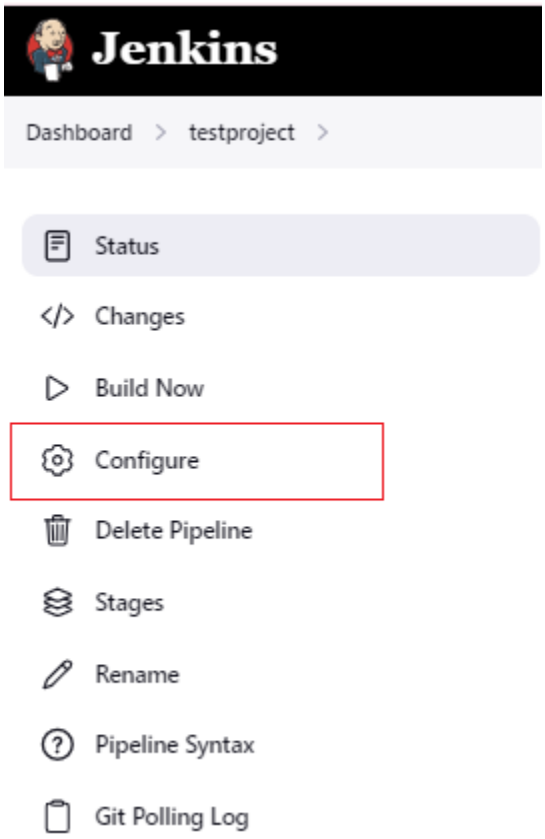
[Pipeline] // node
[Pipeline] End of Pipeline
**Finished: SUCCESS**



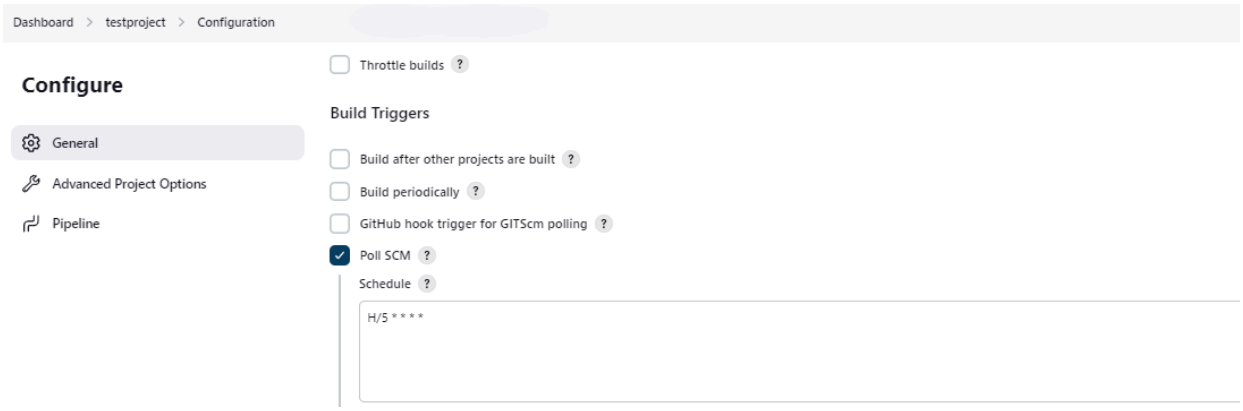

To configure Jenkins to poll your GitHub repository at regular intervals (every 5 minutes, for example), you can use the Poll SCM option.
Jenkins will automatically check GitHub for changes at regular intervals and trigger a build if there are any.

**Steps to Set Poll SCM in Jenkins:**

1. Open your Jenkins job.




2. Click on Configure.
3. Scroll down to the Build Triggers section.
4. Check the box labeled Poll SCM.
5. In the Schedule field, enter H/5 * * * *



6. Click apply & Save.
H/5 means Jenkins will poll every 5 minutes, but the H ensures that the polling happens at different minutes for different jobs, reducing the load on the server.
The other * symbols mean "every hour", "every day", "every month", and "every day of the week", respectively.

