SSH, SFTP & RSYNC — COMPLETE COURSE

Remote Access with SSH

9 chapters covering SSH fundamentals, key-based authentication, SFTP file transfers, rsync syncing, port forwarding, and automated backups.

osztromok.com — Learning Series

Chapter 1 — How SSH Works

SSH — Secure Shell — is the standard way to connect to a remote machine over a network. It replaced older, insecure protocols like Telnet and rsh by encrypting everything: your commands, the server's responses, and any files transferred. Understanding how SSH works under the hood makes every subsequent chapter easier — and stops you making the mistakes that get people locked out of their own servers.

> **What SSH gives you:** an encrypted terminal session on a remote machine, running as if you were sitting in front of it. Everything after that — SFTP, rsync, port forwarding, tunnelling — is built on top of this same connection.

The Problem SSH Solves

Before SSH, protocols like Telnet sent everything in plain text. Anyone on the same network — a colleague on the same office switch, anyone between you and the server on the internet, or someone with access to router logs — could read your username, password, and every command you typed. SSH eliminates this by encrypting the entire session.

It solves three distinct problems:

- **Confidentiality** — everything is encrypted; an eavesdropper sees only scrambled bytes.
- **Integrity** — each message is signed; tampering with it in transit causes the connection to fail.
- **Authentication** — both sides verify who they're talking to; the server proves it's the right machine, and you prove you're an authorised user.

The SSH Handshake — What Happens When You Connect

When you run `ssh user@hostname`, a precise sequence of events takes place before you see a prompt. The whole thing happens in under a second.

YOUR MACHINE (CLIENT)	REMOTE MACHINE (SERVER)
1. TCP connection	
	2. Send host public key

Opens a TCP connection to port 22 on the server.

↓

3. Check host key
Compares server's public key against `~/.ssh/known_hosts`.

↓

5. Agree on session key
Uses Diffie-Hellman to derive a shared secret — without ever sending it over the wire.

↓

7. Authenticate
Proves identity with a password or private key.

↓

9. Encrypted session
All traffic from here is encrypted with the shared session key.

⇄

Sends its public key so the client can verify it is the right server.

↓

4. Negotiate algorithms
Both sides agree on the encryption, MAC, and key-exchange algorithms to use.

↓

6. Session key established
Both sides now have the same shared secret without it having crossed the network.

↓

8. Grant or deny access
Checks credentials against `~/.ssh/authorized_keys` or the system password database.

↓

10. Shell / command
Starts the shell or runs the requested command inside the encrypted channel.

> **Diffie-Hellman key exchange** is the mathematical trick that lets two parties agree on a shared secret over a public channel without ever sending that secret. Even if someone records every byte of your SSH handshake, they cannot derive the session key — it's never transmitted, only calculated independently on each side.

Public Key Cryptography — the Heart of SSH

SSH uses **asymmetric cryptography** in two places: verifying the server's identity, and (optionally, but preferably) authenticating the user. The concept is always the same: a mathematically linked key pair where anything encrypted with one key can only be decrypted with the other.

PUBLIC KEY

`~/.ssh/id_ed25519.pub`

Safe to share with anyone. Goes on the server in `~/.ssh/authorized_keys`. Like a padlock — anyone can lock with it, only you can unlock.

PRIVATE KEY

`~/.ssh/id_ed25519`

Never leaves your machine. Never share it. If someone has your private key, they are you as far as any server is concerned. Protect it with a passphrase.

When you connect with a key pair, the server challenges your client to prove it holds the private key matching the public key in `authorized_keys`. The private key signs a piece of data; the server verifies the signature with your public key. Your private key is never sent — the server just confirms the maths works out.

