Chapter 23

Local Raspberry Pi Git Server with External Shared Storage

23.1 Introduction

In today's collaborative environments, managing code and file versions effectively is crucial. However, many developers face challenges when trying to share files and track changes within their local networks. Ever wished you could seamlessly work on a single project, or even a single file, from any computer in your home without the hassle of tracking which machine holds the latest version? Imagine the convenience of accessing shared storage and having a reliable system to manage changes. This chapter provides a comprehensive guide to achieving precisely that (Figure 23.1).

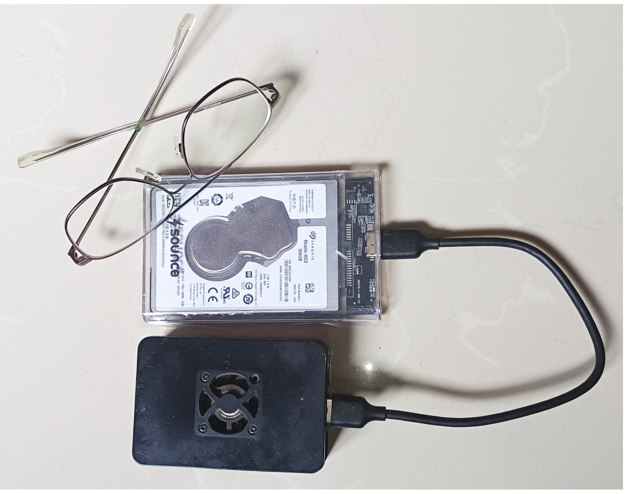


Figure 23.1: Easy-to-Build Raspberry Pi Local Git Server Setup

We'll walk you through setting up a collaborative development environment (configuring the Raspberry Pi, setting up the shared storage, installing and configuring Git, and enabling remote access locally) using a Raspberry Pi as a local Git repository host. This includes configuring an external hard drive for shared storage instead of using Pi's own SD storage, enabling efficient file sharing for your team, and storing the Git repository on this shared drive.

Many collaborative solutions (like GitHub) require your data to reside on the internet, raising concerns about privacy. This chapter addresses those concerns directly. We'll be setting up a *local* Git server, entirely independent of the internet. This means your code and project files remain within your local network, ensuring privacy and security.

Furthermore, we'll demonstrate how to build this system using a low-cost Raspberry Pi and an external hard drive. You'll get 500GB of storage for under INR 7,500, delivering exceptional value compared to the typical INR 20,000+ investment in a NAS. The approach explained in this chapter, offers a powerful and affordable alternative to commercially available Network Attached Storage (NAS) devices, making collaborative development accessible without a significant financial investment. This setup will allow multiple users on your local network to share files, track changes, and collaborate on projects efficiently, all while maintaining control over your data. This chapter will guide you through the process of configuring the Raspberry Pi, setting up the shared storage, installing and configuring Git, and enabling remote access.

If you're looking for a quick setup, you can skip the detailed explanations and focus on the command-line instructions provided in the boxed code examples, along with the brief explanations that accompany them throughout this chapter. These instructions and explanations will guide you through the essential steps to get your local Git server up and running efficiently.

23.2 Requirements

This chapter requires the following hardware and software components:

23.2.1 Hardware Components

- **Raspberry Pi:** Central server. (Components Used: Raspberry Pi 4 2GB; Price: INR5,000)
- **SD Card:** For the Raspberry Pi OS. (Components Used: 16GB SanDisk Class 10; Price: INR400)

- **External HDD:** For shared storage and Git repo. (Components Used: Laptop's internal HDD in external casing; Price: INR1,000 + 350)
  You can use SSD also. Steps are exactly same as HDD.
- **Power Supply:** For the Raspberry Pi. (Components Used: 5.1V 3A DC, Part:KSA-15E-051300-HI USB C; Price: INR700)
- **Wi-Fi Router:** For network connectivity. (Components Used: Syrotech Wi-Fi Router; Price: INR2,000, You can go for cheaper too)
- **Windows 11 PC:** The client machine.
- **Standard SD Card Reader:** For SD card setup (Price: INR70).

23.2.2 Software Tools

- **Raspberry Pi Imager:** For flashing the OS. (Version Used: 1.8.5)
- **Raspbian Lite OS:** Operating System (OS) for the Raspberry Pi.
- **Git:** Version control software.
- **PowerShell (SSH Client):** For remote access.
- **Text Editor:** For configuration files.

23.3 What You Will Achieve

By following the steps in this chapter, you will accomplish the following:

1. **Build** a Fully Functional, Ultra Low-Cost Local Git Server: You will have a complete Git server running on your Raspberry Pi, utilizing an external hard drive for storage, providing a cost-effective alternative to cloud-based solutions or NAS devices.
2. **Gain** Raspberry Pi Configuration Proficiency: You will gain hands-on experience installing and configuring the Raspberry Pi operating system, even if you are a first-time user.
3. **Master** Understanding and Configuring SSH Keys: You will learn about SSH keys and how to configure them for secure, passwordless access to your Raspberry Pi server from the specific machines.

4. **Learn** External Hard Drive Configuration and Mounting: You will learn how to configure an external hard drive for shared storage on the Raspberry Pi and how to mount it automatically, ensuring it's readily available for your Git repository and project files.

5. **Enable** Collaborative Workflow: You will have a shared storage and Git setup, allowing you and your team to collaborate on projects efficiently.

6. **Develop** Troubleshooting Skills: You will gain experience in troubleshooting common configuration issues, preparing you to handle similar challenges in the future.

7. **Enhance** Security: By setting up a local Git server, you'll keep your code private and secure within your local network, avoiding the potential security risks associated with storing your code on external servers.

8. **Realize** Cost Savings: You will have built a powerful collaborative development environment at a fraction of the cost of commercial alternatives.

23.4 Conventions Used in This Chapter

Throughout this chapter, you will encounter certain conventions to help you understand and execute commands effectively:

- **Commands:** Text enclosed in boxes, like this: This is a command, represents commands (for Linux or Windows) that should be executed in the terminal (command prompt or PowerShell for windows).

- **Inline Comments:** Within command examples, you will sometimes see comments at the end of a line, in a new line following a '#' (mostly) or '%' (sometimes). These are inline comments that explain the command or provide additional context. You do not need to type the comment; only the command itself is required. For example, in Linux:

  ```
  sudo umount /mnt/external_hdd # Or wherever it's mounted
  ```

  And in Windows:

  ```
  ipconfig /all % Display all network configuration
  ```

  Only the command ('`sudo umount /mnt/external_hdd`' or '`ipconfig /all`') should be entered in the terminal or command prompt.

23.5 Raspberry Pi Setup and Initial Configuration

Lets not waste more time on explaining what raspberry pi is. Lets directly move to the topic starting with how to set it up. Using Raspberry Pi Imager, one can install the Operating System (OS) very easily and quickly. Click here to download it. You can also install it using terminal (in linux environment).

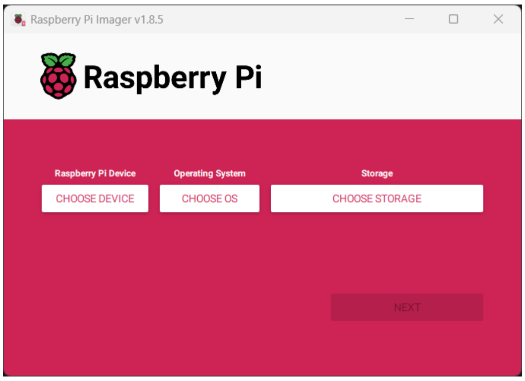
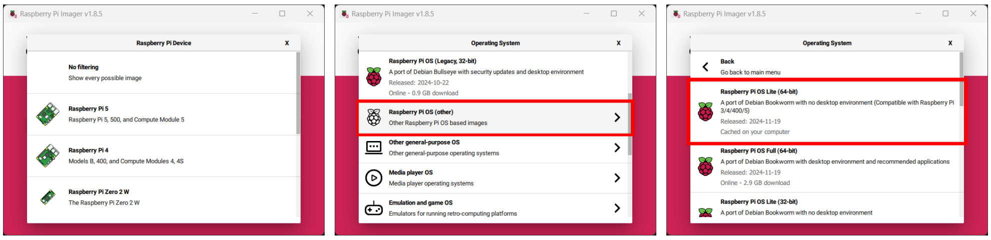


Figure 23.2: Raspberry Pi Imager Interface

After installation, run the Raspberry Pi Imager by searching the program in your PC. You will see the interface as shown in Figure 23.2. The interface is very neat & clean with 4 buttons (among them one remains inactive until all other selections are made first). You have to select device (i.e. your raspberry pi hardware type) then choose operating system & finally the storage (your SD card) before start writing (by clicking the fourth button labelled as NEXT) into that storage. Please note that the interface may vary a little depending on the imager version and the system in which you are using it, but the basic is same.