**How to check?**
Modify the java program,add to the staging area,commit and push to github.
Wait for 5 min and check jenkins the job will be executed automatically

**6. Explore Docker commands for content management.**

**List Running Containers**

**docker ps:**

Displays all running containers with details like container ID, image name, and status.

**List All Containers (including stopped)**

**docker ps -a:**

Shows both running and stopped containers, useful for reviewing historical activity.

**Start a Container**

**docker start <container_name_or_id>**

Starts an existing container.

**Stop a Container**

**docker stop <container_name_or_id>**

Stops a running container gracefully.

**Remove a Container**

**docker rm <container_name_or_id>**

Removes a stopped container. For removing running containers, first stop them.

**Run a New Container**

**docker run -d -p <host_port>:<container_port> --name <container_name> <image_name>**

Starts a container in detached mode (-d) and maps the host port to the container port.

**List Docker Images**

**docker images**

Displays a list of locally stored Docker images.

**Pull an Image from Docker Hub**

**docker pull <image_name>**

Downloads a specific image from Docker Hub to the local machine.

**Build an Image from a Dockerfile**

**docker build -t <image_name> <path_to_dockerfile>**

Builds a Docker image from a Dockerfile located in the specified directory. The -t flag tags the image with a name.

**Tag an Image**

**docker tag <source_image> <repository_name>:<tag>**

Assigns a tag to an image, helpful for versioning.

**Push an Image to Docker Hub**

**docker push <repository_name>:<tag>**

Pushes an image from your local system to Docker Hub or another repository.

**7. Develop a simple containerized application using Docker.**

```
P c@DESKTOP-TEKQI08 MINGW64 ~ (master)
$ cd e:

P c@DESKTOP-TEKQI08 MINGW64 /e
$ mkdir javadocker

P c@DESKTOP-TEKQI08 MINGW64 /e
$ cd javadocker

P c@DESKTOP-TEKQI08 MINGW64 /e/javadocker
$ git init
Initialized empty Git repository in E:/javadocker/.git/

P c@DESKTOP-TEKQI08 MINGW64 /e/javadocker (master)
$ git config user.email "gujjuladeepak@gmail.com"

P c@DESKTOP-TEKQI08 MINGW64 /e/javadocker (master)
$ git config user.name "Deepak"

P c@DESKTOP-TEKQI08 MINGW64 /e/javadocker (master)
$ vim Sample.java
```

```
public class Sample
{
    public static void main(String args[])
    {
        System.out.println("Gujjula Deepak");
    }
}
```

```
public class Sample
{
        public static void main(String args[])
        {
                System.out.println("Gujjula Deepak");
        }
}
```

```
P c@DESKTOP-TEKQI08 MINGW64 /e/javadocker (master)
$ touch Dockerfile

P c@DESKTOP-TEKQI08 MINGW64 /e/javadocker (master)
$ vim Dockerfile
```

**Dockerfile should contain the following code**

# Use an official OpenJDK runtime as a parent image
FROM openjdk:11-jdk-slim
# Set the working directory inside the container
WORKDIR /app
# Copy the current directory contents into the container at /app
COPY . .
# Compile the Java program
RUN javac Sample.java
# Command to run the program
CMD ["java", "Sample"]

```
# Use an official OpenJDK runtime as a parent image
FROM openjdk:11-jdk-slim

# Set the working directory inside the container
WORKDIR /app

# Copy the current directory contents into the container at /app
COPY . .

# Compile the Java program
RUN javac Sample.java

# Command to run the program
CMD ["java", "Sample"]
```

```
P c@DESKTOP-TEKQI08 MINGW64 /e/javadocker (master)
$ git add .
warning: in the working copy of 'Dockerfile', LF will be replaced by CRLF the next time Git touches it
warning: in the working copy of 'Sample.java', LF will be replaced by CRLF the next time Git touches it

P c@DESKTOP-TEKQI08 MINGW64 /e/javadocker (master)
$ git commit -m "java Docker test"
[master (root-commit) c31db75] java Docker test
 2 files changed, 21 insertions(+)
 create mode 100644 Dockerfile
 create mode 100644 Sample.java
```

```
P c@DESKTOP-TEKQI08 MINGW64 /e/javadocker (master)
$ git remote add origin https://github.com/deepak574/javadockertest.git

P c@DESKTOP-TEKQI08 MINGW64 /e/javadocker (master)
$ git branch -M main

P c@DESKTOP-TEKQI08 MINGW64 /e/javadocker (main)
$ git push -u origin main
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 4 threads
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 569 bytes | 189.00 KiB/s, done.
Total 4 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
To https://github.com/deepak574/javadockertest.git
 * [new branch]      main -> main
branch 'main' set up to track 'origin/main'.
```