Key Algorithms — Which to Use

Algorithm	Key file names	Status	Notes
Ed25519	id_ed25519 / id_ed25519.pub	Use this	Modern elliptic curve. Fast, small keys, very strong. Default recommendation.
ECDSA	id_ecdsa / id_ecdsa.pub	Acceptable	Older elliptic curve. Fine but Ed25519 is preferable on all modern systems.
RSA (4096-bit)	id_rsa / id_rsa.pub	Still common	Widely supported including older servers. Use 4096-bit if RSA is required. 2048-bit is marginal.
DSA	id_dsa / id_dsa.pub	Avoid	Fixed 1024-bit key size. Disabled in modern OpenSSH. Do not generate or use.

Host Verification — the known_hosts File

Server authentication is the part most people ignore — but it's what protects you from connecting to the wrong machine. The first time you connect to a host, SSH presents the server's public key fingerprint and asks you to verify it:

philip@debian — first connection to a new host

```
philip@debian:~$ ssh philip@192.168.1.100 The authenticity of host '192.168.1.100 (192.168.1.100)'
can't be established. ED25519 key fingerprint is SHA256:abc123XYZdef456GHI789jklMNO012pqrSTU345vwx.
This key is not known by any other names. Are you sure you want to continue connecting
(yes/no/[fingerprint])? yes Warning: Permanently added '192.168.1.100' to the list of known hosts.
philip@server:~$
```

After typing `yes`, SSH saves the server's public key to `~/.ssh/known_hosts` on your machine. Every subsequent connection checks this file — if the server presents a different key, SSH refuses the connection and raises an alarm.

What known_hosts looks like

```
# ~/.ssh/known_hosts — one line per known host
# Format: hostname/IP    key-type    public-key-data

192.168.1.100 ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAI...
192.168.1.1   ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAI...
osztromok.com ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAI...
```

When the host key changes — a WARNING to take seriously

philip@debian — host key mismatch warning

```
philip@debian:~$ ssh philip@192.168.1.100
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ @ WARNING: REMOTE HOST IDENTIFICATION
HAS CHANGED! @ @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ IT IS POSSIBLE THAT
SOMEONE IS DOING SOMETHING NASTY! Someone could be eavesdropping on you right now (man-in-the-middle
attack)! It is also possible that a host key has just been changed. The fingerprint for the ED25519
key sent by the remote host is SHA256:XYZ... Please contact your system administrator. Add correct
host key in /home/philip/.ssh/known_hosts to get rid of this message. Offending ED25519 key in
/home/philip/.ssh/known_hosts:3 Host key verification failed.
```

Do not blindly dismiss this warning. It appears for legitimate reasons — you reinstalled the OS on the server, the server's IP was reused for a different machine, or you updated SSH keys intentionally. But it also appears if someone is intercepting your connection. Verify out-of-band (phone/Slack the server owner, check the console) before overriding it.

Removing a stale known_hosts entry

If you know the key changed for a legitimate reason (you reinstalled the server, for example), remove the old entry with `ssh-keygen -R`:

philip@debian — remove stale known_hosts entry

```
philip@debian:~$ ssh-keygen -R 192.168.1.100 # Host 192.168.1.100 found: line 3
/home/philip/.ssh/known_hosts updated. Original contents retained as
/home/philip/.ssh/known_hosts.old # Then reconnect — you'll get the first-time fingerprint prompt
again philip@debian:~$ ssh philip@192.168.1.100
```

Verifying a Fingerprint Before Connecting

The right way to confirm a server's fingerprint on first connection is to check it *on the server itself*, through a separate channel (physical access, the hosting provider's console, or a colleague who has access). Run this on the server:

philip@server — get the server's own fingerprint

```
philip@server:~$ ssh-keygen -lf /etc/ssh/ssh_host_ed25519_key.pub 256
SHA256:abc123XYZdef456GHI789jklMNO012pqrSTU345vwx root@server (ED25519)
```

Compare this fingerprint character-by-character with what SSH showed you on the client side. If they match, type `yes`. If they don't, stop and investigate.