23.5.1 Advanced Configuration (Optional but highly recommended)

For a headless setup (no monitor required), advanced configuration options are available in the Raspberry Pi Imager. Using Ctrl+Shift+X in the imager allows pre-configuring settings like hostname, SSH, and Wi-Fi before the first boot. This streamlines the initial setup process. You can see it in detail later in Section 23.5.3 along with the picture. It is advisable to follow the Section 23.5.3 first and then you can choose the operationg system and write finally as discussed in Section 23.5.2 as follows.

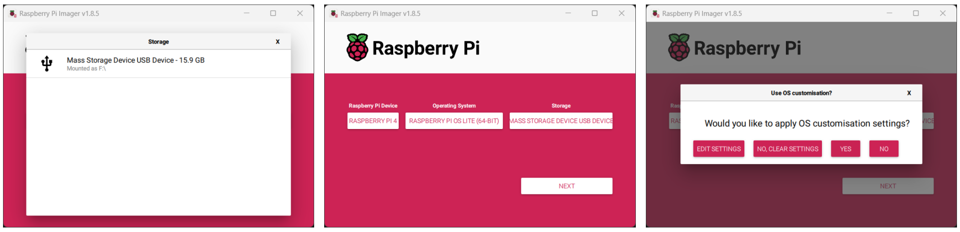

Raspberry Pi Imager v1.8.5
Raspberry Pi Device
X
No filtering
Show every possible image
Raspberry Pi 5
Raspberry Pi 5, 500, and Compute Module 5
Raspberry Pi 4
Models B, 400, and Compute Modules 4, 4S
Raspberry Pi Zero 2 W
The Raspberry Pi Zero 2 W
Raspberry Pi Imager v1.8.5
Operating System
X
Raspberry Pi OS (Legacy, 32-bit)
A port of Debian Bullseye with security updates and desktop environment
Released: 2024-10-22
Online - 0.9 GB download
Raspberry Pi OS (other)
Other Raspberry Pi OS based images
Other general-purpose OS
Other general-purpose operating systems
Media player OS
Media player operating systems
Emulation and game OS
Emulators for running retro-computing platforms
Raspberry Pi Imager v1.8.5
Operating System
X
Back
Go back to main menu
Raspberry Pi OS Lite (64-bit)
A port of Debian Bookworm with no desktop environment (Compatible with Raspberry Pi 3/4/400/5)
Released: 2024-11-19
Cached on your computer
Raspberry Pi OS Full (64-bit)
A port of Debian Bookworm with desktop environment and recommended applications
Released: 2024-11-19
Online - 2.9 GB download
Raspberry Pi OS Lite (32-bit)
A port of Debian Bookworm with no desktop environment

(a) Device selection (b) Select OS (c) Select OS

Figure 23.3: Raspberry Pi Device and OS selection

23.5.2 Operating System Installation

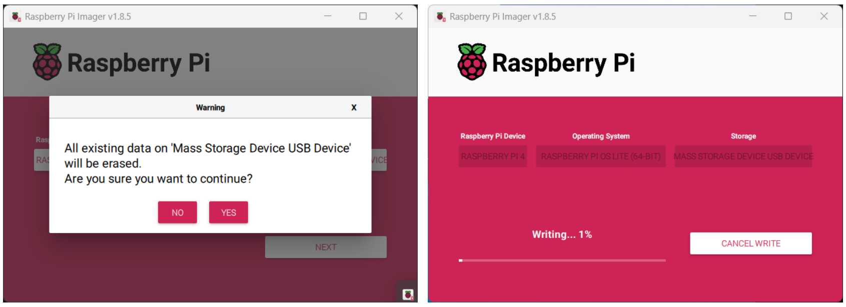
If you have already completed initial configuration mentioned in Section 23.5.3 then we are ready to write the OS otherwise please do that first and come back here. We recommend using Raspbian Lite OS for this setup due to its smaller footprint. Install it onto an SD card using the Raspberry Pi Imager. The steps to be followed are shown in Figure 23.3 to Figure 23.5. Selection of Lite version is shown in Figure 23.3b & 23.3c. After selecting it, select the storage (Figure 23.4a) and click on the button labeled NEXT as you can see in Figure 23.4b. When you do so, it will prompt you a message with options for OS customization settings as in Figure 23.4c. Once you allow it by choosing the option YES followed by accepting the warning as in Figure 23.5a, the writing process will begin as shown in Figure 23.5b. When the process is complete, simply insert the SD card into your Raspberry Pi and you are ready to proceed with the next steps, as described in Section 23.6.


Raspberry Pi Imager v1.8.5
Storage
X
Mass Storage Device USB Device - 15.9 GB
Mounted as F:\
Raspberry Pi Imager v1.8.5
Raspberry Pi
Raspberry Pi Device
Operating System
Storage
RASPBERRY PI 4
RASPBERRY PI OS LITE (64-BIT)
MASS STORAGE DEVICE USB DEVICE
NEXT
Raspberry Pi Imager v1.8.5
Raspberry Pi
Use OS customisation?
X
EDIT SETTINGS
NO, CLEAR SETTINGS
YES
NO
NEXT

Would you like to apply OS customisation settings?

(a) Select Storage (b) Write the OS (c) OS customisation

Figure 23.4: Raspberry Pi storage selection & write



All existing data on 'Mass Storage Device USB Device' will be erased.
Are you sure you want to continue?

Raspberry Pi Device: RASPBERRY PI 4
Operating System: RASPBERRY PI OS LITE (64-BIT)
Storage: MASS STORAGE DEVICE USB DEVICE

Writing... 1%

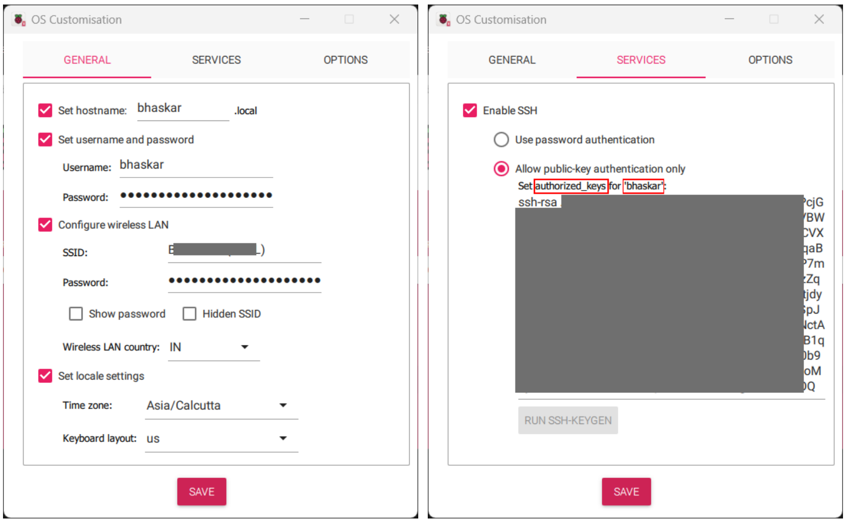
(a) Warning (b) Writing in progress

Figure 23.5: Raspberry Pi OS writing

23.5.3 SSH Access

It is highly recommended that you enable SSH during the OS installation process. While you can enable it later, doing so can be significantly more complex and time-consuming. You may need to perform several tasks, such as verifying SSH server configuration, configuring the '`/etc/ssh/sshd_config`' file manually, and troubleshooting potential issues with other related configuration files. Even after these steps, you might not achieve the desired result if you miss subtle details or other file configurations. Therefore, it is strongly recommended to enable SSH during the initial OS installation using the Raspberry Pi Imager. This will allow you to access the Raspberry Pi remotely from your other machines. If you choose the "Allow public-key authentication only" option during initial configuration and use the "RUN SSH-KEYGEN" feature from the SERVICES tab as shown in Figure 23.6b, the installer will generate an SSH key pair and add the *public* key of the machine (not of the raspberry pi but where you are performing action just now) running the Raspberry Pi Imager software to the `authorized_keys` file (located in the `.ssh` directory within the home directory of the user being created (like *bhaskar* as you see in Figure 23.6a), such as `/home/bhaskar/.ssh/authorized_keys`). This initial `authorized_keys` file, as illustrated in Figure 23.6b, contains the public key of the machine where you first set up the OS. *Please note that the examples you will see in this chapter will be using a Raspberry Pi 4 2GB, and the primary username is "bhaskar" with the hostname "bhaskar.local".*

You can SSH into the Raspberry Pi without a password. However, this is only possible from the specific host machine whose public key is registered in the `authorized_keys` file of the SD card OS image. First, insert the SD card and boot the Pi. From that specific machine mentioned, use a command like `ssh userName@<raspberry pi IP>` (for example, `ssh bhaskar@bhaskar.local`). To enable passwordless SSH access from other



☑ Set hostname: bhaskar .local

☑ Set username and password

Username: bhaskar

Password: ••••••••••••••••••••

☑ Configure wireless LAN

SSID: E[redacted])

Password: ••••••••••••••••••••

□ Show password □ Hidden SSID

Wireless LAN country: IN

☑ Set locale settings

Time zone: Asia/Calcutta

Keyboard layout: us

☑ Enable SSH

○ Use password authentication

● Allow public-key authentication only

Set authorized_keys for 'bhaskar':

ssh-rsa [redacted]

(a) Host name and network (b) SSH config

Figure 23.6: pre-configuring settings

PCs also, you must add (append to be more specific) the *public* SSH key of each of those PCs to the `authorized_keys` file on the Raspberry Pi. This can be done by copying the public key (typically named `id_rsa.pub` or `id_ed25519.pub`) from each client machine to the Raspberry Pi and appending it to the `authorized_keys` file. For an example of copying and adding SSH keys, refer to Section 23.7.4 on "Setting Up SSH Access for the Git User".