Open docker desktop

Open cmd

```
C:\Users\P c>docker --version
Docker version 27.2.0, build 3ab4256
```

docker build -t javadockerapp https://github.com/deepak574/javadockertest.git

```
C:\Users\P c>docker build -t javadockerapp https://github.com/deepak574/javadockertest.git
[+] Building 10.9s (8/8) FINISHED                                                    docker:desktop-linux
 => [internal] load git source https://github.com/deepak574/javadockertest.git                        4.2s
 => [internal] load metadata for docker.io/library/openjdk:11-jdk-slim                                2.5s
 => [auth] library/openjdk:pull token for registry-1.docker.io                                        0.0s
 => [1/4] FROM docker.io/library/openjdk:11-jdk-slim@sha256:868a4f2151d38ba6a09870cec584346a5edc8e9b71fde275eb2e0  0.1s
 => => resolve docker.io/library/openjdk:11-jdk-slim@sha256:868a4f2151d38ba6a09870cec584346a5edc8e9b71fde275eb2e0  0.1s
 => CACHED [2/4] WORKDIR /app                                                                         0.0s
 => [3/4] COPY . .                                                                                    0.1s
 => [4/4] RUN javac Sample.java                                                                       3.1s
 => exporting to image                                                                                0.5s
 => => exporting layers                                                                               0.2s
 => => exporting manifest sha256:d92ceeebe6280e3c3bf967ced854107a45551c115376ddf0b30579bbb8c7d78b     0.0s
 => => exporting config sha256:15a0acb0e4454f2b533184ab28ebbda437867441aaf94d1de1b78a4f65bac217       0.0s
 => => exporting attestation manifest sha256:1e3cc2a1416e22789c266de6aa53663f32c3d68d86f25fd16a83874a0426b595  0.0s
 => => exporting manifest list sha256:97aa58d4f5b453978920de3d5e9bb16e095db27bf009cc2942667ac1aadb2431  0.0s
 => => naming to docker.io/library/javadockerapp:latest                                               0.0s
 => => unpacking to docker.io/library/javadockerapp:latest                                            0.1s

What's next:
    View a summary of image vulnerabilities and recommendations → docker scout quickview
```

docker images

```
C:\Users\P c>docker images
REPOSITORY      TAG       IMAGE ID       CREATED         SIZE
javadockerapp   latest    97aa58d4f5b4   9 seconds ago   673MB
```

Docker run javadockerapp

```
C:\Users\P c>docker run javadockerapp
Gujjula Deepak
```

Open browser and login to docker hub
Push the Docker Image to Docker Hub

**Login to Docker Hub from the Command Line**

First, you need to log in to your Docker Hub account from your terminal.

```
C:\Users\P c>docker login
Authenticating with existing credentials...
Login Succeeded
```