The SSH Directory — File Roles at a Glance

```
# On your client machine (e.g. philip@debian or philip@windows-pc)
~/.ssh/
├── known_hosts         Keys of every server you've connected to (do not edit by hand)
├── id_ed25519          Your private key — NEVER share or copy to a server
├── id_ed25519.pub      Your public key — copy this to servers
├── config              SSH client config — shortcuts, per-host settings (Chapter 4)
└── authorized_keys     On SERVERS: list of public keys allowed to log in

# On the server (e.g. philip@server:/etc/ssh/)
/etc/ssh/
├── sshd_config             SSH server configuration
├── ssh_host_ed25519_key     Server's private host key (never leave this machine)
└── ssh_host_ed25519_key.pub  Server's public host key (sent to clients on connect)
```

SSH vs Older Protocols

Protocol	Encrypted	Still used	Verdict
SSH	**Yes — everything**	Universally	**The correct choice for all remote access**
Telnet	No — plain text	Legacy embedded systems only	Never use on any network you don't fully control
rsh / rlogin	No — plain text	Essentially extinct	Replaced by SSH in the 1990s
Mosh	Yes (UDP-based)	Niche	SSH wrapper that handles roaming and intermittent connections — useful on mobile networks

Key Takeaways

- **SSH encrypts everything** — credentials, commands, output, and file transfers all travel as ciphertext.
- **The handshake happens in two stages** — server authentication first (is this the right machine?), then user authentication (are you allowed in?).
- **Ed25519 is the key algorithm to use** in 2026 — faster and stronger than RSA, supported by all modern SSH implementations.
- **known_hosts is your protection against man-in-the-middle attacks** — SSH checks it automatically on every connection.
- **A changed host key warning demands investigation** — verify out-of-band before removing the old entry and reconnecting.
- **Your private key never leaves your machine** — if asked to copy it to a server, something has gone wrong.

Next — Chapter 2: Connecting to a Remote Host. With the theory in place, Chapter 2 covers the practical side: the `ssh` command in full, useful flags, connecting with a specific user or port, running a single command without opening a shell, and what to do when the connection is refused.

CHAPTER 2 — CONNECTING TO A REMOTE HOST

Chapter 2 — Connecting to a Remote Host

Chapter 1 explained the theory — encryption, key pairs, host verification. This chapter is where you actually type things. The `ssh` command is surprisingly flexible: it can open an interactive shell, run a single command and exit, forward a port, or proxy through an intermediate host. This chapter covers the core syntax, the most useful flags, running remote commands without opening a full shell, and diagnosing the most common connection errors.

The Basic Command

The minimum you need is a username and a hostname or IP address:

```
ssh    -p  2222    philip@192.168.1.100
```

`ssh`	The SSH client command
`-p 2222`	Optional — connect on port 2222 instead of the default port 22
`philip`	The username to log in as on the remote machine
`@`	Separator between user and host
`192.168.1.100`	The hostname or IP address of the remote machine

> **Username shortcut:** if your local username matches your remote username, you can omit the `user@` part entirely and just type `ssh hostname`. SSH uses your current username by default.

Your First Connection — Step by Step

philip@debian — connecting to a LAN server

```
philip@debian:~$ ssh philip@192.168.1.100 The authenticity of host '192.168.1.100 (192.168.1.100)'
can't be established. ED25519 key fingerprint is SHA256:abc123XYZdef456GHI... Are you sure you want
to continue connecting (yes/no/[fingerprint])? yes Warning: Permanently added '192.168.1.100' to the
list of known hosts. philip@192.168.1.100's password: (type password — nothing appears on screen)
Welcome to Debian GNU/Linux 12 (bookworm) Last login: Tue Jun 17 20:14:32 2026 from 192.168.1.50
philip@server:~$ ← you are now on the remote machine
```

Notice the prompt changed — `philip@debian` became `philip@server`. Every command you type now runs on the remote machine. To return to your local machine, type `exit` or press `Ctrl+D`.

philip@server — ending the session

```
philip@server:~$ exit logout Connection to 192.168.1.100 closed. philip@debian:~$ ← back on your local machine
```