Why configure SSH access before the first boot?

Configuring SSH access *before* the first boot simplifies the initial setup process, especially for headless configurations (where you don't have a monitor, keyboard, or mouse connected to the Raspberry Pi). By enabling SSH and setting up key-based authentication during the imaging process, you can immediately access the Raspberry Pi remotely as soon as it boots. This eliminates the need to physically interact with the Raspberry Pi after flashing the SD card, making the setup more streamlined and efficient. It also allows you to configure the rest of the system remotely, which is often more convenient. Without pre-configured SSH access, you would need to find another way to interact with the Raspberry Pi initially (e.g., connecting a monitor and keyboard), which can be cumbersome, especially if you plan to use the Pi in a location where physical access is difficult. Crucially, this method significantly reduces the manual configuration and detailed steps required to enable SSH-based remote access post-boot. Moreover, post-boot configuration is inherently more error-prone, as SSH configuration files (like `sshd_config` and `authorized_keys`) are highly sensitive to syntax; even a minor typo can break SSH, requiring time-consuming troubleshooting.

23.6 External HDD Configuration & Mount

This section details the process of configuring an external Hard disk (HDD) for shared storage on the Raspberry Pi, making it accessible to multiple users.

Once the operating system is written to the SD card followed by inserting it in the Pi and the board is booted up, you can configure the hard drive. This section details the process of configuring an external HDD for shared storage on a Raspberry Pi, making it accessible to multiple users. The following subsections clearly explain the detailed steps to be followed. Please note that all the commands you will see here are Raspberry Pi commands (i.e., after logging into the Pi account, you execute these in that shell).

23.6.1 Identifying the External HDD

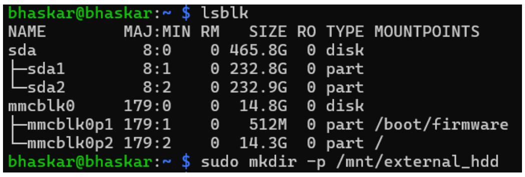
To configure anything, first we need to identify it. For example, if there are many storage devices, first identify the one we want. Lets explore step-wise.

1. First connect the HDD to your Raspberry Pi USB port.

2. Use the `lsblk` command to identify the correct device name (e.g., `/dev/sda1`, used in this chapter). Double-check this! Formatting the wrong device will lead to data loss.

```
lsblk
```

Example output:



```
bhaskar@bhaskar:~ $ lsblk
NAME        MAJ:MIN RM   SIZE RO TYPE MOUNTPOINTS
sda           8:0    0 465.8G  0 disk
├─sda1        8:1    0 232.8G  0 part
└─sda2        8:2    0 232.9G  0 part
mmcblk0     179:0    0  14.8G  0 disk
├─mmcblk0p1 179:1    0   512M  0 part /boot/firmware
└─mmcblk0p2 179:2    0  14.3G  0 part /
bhaskar@bhaskar:~ $ sudo mkdir -p /mnt/external_hdd
```

Figure 23.7: lsblk output

In this example, `/dev/sda` is the HDD, and `/dev/sda1` is the partition (the HDD used here is having 2 partitions) we'll use.

23.6.2 Unmounting the HDD (If Mounted)

Why unmount the HDD?

It's crucial to unmount the HDD before making any changes to its partitions or file system. Unmounting ensures that the operating system is not actively using the drive, preventing data corruption or errors during partitioning or formatting. When a drive is mounted, the OS has open files and is actively reading and writing data. Attempting to modify the drive's structure while it's in use can lead to serious problems, including data loss. Unmounting essentially tells the OS to "release" the drive, making it safe to modify. Think of it like safely ejecting a USB drive from computer before physically removing it. It's a necessary precaution to protect your data.

So, if the HDD is currently mounted (you can see the mount point using *lsblk* as seen in Figure 23.7), unmount it:

```
sudo umount /mnt/mountPoint  # Wherever it's mounted
```

Use `lsblk` again to confirm it's unmounted. Before unmounting, it's recommended (optional) to check if any processes are using the drive:

```
sudo lsof /mnt/mountPoint
sudo fuser -m /mnt/mountPoint
```

23.6.3 Formatting the HDD (Ext4 Recommended)

Unless it is already in ext4, format the HDD partition to ext4 (or your preferred Linux filesystem). **Warning:** This will erase all data on the drive.

Ext4 is the recommended filesystem for the external HDD in this shared storage setup due to its robust handling of permissions, ownership, and overall compatibility with the Linux environment of the Raspberry Pi. *While vfat might seem like a convenient option for cross-platform compatibility, it (vfat) can lead to unexpected issues on Linux, such as files being written to the Raspberry Pi's internal storage instead of the external drive due to subtle mounting or permission conflicts.* Ext4, being a Linux-native filesystem, avoids these pitfalls, ensuring that files are reliably written to the intended external HDD and that access permissions are enforced consistently. Furthermore, ext4 generally offers better performance and reliability on Linux systems compared to vfat, making it a more suitable choice for shared storage. Therefore, formatting the drive as ext4 provides a more stable and secure foundation for shared storage on the Raspberry Pi. **It's highly recommended that you use ext4 for the partition to be mounted.**

```
sudo mkfs.ext4 /dev/sda1
#Replace /dev/sda1 with the actual device name. TRIPLE-CHECK!
```

23.6.4 Creating the Mount Point Directory

Why create a mount point directory?

A mount point is a directory in your file system where you "attach" or "mount" a storage device (like your external HDD). Think of it as a doorway to the files on the HDD. When you mount the HDD to a mount point, the files on the HDD become accessible through that directory. Creating a dedicated directory for this purpose keeps your file system organized and makes it easy to access the files on the HDD. Without a mount point, the operating system wouldn't know where to make the files on the external drive accessible. It's like having a USB drive but not having a way to see its contents on your computer.

So, first create the directory where you'll mount the HDD:

```
sudo mkdir -p /mnt/external_hdd # Or any name you choose
```

While it's common practice to create mount points under the '`/mnt`' directory, it's not strictly required. You can create a mount point anywhere in your file system. However, '`/mnt`' is a traditional location for temporarily mounted file systems, so it's a good convention to follow for consistency and organization. It clearly signals that the directory is used for mounting external storage. If you have a specific reason to use a different location, you can, but '`/mnt`' is generally recommended.

23.6.5 Setting Permissions on the Mount Point Directory (Crucial)

Set the correct ownership and permissions on the mount point directory **before** mounting the drive. **This is the part most likely to cause you problems if not done correctly.** We will use a group called `shared_storage` (or any other name you prefer) to manage access to the mounted drive. If this group doesn't already exist (which it probably won't), you'll need to create it first. To create a group in Linux, we use following command:

```
sudo addgroup shared_storage #Create the shared_storage group
```

Considering you have already created the mount point `/mnt/external_hdd` as mentioned in Section 23.6.4, now, set the ownership and permissions:

```
sudo chown :shared_storage /mnt/external_hdd
# Change group ownership to shared_storage
sudo chmod g+rwX /mnt/external_hdd
# Set group read, write, and execute
sudo chmod g+s /mnt/external_hdd
# Set the setgid bit (for new files)
```

Note: You *do not* need to run `sudo systemctl daemon-reload` after setting permissions with `chown`, `chmod`, or `setfacl`. These commands take effect immediately. `systemctl daemon-reload` is only required after making changes to systemd configuration files (like unit files) or the `/etc/fstab` file. It is not related to file permissions. If this note seems confusing, you may skip this paragraph. Its not that important.

Explanation of each line: (You may avoid this if you want)

1. `sudo chown :shared_storage /mnt/external_hdd`: Changes the group ownership of `/mnt/external_hdd` to the `shared_storage` group. The colon ':' means only the group is changed.

2. `sudo chmod g+rwX /mnt/external_hdd`: Sets permissions for the `shared_storage` group on `/mnt/external_hdd`. `g+rwX` grants read, write, and execute (only if directory or already executable) permissions.