**Tag Your Docker Image for Docker Hub**

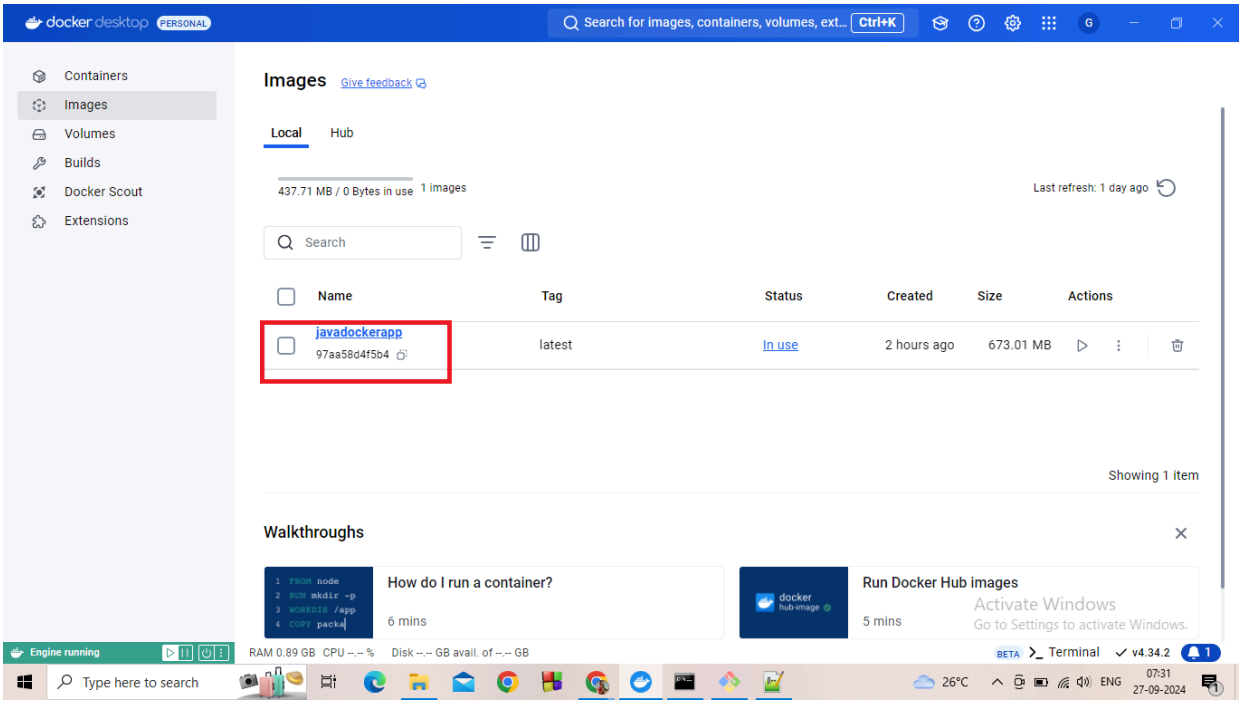
Docker images need to be tagged with your Docker Hub username and the repository name you will use.

```
C:\Users\P c>docker tag javadockerapp gujjuladeepak/javadockerapp:latest
```

**Push the Docker Image to Docker Hub**

After tagging the image, push it to Docker Hub using the following command:

```
C:\Users\P c>docker push gujjuladeepak/javadockerapp:latest
The push refers to repository [docker.io/gujjuladeepak/javadockerapp]
e4dc918eef66: Pushed
69e15dccd787: Pushed
1efc276f4ff9: Pushed
9421bb337be6: Pushed
a2f2f93da482: Pushed
c1a41d8e9e7a: Pushed
9e58bcf4d106: Pushed
12cca292b13c: Pushed
latest: digest: sha256:97aa58d4f5b453978920de3d5e9bb16e095db27bf009cc2942667ac1aadb2431 size: 856
```




## Step 5: Pull the Docker Image from Docker Hub

Now that the image is on Docker Hub, you can pull it from anywhere, such as another server, your local machine, or a cloud environment.

**1. Pull the Docker Image**

Use the docker pull command to pull the image from Docker Hub:

Example

```
docker pull gujjuladeepak/javadockerapp:latest
```

Docker will download the image from Docker Hub to your local machine or server.

**2. Run the Docker Container**

Once the image is pulled, you can run it just like any other Docker container:

```
docker run gujjuladeepak/javadockerapp:latest
```

This will execute the Java program inside the container, and you should see the output:

**8. Integrate Kubernetes and Docker**

kubernetes (often abbreviated as K8s) is an open-source platform designed to automate the deployment, scaling, and management of containerized applications. It is widely used in modern software development to manage applications packaged in containers (e.g., Docker containers).

Key Features of Kubernetes 1. Container Orchestration: Kubernetes manages multiple containers, ensuring they work together seamlessly. 2. Scaling: Automatically increases or decreases the number of application instances based on traffic or demand. 3. Load Balancing: Distributes traffic evenly across application instances to ensure reliability and performance. 4. Self-Healing: Automatically restarts failed containers or replaces them if they crash.