Connecting by Hostname vs IP Address

You can use either an IP address or a hostname. On a local network, your router usually provides hostname resolution so you can use the machine's name directly:

philip@debian — hostname vs IP

```
# These may connect to the same machine: philip@debian:~$ ssh philip@192.168.1.100 ← IP address philip@debian:~$ ssh philip@raspberrypi ← hostname (if DNS resolves it) philip@debian:~$ ssh philip@raspberrypi.local ← mDNS hostname (Bonjour/Avahi) # For public servers, use the domain name: philip@debian:~$ ssh philip@osztromok.com
```

> **.local hostnames** (like `raspberrypi.local`) use mDNS (multicast DNS), which works on local networks without a DNS server. On Linux this requires `avahi-daemon` to be running. On Windows, Bonjour (installed with iTunes or Apple devices) provides the same feature.

Essential SSH Flags

Flag	What it does	Example
`-p PORT`	Connect on a non-default port (default is 22)	`ssh -p 2222 philip@server`
`-i FILE`	Use a specific private key file instead of the default	`ssh -i ~/.ssh/id_work philip@server`
`-l USER`	Specify the login username (alternative to user@host)	`ssh -l philip server`
`-v`	Verbose — show the connection and auth process. Use -vv or -vvv for more detail. Invaluable for debugging.	`ssh -v philip@server`
`-A`	Forward your SSH agent — lets you use your local keys on the remote machine to connect onwards to a third machine	`ssh -A philip@jumphost`
`-X`	Enable X11 forwarding — run graphical apps on the remote machine, display locally	`ssh -X philip@server`
`-N`	Do not execute a remote command — used with port forwarding (Chapter 5) when you only want the tunnel	`ssh -N -L 8080:localhost:80 philip@server`

Flag	What it does	Example
`-f`	Go to background after connecting — used with -N for background tunnels	`ssh -fN -L 8080:localhost:80 philip@server`
`-q`	Quiet mode — suppress warnings and diagnostic messages. Useful in scripts.	`ssh -q philip@server uptime`
`-o OPTION=VALUE`	Set any config option inline without editing the config file	`ssh -o StrictHostKeyChecking=no philip@server`

Running a Single Remote Command

You don't have to open an interactive shell every time. Append a command after the host and SSH will run it, print the output, and exit — no prompt, no interaction required. This is very useful in scripts.

philip@debian — single remote commands

```
# Check disk space on the server without opening a shell philip@debian:~$ ssh philip@server df -h
Filesystem Size Used Avail Use% Mounted on /dev/sda1 50G 12G 36G 25% / /dev/sdb1 2.0T 800G 1.1T 43%
/data # Check system uptime philip@debian:~$ ssh philip@server uptime 20:14:32 up 14 days, 3:22, 1
user, load average: 0.01, 0.03, 0.00 # Restart a service on the remote server philip@debian:~$ ssh
philip@server sudo systemctl restart apache2 (password prompt appears if using password auth, or
silently succeeds with key auth) # Run a multi-word command — quote it or use -- to be safe
philip@debian:~$ ssh philip@server "df -h | grep sda" /dev/sda1 50G 12G 36G 25% / # Run a script
that exists on the remote machine philip@debian:~$ ssh philip@server "bash /home/philip/backup.sh"
```

> **Quote your commands** when they contain pipes, redirections, or spaces — otherwise your local shell interprets them before SSH sees them. `ssh server "df -h | grep sda"` sends the whole string to the remote shell. Without quotes, the pipe runs locally and SSH never sees it.

Running a Local Script on a Remote Machine

You can pipe a local script into SSH and have the remote machine execute it — without copying the file first:

philip@debian — pipe a local script to a remote shell

```
philip@debian:~$ ssh philip@server 'bash -s' < local_script.sh
```

Connecting on a Non-Standard Port

Many administrators move SSH away from port 22 to reduce automated scanning noise. If the server listens on a different port, use `-p`:

philip@debian — non-standard port

```
philip@debian:~$ ssh -p 2222 philip@server # Or connect to a VPS that uses a custom port
philip@debian:~$ ssh -p 4422 philip@myserver.example.com
```

> **Security note:** moving SSH off port 22 reduces automated brute-force scanning (bots constantly probe port 22) but is not a security measure on its own — a port scan reveals the new port within seconds. Real security comes from key-based authentication and fail2ban, covered in Chapters 3 and the Web Server Security course. That said, fewer log noise entries can be genuinely useful.