3. `sudo chmod g+s /mnt/external_hdd`: Sets the setgid bit. New files/subdirectories inherit the group ownership of `/mnt/external_hdd` (i.e. `shared_storage`).

23.6.6 Editing `/etc/fstab`

After following the steps in Section 23.6.5, the next task is to configure automounting during boot. This requires configuring the `/etc/fstab` file.

The '`/etc/fstab`' file (File System Table) is a crucial configuration file that tells the system how to mount filesystems at boot time. Without an entry in '`/etc/fstab`' for your external HDD, the drive will not be automatically mounted when the Raspberry Pi starts up. You would have to manually mount it every time, which is not ideal for a shared storage setup. Therefore, we need to edit '`/etc/fstab`' to add an entry for the external HDD so that it is automatically mounted at boot.

Open the `/etc/fstab` file for editing:

```
sudo nano /etc/fstab
```

Add the following line (adjusting for your device and desired options). Be very attentive while editing this file; you can explore the potential consequences of errors in trouble shooting part of Section 23.6.6. Once you finalize editing the `/etc/fstab` file, proceed to Section 23.6.7 to instruct systemd to use the updated configuration.

```
/dev/sda1 /mnt/external_hdd ext4 defaults,noatime 0 2
# Example for ext4
#/dev/sda1 /mnt/external_hdd vfat defaults,noatime 0 2
# Example for vfat(if needed) - Only use ONE of these lines!
```

- `/dev/sda1`: The correct device name(from `lsblk`). **Double-check it!**

- `/mnt/external_hdd`: The mount point directory.
- `ext4`: Filesystem type. **Make sure it matches actual filesystem!**
- `defaults`: Standard mount options.
- `noatime`: Prevents updating access times (improves performance).
- `0 2`: Dump and fsck order. So, in the command, the 0 means your external drive won't be backed up by dump, and the 2 means it will be checked by fsck at boot time after the root partition.

Files have three timestamps: Access Time (when a file is used), Modification Time (when content changes), and Change Time (when permissions/ownership changes). The system normally updates Access Time every time a file is read, which can slow things down. noatime tells the system not to update the Access Time, which speeds up file access (especially reading) and is usually safe. Some programs might need Access Time information, but this is rare. Unless you know you need it, noatime is generally recommended for better performance.

Imagine you have a digital photo album on your external hard drive. Every time you open a photo to view it, the computer records the exact time you opened it (that's the Access Time or atime). If you have thousands of photos and you're browsing through them, your computer is constantly writing these access times to the hard drive, even though you're not changing the photos themselves. This constant writing can slow down your browsing and put unnecessary wear on your hard drive.

noatime is like telling your computer: "Don't bother recording the exact time I view each photo. Just remember when I last changed the photos (Modification Time) or their information (Change Time)." By doing this, your computer doesn't have to write the access time every time you view a photo, making browsing much faster.

So, in the context of your Raspberry Pi Git server, if you have lots of users frequently accessing files in the shared storage, using noatime will prevent the system from constantly updating the access times of those files. This reduces the load on the Raspberry Pi and improves performance.

In simpler terms: Think of noatime as a "don't disturb" sign for the Access Time. It tells the system to leave the Access Time alone, which makes things faster.

Troubleshooting Boot Issues Due to Incorrect `/etc/fstab` Entries (if needed in case of No Pi detected after modifying fstab)

The `/etc/fstab` file (File System Table) is critical for correctly mounting drives at boot. Errors in this file, particularly typos in device names, can prevent the Raspberry Pi from booting properly or connecting to the network. A common problem is mistyping the device identifier for the external HDD (e.g., using `/dev/sdX1` instead of the correct `/dev/sda1`). If the Raspberry Pi attempts to mount a non-existent device, it can hang during startup.

If you encounter such a boot issue after editing `/etc/fstab` and rebooting, the Raspberry Pi might appear to be unresponsive, and you might not be able to find it on your network. In this situation, you'll need to correct the `/etc/fstab` file directly. The easiest way to do this is:

1. Remove the SD card from the Raspberry Pi.

2. Insert the SD card into a card reader and connect it to another Linux machine (say Ubuntu).

3. Within your Linux environment, mount the root partition of this SD card containing the Raspberry Pi OS. (Usually, this is the second partition; use lsblk to identify it). You can also navigate to the corresponding location using a graphical file manager within your Linux environment.

4. Navigate to the `/etc` directory on the mounted partition.

5. With a text editor (e.g., `sudo nano /etc/fstab`) open the `/etc/fstab` file. **Here, be very much cautious that you edit the `/etc/fstab` file of the Raspberry Pi SD card, and not the `/etc/fstab` file of your Linux PC.** Ensure you are working on the correct file by verifying the path. The location of the `/etc/fstab` file when accessing the Raspberry Pi's root partition from another system depends on how you mounted it. `/media/your_username/rootfs/etc/fstab` is generally the common path, but other variations are possible. **Double-check the path before making any changes.**

6. Correct the typo (e.g., change `/dev/sdX1` to `/dev/sda1` or whatever is correct for your case) in the device name.

7. Save the `/etc/fstab` file.

8. Unmount the SD card partitions cleanly.

9. Remove the SD card from the card reader.

10. Reinsert the SD card into the Raspberry Pi and reboot.

After following these steps, the Raspberry Pi should boot correctly, and you should be able to access it on the network. This highlights the importance of carefully reviewing the `/etc/fstab` file for any errors before rebooting.

Sometimes, even after configuring everything right, you may face problems finding the Pi on the network due to HDD searching during boot. A simple power off followed by removal and reattaching the HDD USB, then rebooting, may resolve the issue.

23.6.7 Reload Systemd Daemon and Mount

The `/etc/fstab` file defines how filesystems are mounted at boot. After editing it, `sudo systemctl daemon-reload` instructs systemd (the system manager) to re-read the updated `/etc/fstab` file, ensuring the changes take effect for subsequent mount operations (including at boot). Without this command, systemd uses the old version, and changes are not applied.

Reload the systemd daemon to recognize the `/etc/fstab` changes:

```
sudo systemctl daemon-reload
```

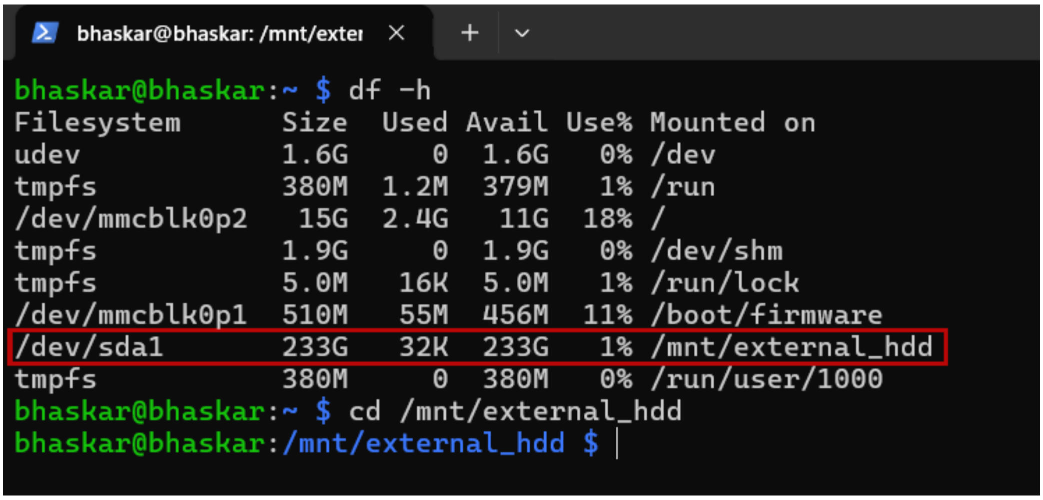
Mounting the Drive (Optional): Once systemd daemon is reloaded and you don't want to restart the Pi immediately, but want the HDD to be mounted now, you can use either of the following commands:

```
sudo mount -a  # Mounts all drives according to /etc/fstab
# or
sudo mount /mnt/external_hdd # Mounts only the specific drive
```

23.6.8 Verify the Mount

Now that boot time mounting is configured, verify that the HDD is mounted at `/mnt/external_hdd` by rebooting the Raspberry Pi and using the `df -h` command. If everything is set up correctly, you will see something like Figure 23.8.

```
df -h
```



```
bhaskar@bhaskar:~ $ df -h
Filesystem      Size  Used Avail Use% Mounted on
udev            1.6G     0  1.6G   0% /dev
tmpfs           380M  1.2M  379M   1% /run
/dev/mmcblk0p2   15G  2.4G   11G  18% /
tmpfs           1.9G     0  1.9G   0% /dev/shm
tmpfs           5.0M   16K  5.0M   1% /run/lock
/dev/mmcblk0p1  510M   55M  456M  11% /boot/firmware
/dev/sda1       233G   32K  233G   1% /mnt/external_hdd
tmpfs           380M     0  380M   0% /run/user/1000
bhaskar@bhaskar:~ $ cd /mnt/external_hdd
bhaskar@bhaskar:/mnt/external_hdd $ 
```

Figure 23.8: df command output

23.6.9 Add Users to the shared_storage Group

You have seen earlier that the shared_storage group was created and configured in Section 23.6.5, setting it as the group that controls access to the HDD's mount point. Now, you need to add the users who require access to the shared storage to this group (or the group you created). The command to add users is as follows:

```
sudo usermod -a -G shared_storage bhaskar
sudo usermod -a -G shared_storage git #(explaned below)
# Add other users as needed
```

Note on the 'git' user: Use that command (the second one of the box) only after creating the user. The 'git' user should be added to the shared storage group. This user is typically created for Git-related operations. For details on creating the 'git' user, please refer to Section 23.7.2. Once you have created the user, come back here and complete adding it to the group.