Components

1. Cluster: A Kubernetes cluster is a collection of machines that run your containerized applications.
2. Nodes: A node is a machine (physical or virtual) in the cluster.It runs the containers and is managed by Kubernetes.
3. Master Node: Controls the cluster.
4. Worker Nodes: Run the actual applications in Pods.
5. Pods: The smallest deployable unit in Kubernetes. Each pod contains one or more tightly coupled containers (e.g., a web server and a logging agent).
6. Services: Expose your pods to the outside world or within the cluster (e.g., via NodePort or LoadBalancer).
7. YAML Files: Kubernetes uses YAML (or JSON) files to define applications, their scaling, and network configurations.

Example: File name : javadockerapp-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
name: javadockerapp-deployment
labels:
app: javadockerapp
spec:
replicas: 1
selector:
matchLabels:
app: javadockerapp
template:
metadata:
labels:
app: javadockerapp
spec: containers:
- name: javadockerapp-container
```

```
 image: javadockerapp:latest
 ports:
- containerPort: 8080
```

**Filename: javadockerapp-service.yaml**

```
apiVersion: v1
 kind: Service
 metadata:
name: javadockerapp-service
 spec:
 selector:
app: javadockerapp
ports:
 - protocol: TCP
 port: 8080
 targetPort: 8080
type: NodePort
```

**Execution:**

```
kubectl apply -f javadockerapp-deployment.yaml
kubectl apply -f javadockerapp-service.yaml
```

**9. Automate the process of running containerized application for exercise 7 using Kubernetes.**

To automate the process of running a containerized Java application (Exercise 7: prints "Gujjula Deepak") using Kubernetes, We have to

1. Create a simple Java app.
2. Containerize it with Docker.
3. Push the image to a container registry.
4. Write Kubernetes manifests (Deployment and Service).
5. Apply the manifests to run it on a Kubernetes cluster.

## Create the Java App (Exercise 7)

```
public class HelloWorld {
   public static void main(String[] args) {
      System.out.println("Gujjula Deepak");
   }
}
```

**Dockerfile**

```
FROM openjdk:17
COPY HelloWorld.java /HelloWorld.java
RUN javac HelloWorld.java
CMD ["java", "HelloWorld"]
```

**Step 2: Build and Push Docker Image**

```
# Build the Docker image
docker build -t your-dockerhub-username/hello-world-java .
# Push to Docker Hub (or another registry)
docker push your-dockerhub-username/hello-world-java
```

**Step 3: Kubernetes Deployment Manifest**

Create a file named hello-world-deployment.yaml:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-world-deployment
spec:
  replicas: 1
```

```
  selector:
    matchLabels:
      app: hello-world
  template:
    metadata:
      labels:
        app: hello-world
    spec:
      containers:
      - name: hello-world
        image: your-dockerhub-username/hello-world-java
        imagePullPolicy: Always
```

**Step 4: Apply Manifests to Kubernetes**

```
kubectl apply -f hello-world-deployment.yaml
# (Optional)
kubectl apply -f hello-world-service.yaml
```

**To see Output:**

```
kubectl get pods
kubectl logs hello-world-deployment.yaml
```

**Output :**
**Gujjula Deepak**

## 10. Install and Explore Selenium for automated testing.

### Selenium:

Selenium is an open-source tool used for automating web browsers. It allows testers to simulate user interactions on web applications and is widely used for testing purposes. Selenium supports multiple programming languages like Python, Java, JavaScript, C#, and Ruby.

### Components of Selenium:

**Selenium WebDriver:** A tool for automating browsers by sending commands to interact with web elements.
**Selenium IDE:** A browser extension for recording and playback of test scripts.
**Selenium Grid:** Allows parallel test execution across multiple machines.

### Steps to Install Selenium for Automated Testing:

Install a programming language environment such as Python or Node.js.
Install Selenium using a package manager:
For Python: `pip install selenium`
For JavaScript: `npm install selenium-webdriver`
Download and configure the browser driver (e.g., ChromeDriver, GeckoDriver).
Write a simple script to automate tasks like opening a browser, searching, or clicking elements.