Connecting from Windows

Windows 10 and 11 include OpenSSH as an optional feature — the same `ssh` command works in PowerShell and Command Prompt. If it's not installed, enable it from Settings → Optional Features.

PowerShell on Windows — SSH works the same way

```
PS C:\Users\Philip> ssh philip@192.168.1.100 philip@192.168.1.100's password: Welcome to Debian
GNU/Linux 12 (bookworm) philip@server:~$
```

GUI SSH clients are also available if you prefer a point-and-click interface:

- **PuTTY** — the classic Windows SSH client. Free, lightweight, saves session profiles. Still widely used in enterprise environments.
- **Kitty** — a PuTTY fork with extra features (tabs, automatic reconnect, URL highlighting). Philip uses this.
- **MobaXterm** — SSH client + X11 server + file browser in one. Popular for Linux admin from Windows.
- **Windows Terminal** — Microsoft's modern terminal. Use the built-in `ssh` command inside it for a clean experience.

Debugging with Verbose Mode

When a connection fails silently or behaves unexpectedly, `-v` reveals exactly what SSH is doing at each step. Add more `v`s for progressively more detail:

philip@debian — verbose connection (truncated)

```
philip@debian:~$ ssh -v philip@server OpenSSH_9.2p1 Debian-2+deb12u5, OpenSSL 3.0.15 debug1: Reading configuration data /home/philip/.ssh/config debug1: Connecting to server [192.168.1.100] port 22. debug1: Connection established. debug1: Server host key: ssh-ed25519 SHA256:abc123... debug1: Host '192.168.1.100' is known and matches the ED25519 host key. debug1: Offering public key: /home/philip/.ssh/id_ed25519 ED25519 SHA256:xyz789 debug1: Server accepts key: /home/philip/.ssh/id_ed25519 debug1: Authentication succeeded (publickey). philip@server:~$
```

The key lines to look for: which key file SSH is trying, whether the server accepts it, and which authentication method ultimately succeeded.

Troubleshooting Common Errors

`ssh: connect to host server port 22: Connection refused`

Cause: The SSH service isn't running on the server, the server is listening on a different port, or a firewall is blocking port 22.

Fix: Check the SSH service: `sudo systemctl status sshd`. Check the port in `/etc/ssh/sshd_config`. Check UFW: `sudo ufw status`. If using a custom port, add `-p PORT` to your ssh command.

`ssh: connect to host server port 22: No route to host`

Cause: The hostname or IP address doesn't resolve, the server is off, or there's no network path to it.

Fix: Verify the IP: `ping 192.168.1.100`. Confirm the server is on. Confirm you're on the right network. Check DNS: `nslookup hostname`.

`Permission denied (publickey,password)`

Cause: Wrong username, wrong password, or (with key auth) the key is not in the server's authorized_keys.

Fix: Confirm the username matches an account on the server. Try `ssh -v` to see which keys are being offered. If using keys, verify the public key is in `~/.ssh/authorized_keys` on the server with permissions 600.

`WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!`

Cause: The server's host key doesn't match the one stored in your known_hosts. Could be a reinstalled OS, a reused IP, or — take seriously — a man-in-the-middle attack.

Fix: Verify out-of-band that the server's key legitimately changed. If confirmed, remove the old entry: `ssh-keygen -R hostname`. Reconnect and accept the new fingerprint.

`ssh_exchange_identification: read: Connection reset by peer`

Cause: The SSH daemon accepted the TCP connection but then dropped it. Often caused by `hosts.deny`, `AllowUsers` / `DenyUsers` in sshd_config, or fail2ban banning your IP.