23.6.10 Final Verification (as a normal user)

As the user bhaskar (or any other user in the shared_storage group), create a file:

```
touch /mnt/external_hdd/finaltestfile
```

Check the file details:

```
ls -l /mnt/external_hdd/finaltestfile
```

Verify that:

- The file is owned by `bhaskar` (or the user who created it).
- The file belongs to the `shared_storage` group.
- The group has read and write permissions (`rw`).

If you see all of this, then the setup is complete and correct. You should no longer need `sudo` to create, modify, or delete files within `/mnt/external_hdd`.

You should check one more thing: that the file is written to the external drive and not the Pi's SD card. For that, unmount the drive:

```
sudo umount /mnt/external_hdd
```

Check if the drive is unmounted:

```
lsblk
```

Then, check the mount point content. It shouldn't show any files there:

```
ls /mnt/external_hdd
```

Even after un-mounting, if you see files, it means they are on the Pi's SD card, and your configuration is incorrect, most likely in the partition formatting (ext4 is suggested) or in the group and mount point directory ownership. Check if formatting was done properly. If you are still facing difficulties, delete the mount point directory:

```
sudo rm -r /mnt/external_hdd #Be very careful with this!
```

Then, strictly follow all the steps from Section 23.6.3 onwards again.

23.7 Git Server Setup

This section guides you through setting up the Git server on your Raspberry Pi. Git is a powerful version control system that tracks changes to your files and allows you to collaborate effectively on projects. It's essential for managing code, documents, and other files where you need to keep track of revisions and work with others. Using a local Git server offers several advantages, including enhanced privacy (your code stays within your network), improved performance (faster access compared to remote servers), and cost savings (no subscription fees for external hosting). It also gives you complete control over your data. Now lets see how to do it very easily.

23.7.1 Installing Git

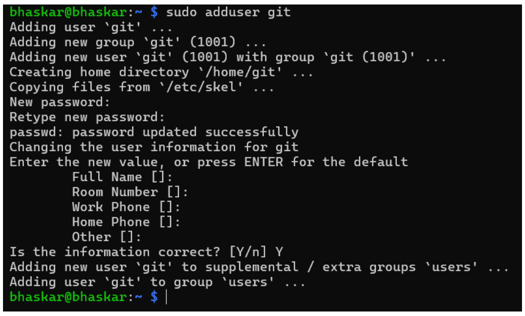
Now that we understand the benefits of a local Git server, let's install Git on the Raspberry Pi:

```
sudo apt update && sudo apt upgrade -y
sudo apt install git
```

23.7.2 Creating the Git User

To create a dedicated user for Git operations, we will create a user named *git*. While the name is the same as the version control tool, it's important to understand that this is simply a Linux user account, not the Git software itself. You could choose any name for this user, but "git" is a common and easily recognizable convention. Create the user as follows (see Figure 23.9. for detail):

```
sudo adduser git
```



```
bhaskar@bhaskar:~ $ sudo adduser git
Adding user `git' ...
Adding new group `git' (1001) ...
Adding new user `git' (1001) with group `git (1001)' ...
Creating home directory `/home/git' ...
Copying files from `/etc/skel' ...
New password:
Retype new password:
passwd: password updated successfully
Changing the user information for git
Enter the new value, or press ENTER for the default
        Full Name []:
        Room Number []:
        Work Phone []:
        Home Phone []:
        Other []:
Is the information correct? [Y/n] Y
Adding new user `git' to supplemental / extra groups `users' ...
Adding user `git' to group `users' ...
bhaskar@bhaskar:~ $ |
```

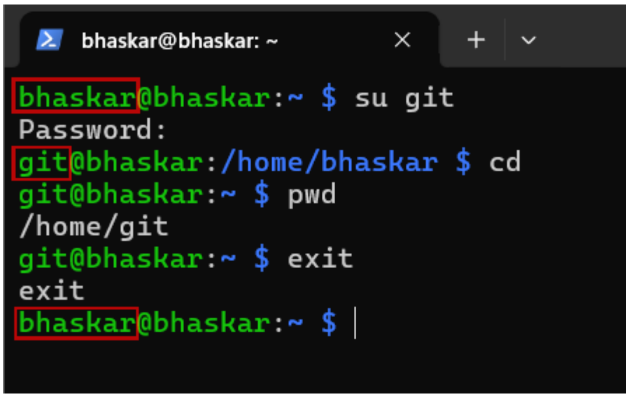
Figure 23.9: Creation of an user named *git*

You can keep all those fields blank as you can see in the picture. You can check if the user with the name *git* is created successfully by using the command below and you will get the result as shown in Figure 23.10 if it is created successfully.

```
su git
```

23.7.3 Setting Up SSH Access for the Default Pi User (`bhaskar`)

SSH access for the default Pi user (in this case, `bhaskar`) is covered in Section 23.5.3. It is highly recommended that you enabled SSH access during the



```
bhaskar@bhaskar:~ $ su git
Password:
git@bhaskar:/home/bhaskar $ cd
git@bhaskar:~ $ pwd
/home/git
git@bhaskar:~ $ exit
exit
bhaskar@bhaskar:~ $ |
```

Figure 23.10: Checking the newly created user named *git*

initial Raspberry Pi OS setup with Raspberry Pi Imager. If you did not, please refer to that section for instructions and visual guidance.

23.7.4 Setting Up SSH Access for the Git User

Configure SSH access for the user `git`. Since the primary Pi user (`bhaskar`) already has SSH access configured (as described in Section 23.5.3), the easiest way to grant SSH access to the `git` user is to leverage the existing `authorized_keys` file.

It's already been mentioned why SSH configuration is important, but to reiterate briefly: When you configure the `authorized_keys` file (i.e., edit to append) with the public keys of specific machines—say, your Windows PC (A), Linux PC (B), and your sibling's PC (C)—then only those machines (A, B, and C) can remotely access the Pi (specifically, the "bhaskar" user). This provides essential security. Furthermore, A, B, and C can access the Pi without needing a password.

Now, to grant the user "git" the same SSH access as the primary user ("bhaskar") has, copy the existing `authorized_keys` file using the primary user ("bhaskar") privileges **and set the appropriate ownership and permissions**. Follow these steps (please note that the following commands with `sudo` are run from the `bhaskar` user's shell, as the `git` user is not in the sudo group).

1. **Create `.ssh` directory:** Create the `.ssh` directory in the `git` user's home directory if it is not there. You can check first using ls.

   ```
   sudo mkdir -p /home/git/.ssh
   ```

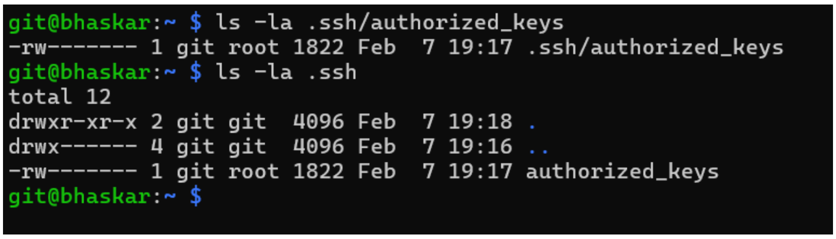
2. **Copy `authorized_keys`:** Copy the `authorized_keys` file from the primary user's `.ssh` directory to the `git` user's `.ssh` directory:

   ```
   sudo cp /home/bhaskar/.ssh/authorized_keys /home/git/.ssh
      /
   ```

   Copying this file issuing command from the user "bhaskar"s shell will copy the file to the desired user and corresponding directory, but its ownership (due to the use of 'sudo') will be root. This needs to be changed to the "git" user. Follow the next section for that.

3. **Set correct ownership and permissions:** Ensure the `git` user owns the `authorized_keys` file and the `.ssh` directory, and set secure permissions (your directory and file permission should look something like Figure 23.11 when you check as the user `git`.). Double-check that you are setting ownership and permissions for the intended user ("git").