### Features Explored in Selenium:

Browser automation: Opening and interacting with web pages.
Locating elements: Using locators like ID, Name, XPath, and CSS Selectors.
Form handling: Filling input fields and submitting forms.
Capturing screenshots: Saving browser snapshots during tests.
Synchronization: Handling dynamic elements using implicit or explicit waits.

### Advantages of Selenium:

Open-source and platform-independent.
Supports multiple browsers (Chrome, Firefox, Edge, Safari).
Integrates with test frameworks and CI/CD tools.
Enables testing in multiple programming languages.

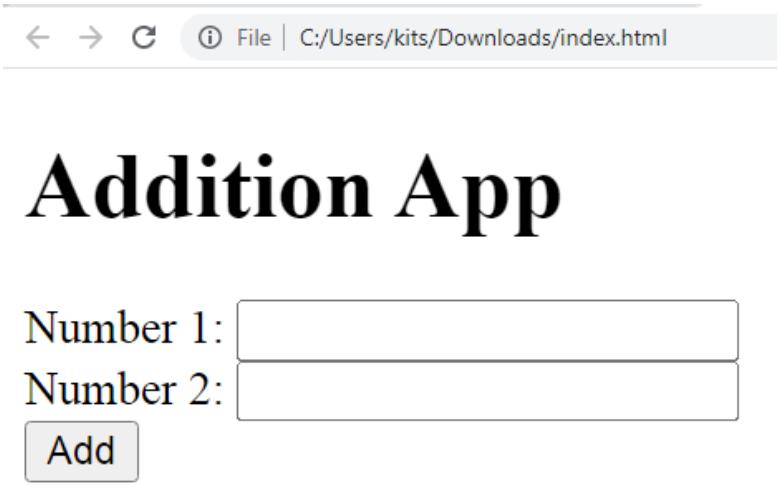
**11. Write a simple program in JavaScript and perform testing using Selenium.**

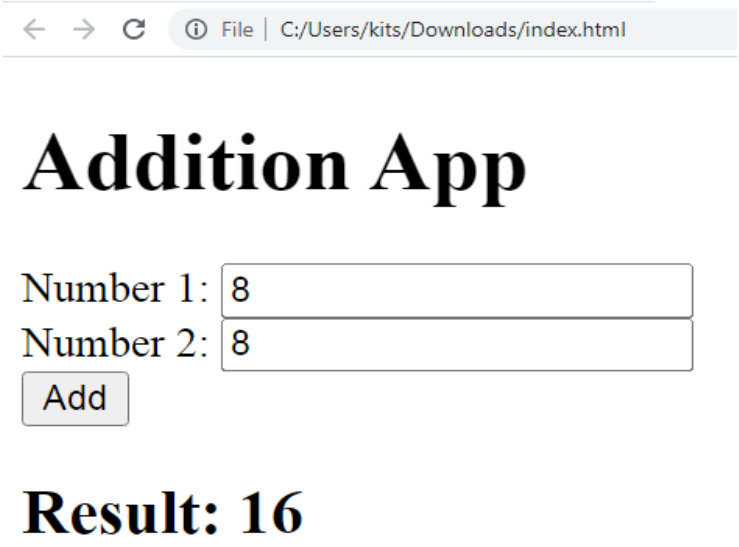
1. Download index.html and test.js
2. Create a folder selenium and save two files in the folder.
3. Open visual studio code
   Open new folder – Selenium.