   ```
   sudo chown git:root /home/git/.ssh/authorized_keys
   sudo chmod 600 /home/git/.ssh/authorized_keys
   sudo chown git:git /home/git/.ssh
   ```



```
git@bhaskar:~ $ ls -la .ssh/authorized_keys
-rw------- 1 git root 1822 Feb  7 19:17 .ssh/authorized_keys
git@bhaskar:~ $ ls -la .ssh
total 12
drwxr-xr-x 2 git git  4096 Feb  7 19:18 .
drwx------ 4 git git  4096 Feb  7 19:16 ..
-rw------- 1 git root 1822 Feb  7 19:17 authorized_keys
git@bhaskar:~ $
```

Figure 23.11: lsblk output

If you don't want to copy the `authorized_keys` file from "bhaskar" (or the user name you have chosen during OS imaging) to "git" but prefer to create it directly within the "git" user's shell (e.g., using 'touch' or 'nano') to avoid the ownership configurations, you can copy a specific public key (e.g., `id_rsa.pub`) instead of the entire `authorized_keys` file. This is useful if you want the "git" user to have access only from specific machines. However, copying the entire `authorized_keys` file is often simpler if the primary user already has the desired access configured. If you are having trouble logging into the Pi and are unable to append any key to the file, refer to Section 23.8.3 for assistance.

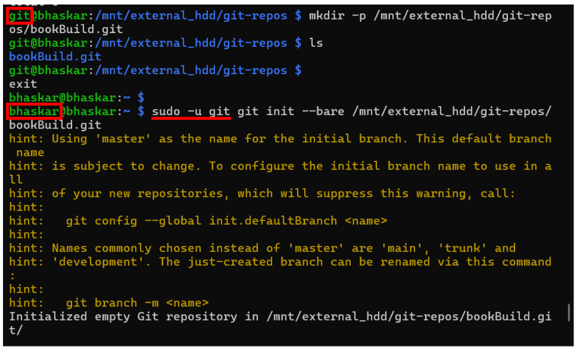
23.7.5 Creating a Bare Git Repository

Create a bare Git repository on the external HDD as the `git` user:

```
sudo -u git mkdir -p /mnt/external_hdd/git-repos/bookBuild.
    git
# Create the repository directory (as git user)
sudo -u git git init --bare /mnt/external_hdd/git-repos/
    bookBuild.git
# Initialize a bare Git repository (as git user)
```

Please see Figure 23.12 for detail.

Please note that it is done with `sudo` and from the primary user's (`bhaskar`) terminal/ shell. You can do the same from the user `git`'s terminal also. Just switch the user with `su git` and you can simply issue commands without the initial `sudo -u git` (which is used to mimic that the command is being issued from the `git` user's shell). You can see Figure 23.12 for a demonstration of commands executed from both the `git` and primary user (`bhaskar`) shells, highlighting the flexibility in how you can create the repository. Both represent the commands are used as the `git` user. Pay attention on the rectangular highlighted areas in the figure to track the active shell.



```
git@bhaskar:/mnt/external_hdd/git-repos $ mkdir -p /mnt/external_hdd/git-rep
os/bookBuild.git
git@bhaskar:/mnt/external_hdd/git-repos $ ls
bookBuild.git
git@bhaskar:/mnt/external_hdd/git-repos $
exit
bhaskar@bhaskar:~ $
bhaskar@bhaskar:~ $ sudo -u git git init --bare /mnt/external_hdd/git-repos/
bookBuild.git
hint: Using 'master' as the name for the initial branch. This default branch
 name
hint: is subject to change. To configure the initial branch name to use in a
ll
hint: of your new repositories, which will suppress this warning, call:
hint:
hint:   git config --global init.defaultBranch <name>
hint:
hint: Names commonly chosen instead of 'master' are 'main', 'trunk' and
hint: 'development'. The just-created branch can be renamed via this command
:
hint:
hint:   git branch -m <name>
Initialized empty Git repository in /mnt/external_hdd/git-repos/bookBuild.gi
t/
```

Figure 23.12: Checking the newly created user named *git*

The purpose and role of a bare Git repository are explained in detail in Section 23.9, where you'll find examples and explanations that will greatly simplify setting up your server and pushing your work from any PC that you have configured for SSH access to the Raspberry Pi (by adding its public key to the `authorized_keys` file). It will be much easier than you might think! You'll see for yourself when you reach Section 23.9.

23.8 Access Git Repo from Client Machine

23.8.1 Setting up SSH Key for Client (Windows 11)

Generate an SSH key pair on your Windows 11 machine if you don't have one already. The process for providing SSH access to other machines during Raspbian Lite OS image installation using SD card and Raspberry Pi Imager software is provided briefly in Section 23.5.3. It is already explained in detail earlier in Section 23.7.4 on how to grant access for more clients. However, it is revised along with the key generation process in the following section.

23.8.2 Generating SSH Key

SSH keys are used for secure, passwordless authentication. You need to generate an SSH key pair (a private key and a public key) on your client machine (e.g., your Windows 11 PC) before you can use SSH to connect to the Raspberry Pi.

Open PowerShell or Git Bash on your Windows 11 machine. You can use either of the following commands:

```
ssh-keygen -t ed25519
```

OR

```
ssh-keygen
```

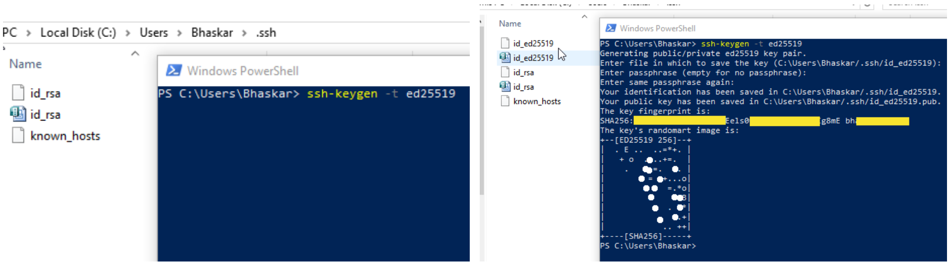
The `ssh-keygen -t ed25519` command generates a more modern and secure Ed25519 key pair. The `ssh-keygen` command (without the '-t' option) generates an RSA key pair, which is also widely used. Ed25519 is generally preferred for its improved security.

When prompted, you can press Enter to accept the default file location (`C:/Users/UserName/.ssh/id_rsa` or `C:/Users/UserName/.ssh/id_ed25519`) and leave the passphrase empty (recommended for ease of use).

The command will generate two files in the '.ssh' directory: a private key file (e.g., 'id_rsa' or 'id_ed25519') and a public key file (e.g., 'id_rsa.pub' or 'id_ed25519.pub'). This process is demonstrated in Figure 23.13, where you can see the directory content before and after key generation, as described in subcaptions 23.13a and 23.13b.

As shown in Figure 23.14a, you can simply press enter to skip the file name selection for the ssh key.

Copy the contents of the **public key** file (the one ending in '.pub') and append it to the 'authorized_keys' file on your Raspberry Pi (as described in Section 23.8.3). Figure 23.14b shows how to check the file properties to make sure which one is the public key file. Now you are ready to use SSH.

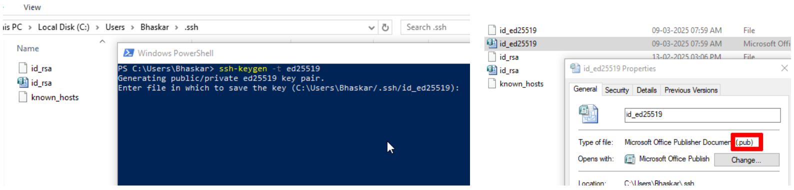


```
PS C:\Users\Bhaskar> ssh-keygen -t ed25519
```

```
PS C:\Users\Bhaskar> ssh-keygen -t ed25519
Generating public/private ed25519 key pair.
Enter file in which to save the key (C:\Users\Bhaskar/.ssh/id_ed25519):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in C:\Users\Bhaskar/.ssh/id_ed25519.
Your public key has been saved in C:\Users\Bhaskar/.ssh/id_ed25519.pub.
The key fingerprint is:
SHA256:[redacted]Eels0[redacted]g8mE bh[redacted]
The key's randomart image is:
+--[ED25519 256]--+
|  . E ..   ..=*+. |
|   + o   . ..+=.  |
|    .      =.   . |
|        =   +...o|
|            =.*o|
|               B|
|            .  *|
|              .+|
|            .. ++|
+----[SHA256]-----+
PS C:\Users\Bhaskar>
```

(a) Before generating Ed25519 key (b) Before generating Ed25519 key

Figure 23.13: SSH key generation (Windows Power shell) & directory content



```
PS C:\Users\Bhaskar> ssh-keygen -t ed25519
Generating public/private ed25519 key pair.
Enter file in which to save the key (C:\Users\Bhaskar/.ssh/id_ed25519):
```

id_ed25519 09-03-2025 07:59 AM File
id_ed25519 09-03-2025 07:59 AM Microsoft Offi
id_rsa 13-02-2025 03:06 PM File
id_rsa
known_hosts

id_ed25519 Properties

id_ed25519

Type of file: Microsoft Office Publisher Documen (.pub)

Opens with: Microsoft Office Publish

Location: C:\Users\Bhaskar\.ssh

(a) Skipped Key Filename (b) File Properties

Figure 23.14: Intermediate Key Steps and Public Key View

23.8.3 Adding SSH Key to Raspberry Pi

Copy the *public* key (e.g., id_rsa.pub or the one that you have generated) from your Windows machine to the Raspberry Pi. For this, scp is a robust tool. In windows power shell you can use the following to copy the file from your machine to the Pi:

```
scp C:\Users\bhask\.ssh\id_rsa.pub bhaskar@<raspberry_pi_ip
   >:~/.ssh/
%Copy to home directory of the primary user (bhaskar)
%You could also use \texttt{ssh-copy-id} command but that
%is often unreliable from Windows. so scp is better to use.
```

Then, on the Raspberry Pi (as the 'bhaskar' user):

```
cat ~/.ssh/id_rsa.pub >> /home/git/.ssh/authorized_keys
#Please note >> which will append but not replace existing
#content of authorized_keys
sudo chown git:git /home/git/.ssh/authorized_keys
sudo chmod 600 /home/git/.ssh/authorized_keys
```

If you have already updated the ownership & permission of the file by following the procedures in Section 23.7.4, you may skip the last two commands in the code block above.

Now, you might be wondering how to copy the public key to the Raspberry Pi if you don't have direct access to its file system from your Windows machine. The answer is that you should have already configured SSH access to the Raspberry Pi, as described in Section 23.5.3. The figures in that section shows a key being generated and configured on the SD card from the beginning. This is the PC where you ran the Raspberry Pi Imager. That PC, therefore, has SSH access to the Pi.

If you've forgotten which PC was used for the initial setup, don't worry. Simply remove the SD card from the Raspberry Pi, insert it into a card reader, and plug the reader into a Linux PC. Navigate to the "pi" user's home directory on the SD card, go to the '.ssh' directory, and edit the 'authorized_keys' file from your Linux PC. The process for editing files on the Raspberry Pi's SD card from a Linux machine is described step-by-step in Section 23.6.6. Refer to that section for detailed instructions on mounting the SD card's root partition and editing the necessary file.

23.8.4 Cloning the Repository

Clone the repository to your local machine:

```
git clone git@<raspberry_pi_ip>:/mnt/external_hdd/git-repos/
    bookBuild.git
```

This bare repository was created in Section 23.7.5. The above command clones the repository into your machine. Please refer to Section 23.9 for more detail. It provides a hands-on walkthrough, covering everything from repository creation (including the directory and bare repository) to configuring remotes, pulling, committing, pushing, and other essential operations.

23.8.5 Configuring Git User Details

Set global Git user email & name (to be performed in client end, not in Pi):

```
git config --global user.email "your.email@example.com"
git config --global user.name "Your Name"
```

Why is this necessary?

Git requires a user name and email address to be associated with each commit. This information is used to identify the author of changes and is included in the commit history. If these details are not configured, Git will prompt you with an error message when you attempt to make a commit, preventing the commit from being recorded.

Checking if Configuration is Already Set:

You can check if your global Git user details are already configured by running the following commands:

```
git config --global user.email
git config --global user.name
```

If the commands return your email address and name, respectively, then the configuration is already set. If they return nothing, you will need to set them using the commands provided earlier.

Important Note: This configuration is to be performed in Git Bash, a terminal, PowerShell, or the command prompt on the client machines that will be collaborating with the Git repository.

23.8.6 Add files in staging area and Commit Changes

Add and commit your changes:

```
git add .
git commit -m "Initial commit with README"
```

23.8.7 Setting Up the Remote (Origin)

Understanding Git Remotes:

In Git, a “remote” is a reference to a repository stored on a server. In our case, it’s the bare Git repository we created on the Raspberry Pi. Adding a remote allows your local Git repository to communicate with the remote repository, enabling you to push and pull changes. The ”origin” is a common name for the default remote repository.

Add the remote URL:

```
git remote add origin git@<raspberry_pi_ip>:/mnt/external_hdd
   /git-repos/bookBuild.git
```

Replace <raspberry_pi_ip>with the actual IP address or hostname of your Raspberry Pi. This command adds a remote named `origin` that points to the bare Git repository on your Raspberry Pi.

23.8.8 Pushing Changes

To push the changes to the remote repository, use `git push`. In some cases, if the repository is not cloned but created locally (a scenario you will see later in Section 23.9.2) and you are pushing for the first time, you need to use `-u` for the *initial* push of a branch:

```
git push -u origin master
```

Since the repository in this section was cloned (in Section 23.8.4), the `-u` option is not needed. A simple `git push` will work. Subsequent pushes, regardless of whether the repository was initially cloned or created locally, can be done with `git push` only.

It is always advisable to clone the bare repository and work from the clone to avoid the complexities of manually configuring upstream branches and other remote settings.

23.9 End-to-End Git Workflow Example

We will now put the previously discussed concepts into practice by creating a repository, accessing it from a client, and demonstrating modifications. This section provides a practical, step-by-step example demonstrating the complete workflow of setting up a local Git repository, connecting it to the remote repository on the Raspberry Pi server, and collaborating with multiple users. This example assumes you have already configured SSH access and set up the external HDD on the Raspberry Pi, as described in the previous sections. Let's start!

23.9.1 Setting Up the Bare Repository on the Raspberry Pi (Server)

The following steps are **performed on the Raspberry Pi server**.

1. **Create Repository Directory:** Create the directory where the bare Git repository will reside. We use the `-u git` option to execute the command as the `git` user, even though it's run from the "bhaskar" user's shell:

   ```
   sudo -u git mkdir -p /mnt/external_hdd/git-repos/testRepo
      .git
   # Please note that it is being done in your Raspberry Pi
   #server, but not in the client windows machine.
   ```

   This command is executed from the "bhaskar" user's shell, but the '-u git' option ensures that `/mnt/external_hdd/git-repos/testRepo.git` directory on the external HDD is created with the "git" user as the owner. The `-p` option creates parent directories as needed.

2. **Initialize Bare Repository:** Initialize a bare Git repository within the newly created directory, again as the `git` user:

```
sudo -u git git init --bare /mnt/external_hdd/git-repos/
   testRepo.git # Being done in your Raspberry Pi server
```

Explanation: This command creates a *bare* Git repository—a repository stripped down to its essentials. Think of it as the control center for your project's history: it stores all the commits, branches, and tags, but no actual files. It's "bare" because it lacks a working directory where you'd normally edit files. When someone does the initial push with the -u option (this -u option is not for run as user, but is an option of git command as in `git push -u`), the commits are transferred, the corresponding branch ref is created/updated in the bare repo, and, most importantly, upstream tracking is established between the local and remote branches for simplified future synchronization. ***If these terms seem confusing, don't worry. You can still achieve your goal without fully understanding every detail. Just follow the remaining steps, and you'll see how easily you can get it working.*** Think of it like riding a bike: you don't need to be an engine specialist to enjoy the ride. Similarly, you don't need to understand every minute detail of Git to use it effectively. If you can successfully complete the workflow, that's what truly matters.

23.9.2 Creating a Local Git Repository and Connecting to the Remote

These steps are performed on your primary client machine (e.g., your Windows 11 PC), from where you will be collaborating.

1. **Create Project Directory:** Create a directory for your project (e.g., `testRepo`). It is created in power shell as you can see in Figure 23.15a. You can also use Git Bash (recommended for consistent command availability) for that. You can open Git Bash in the current directory by right-clicking while holding Shift in File Explorer and selecting "Git Bash Here". After creating the directory, you may create a `README.md` file:

```
mkdir testRepo #creating the directory :on windows system
cd testRepo #Changing the working directory
touch README.md #creating an empty file
```

You can see the directory content as shown in Figure 23.15b.

2. **Initialize Git Repository:** Initialize a Git repository in the same project directory (Figure 23.15c):

   ```
   git init
   ```

   While this creates a new local repository, you could alternatively have cloned the bare repository from the Pi (as shown earlier). Cloning automatically sets up the remote connection, saving you the steps of manually adding the remote and configuring tracking information.