**index.html**

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <title>Addition App</title>
</head>
<body>
   <h1>Addition App</h1>
   <form id="addForm">
      <label for="num1">Number 1:</label>
      <input type="number" id="num1" required>
      <br>
      <label for="num2">Number 2:</label>
      <input type="number" id="num2" required>
      <br>
      <button type="button" onclick="addNumbers()">Add</button>
   </form>
   <h2 id="result"></h2>

   <script>
      function addNumbers() {
         const num1 = parseInt(document.getElementById('num1').value);
         const num2 = parseInt(document.getElementById('num2').value);
         const result = num1 + num2;
         document.getElementById('result').innerText = "Result: " + result;
      }
   </script>
</body>
</html>
```

**Output:**




**Test.js**

```
// test.js
const { Builder, By, until } = require('selenium-webdriver');
const path = require('path');
require('chromedriver'); // WebDriver Manager will handle ChromeDriver setup automatically.
(async function testAdditionApp() {
    // Initialize the Chrome WebDriver
    let driver = await new Builder().forBrowser('chrome').build();
    try {
        // Open the local HTML file in the browser
        const filePath = path.resolve(__dirname, 'index.html');
```

```
    await driver.get(`file://${filePath}`);
    // Wait a little before entering the first number
    await driver.sleep(1000); // Pause for 1 second
    await driver.findElement(By.id('num1')).sendKeys('5');
    // Wait again before entering the second number
    await driver.sleep(1000); // Pause for 1 second
    await driver.findElement(By.id('num2')).sendKeys('3');
    // Wait before clicking the "Add" button
    await driver.sleep(1000); // Pause for 1 second
    await driver.findElement(By.css('button')).click();
    // Wait for the result to be displayed
    await driver.sleep(2000); // Pause for 1 second
    const resultText = await driver.wait(
      until.elementLocated(By.id('result')),
      1000
    ).getText();
    // Verify the result and log the outcome
    if (resultText === "Result: 8") {
      console.log("Test Passed: 5 + 3 = 8");
    } else {
      console.log("Test Failed");
    }
  } finally {
    await driver.quit(); // Close the browser
  }
})();
```

Open new terminal

Enter commands

```
npm init –y
npm express
npm install selenium webdriver chromedriver
node test.js
```

**12. Develop test cases for the above containerized application using selenium.**

**HelloWorldController.java**

```
package com.example.helloworld;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
@RestController
public class HelloWorldController {
   @GetMapping("/")
   public String hello() {
      return "Hello World";
   }
}
```

**Application.java**

```
package com.example.helloworld;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
@SpringBootApplication
public class Application {
   public static void main(String[] args) {
      SpringApplication.run(Application.class, args);
   }
}
```

**pom.xml**

```
<project xmlns="http://maven.apache.org/POM/4.0.0" ...>
   <modelVersion>4.0.0</modelVersion>
   <groupId>com.example</groupId>
   <artifactId>helloworld</artifactId>
   <version>0.0.1</version>
   <dependencies>
      <dependency>
         <groupId>org.springframework.boot</groupId>
```

```
        <artifactId>spring-boot-starter-web</artifactId>
      </dependency>
    </dependencies>
    <build>
      <plugins>
        <plugin>
          <groupId>org.springframework.boot</groupId>
          <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
      </plugins>
    </build>
</project>
```

**Dockerfile for Containerization**

```
FROM openjdk:17-jdk-alpine
COPY target/helloworld-0.0.1.jar app.jar
ENTRYPOINT ["java", "-jar", "app.jar"]

# Build the app
mvn clean package
# Build the Docker image
docker build -t your-dockerhub-username/hello-world-web .
# Push to registry (if needed)
docker push your-dockerhub-username/hello-world-web
```

**Selenium Test Case (Java + JUnit)**

**HelloWorldTest.java**

```
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
```

```
import org.openqa.selenium.By;
import static org.junit.Assert.assertEquals;
public class HelloWorldTest {
   private WebDriver driver;
   @Before
   public void setUp() {
      // Make sure to download chromedriver and add it to your PATH
      System.setProperty("webdriver.chrome.driver", "/path/to/chromedriver");
      driver = new ChromeDriver();
   }
   @Test
   public void testHelloWorldPage() {
      // Assuming the container is running locally on port 8080
      driver.get("http://localhost:8080");
      String bodyText = driver.findElement(By.tagName("body")).getText();
      assertEquals("Hello World", bodyText);
   }

   @After
   public void tearDown() {
      if (driver != null) {
         driver.quit();
      }
   }
}
```

**Run Your App (Locally or via Kubernetes)**

```
docker run -p 8080:8080 your-dockerhub-username/hello-world-web
```