3. **Add and Commit:** Add the `README.md` file into the staging area and make an initial commit. If you encounter an error about user details, configure them globally (*skip those two intermediate lines if you have already done it*):

   ```
   git add .
   git config --global user.email "your.email@example.com"
   git config --global user.name "Your Name"
   git commit -m "Added a README file to the repository"
   ```

4. **Add Remote (Origin):** Add the remote URL pointing to the bare repository that has been created in Section 23.9.1 on the Raspberry Pi:

   ```
   git remote add origin git@<raspberry_pi_ip>:/mnt/
       external_hdd/git-repos/testRepo.git
   ```

   If you get the error like below, then add Git's `bin`, `cmd` and `mingw64/bin` directories to your system's PATH environment variable. You can check the remote status also as shown in Figure 23.16

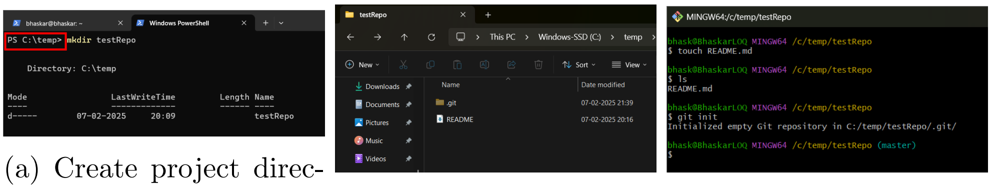
   ```
   bash: $'\302\203git': command not found
   ```

5. **Push to Remote:** Push the local repository to the remote repository. Use the `-u` option for the initial push. Subsequent pushes are done with simple `git push`. You can see Figure 23.16 for detail.

   ```
   git push -u origin master
   ```

   Here, "master" is the branch name. If you are unsure of the branch name, you can use the command `git branch` to list the available local branches.

Note: If you had cloned the bare repository instead of creating a local one with `git init`, you would have avoided steps 1, 2, and 4. Also, you could have used `git push` directly from the initial push without the `-u` option.



```
PS C:\temp> mkdir testRepo


    Directory: C:\temp


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----        07-02-2025     20:09                testRepo
```

(a) Create project directory

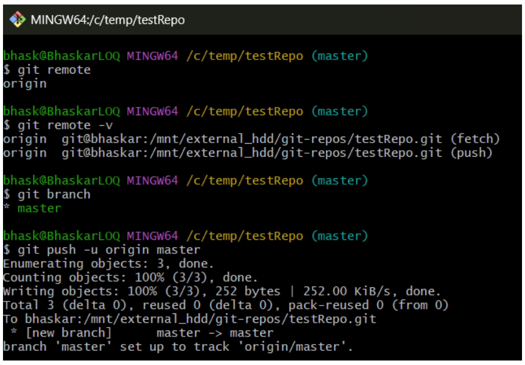
Name	Date modified
.git	07-02-2025 21:39
README	07-02-2025 20:16

(b) Directory content

```
bhask@BhaskarLOQ MINGW64 /c/temp/testRepo
$ touch README.md

bhask@BhaskarLOQ MINGW64 /c/temp/testRepo
$ ls
README.md

bhask@BhaskarLOQ MINGW64 /c/temp/testRepo
$ git init
Initialized empty Git repository in C:/temp/testRepo/.git/

bhask@BhaskarLOQ MINGW64 /c/temp/testRepo (master)
$
```

(c) Repository initiation

Figure 23.15: Project directory creation & local repo init (in windows client)



```
bhask@BhaskarLOQ MINGW64 /c/temp/testRepo (master)
$ git remote
origin

bhask@BhaskarLOQ MINGW64 /c/temp/testRepo (master)
$ git remote -v
origin  git@bhaskar:/mnt/external_hdd/git-repos/testRepo.git (fetch)
origin  git@bhaskar:/mnt/external_hdd/git-repos/testRepo.git (push)

bhask@BhaskarLOQ MINGW64 /c/temp/testRepo (master)
$ git branch
* master

bhask@BhaskarLOQ MINGW64 /c/temp/testRepo (master)
$ git push -u origin master
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Writing objects: 100% (3/3), 252 bytes | 252.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
To bhaskar:/mnt/external_hdd/git-repos/testRepo.git
 * [new branch]      master -> master
branch 'master' set up to track 'origin/master'.
```

Figure 23.16: Creating remote and pushing repository (in windows client)

23.9.3 Cloning and Collaborating from Another Client Machine

These steps are performed on a secondary client PC (not the one that you just have worked on in Section 23.9.2).

Cloning a Git repository creates a complete copy of the repository (including its history) on your local machine. This is essential for collaborating with others on a project or working on the project from multiple locations. When you clone a repository, you gain access to all the files, branches, and commit history, allowing you to contribute to the project, make changes, and synchronize your work with the remote repository. Setting up your own local Git server on a Raspberry Pi makes collaboration very easy. This allows you to make changes, commit them locally, and then push them to your central repository on the Pi.

1. **Clone Repository:** Clone the repository from the Raspberry Pi server:

```
git clone git@<raspberry_pi_ip>:/mnt/external_hdd/git-
    repos/testRepo.git
```

2. **Make Changes, Commit, and Push:** Make changes to the cloned repository, add the changes into the staging area and commit them,

and then push the changes. **You do not need the -u option for subsequent pushes from a cloned repository**:

```
git add .
git commit -m "Added file from second machine"
git push origin master
#Or simply do'git push' if the branch is already'master'.
```

At this point, the Pi-based Git server is updated with the recent changes made by this second client due to the push operation performed. The server now has the latest, modified version of the work.

3. **Update Local Repository (on First PC):** To get the latest changes made by the second PC, use `git pull` on your first PC (the one you have seen in Section 23.9.2). You can check if your local repository is behind the remote repository by using `git fetch` followed by `git status`, and then update your working directory by using `git pull`.

```
git pull
```

   The `git fetch` command downloads the latest changes from the remote repository but does not automatically integrate them into your local branch. `git status` then shows you if your local branch is behind. `git pull` combines `git fetch` and `git merge` in a single command, downloading the changes and merging them into your current branch.

23.9.4 Collaborative Workflow

On other machines, clone the repository and make changes. Pushing changes from a cloned repository does not require the `-u` option. Use `git fetch` to see remote changes and `git pull` to merge them into your local branch.

23.10 Achieving Local Git Mastery: Final Summary

This chapter has provided a comprehensive guide to setting up a local Git server on a Raspberry Pi with external shared storage, enabling seamless collaboration within your local network. You've learned how to configure the Raspberry Pi, set up an external hard drive for shared storage, install and configure Git, and enable remote access from client machines. This setup offers a secure, private, and cost-effective solution for collaborative development.

By following the steps in this chapter, you've gained the ability to:

- Share files efficiently among multiple users on your local network.
- Collaborate on projects using a powerful version control system (Git).
- Maintain privacy and security by keeping your code and data within your local network.
- Set up a robust and affordable alternative to commercial NAS devices.
- Access your Git repository from any machine securely on your network.
- Establish a complete Git workflow, from repository creation to collaboration, using a practical, hands-on example.

The end-to-end workflow demonstrated in Section 23.9 showcased how to create a bare Git repository on the Raspberry Pi, initialize a local repository on a client machine, connect to the remote repository, and collaborate with multiple users. This practical example, covering repository creation, remote configuration, and essential Git operations like pull, commit, and push, provides a solid foundation for managing your projects effectively.

This setup empowers you to take control of your development environment and collaborate efficiently without relying on external services or compromising your data privacy. The skills and knowledge gained in this chapter can be applied to a wide range of projects and scenarios, enabling you to manage your files and collaborate effectively, whether you're working on code, documents, or other types of projects.

Furthermore, by utilizing a Raspberry Pi and an external hard drive, you've created a budget-friendly alternative to expensive Network Attached Storage (NAS) solutions, making collaborative development accessible to a wider audience. The ability to manage your own local Git server not only provides a secure and efficient way to collaborate but also offers a valuable learning experience for those seeking to deepen their understanding of Git and server management.

Resources and Further Reading

This chapter is crafted with the help of numerous valuable resources. Here's a list of the resources and tools that were instrumental in its creation:

Websites

Raspberry Pi Official Website:
Official web link
Essential for Raspberry Pi OS downloads, documentation, and general information.
GitHub SSH Documentation:
Link to the page
A comprehensive guide to SSH key generation and management, particularly helpful for the SSH access sections.

YouTube Videos

Video by Todd Spatafore
Explains the process of converting a Raspberry Pi into a local Git server using Raspberry Pi Imager, highlighting the installation of necessary software, user setup, and repository creation.
Video by SpaceRex
Provides a comprehensive guide on setting up a personal Git server using a Raspberry Pi.

YouTube Playlist

Git Learning Playlist:
Playlist by CodeWithHarry
A valuable resource for learning Git concepts and commands.

AI Tools

Gemini Chatbot: Used for brainstorming, content refinement, and generating explanations.
ChatGPT: Utilized for content generation, editing, and providing alternative perspectives.
Pieces for Developers: Employed for code snippet management, explanation, and presentation.

Book

Unix: Concepts and Applications by Sumitabha Das: Provided fundamental knowledge of Unix/Linux commands and concepts, crucial for the Raspberry Pi and Git server setup.

These resources provided the foundation for the information presented in this chapter, and they are recommended for readers who wish to further explore the topics covered.

If you have any questions, feedback, or suggestions, please feel free to reach me at: explorewithbhaskar@gmail.com