Fix: Check `/etc/hosts.deny` and `/etc/hosts.allow`. Check `sshd_config` for `AllowUsers`. Check fail2ban: `sudo fail2ban-client status sshd` — your IP may be banned.

Timeout / connection hangs and never connects

Cause: A firewall is silently dropping packets (no rejection, just silence), or the server is unreachable.

Fix: Try `ssh -v` — if it hangs on "Connecting to..." the problem is network/firewall before the SSH daemon. Check UFW rules on the server. Check that port 22 is open: `nc -zv server 22` from your machine.

Keeping a Connection Alive

SSH connections that sit idle too long get dropped — by NAT routers, firewalls, or the server's own timeout settings. Fix this by sending keepalive packets from the client side. Either add this to your `~/.ssh/config` (Chapter 4 covers this in full) or pass it inline:

philip@debian — keepalive inline

```
# Send a keepalive packet every 60 seconds; give up after 3 missed replies philip@debian:~$ ssh -o ServerAliveInterval=60 -o ServerAliveCountMax=3 philip@server
```

Quick Reference

Task	Command
Open a shell session	`ssh philip@server`
Connect on a custom port	`ssh -p 2222 philip@server`
Use a specific key file	`ssh -i ~/.ssh/id_work philip@server`
Run one command and exit	`ssh philip@server df -h`
Run a piped command	`ssh philip@server "df -h | grep sda"`
Debug a failed connection	`ssh -v philip@server`
Even more debug detail	`ssh -vvv philip@server`
Remove stale host key	`ssh-keygen -R hostname`
Test TCP connectivity to port 22	`nc -zv server 22`
End the session	`exit` or `Ctrl+D`
Escape sequence (if stuck)	`Enter, then ~.`

Escape sequence `~.` — if your SSH session freezes (network dropped, server hung) and `Ctrl+C` does nothing, press `Enter` then type `~.` (tilde followed by a full stop). This tells the SSH client to drop the connection immediately, returning you to your local prompt.

Next — Chapter 3: Key-Based Authentication. Typing a password every time is tedious and less secure than using a key pair. Chapter 3 covers generating an Ed25519 key pair, copying your public key to a server with `ssh-copy-id`, the `authorized_keys` file, setting the correct permissions, and disabling password authentication once keys are working.

CHAPTER 3 — KEY-BASED AUTHENTICATION

Chapter 3 — Key-Based Authentication

Typing a password every time you connect is slow, and passwords can be guessed, leaked, or brute-forced. Key-based authentication replaces the password with a cryptographic key pair — faster to use, significantly harder to attack, and the standard approach for any server you manage regularly. Once it's set up, connecting feels like magic: you type `ssh philip@server` and you're in, no password prompt at all.

> **How it works in one sentence:** you generate a key pair, put the public key on the server, and SSH proves your identity by signing a challenge with your private key — the server verifies the signature without your private key ever leaving your machine.

The Setup Flow

1. **Generate a key pair on your local machine**
   Creates two files: `~/.ssh/id_ed25519` (private) and `~/.ssh/id_ed25519.pub` (public). Runs on your machine only — nothing touches the server yet.
2. **Copy your public key to the server**
   Use `ssh-copy-id` to append your public key to `~/.ssh/authorized_keys` on the server. This step still uses your password — it's the last time you'll need it.
3. **Test key-based login**
   Connect with `ssh philip@server` — if it works without asking for a password, keys are set up correctly.
4. **Optionally: disable password authentication**
   Once keys work, disable password login on the server entirely. This eliminates brute-force attacks. **Only do this after confirming key login works from a second terminal.**

Step 1 — Generate Your Key Pair

Run this on your **local machine** — the one you're connecting *from*. Use Ed25519 unless you specifically need RSA for an older server.

philip@debian — generate an Ed25519 key pair

```
philip@debian:~$ ssh-keygen -t ed25519 -C "philip@debian" Generating public/private ed25519 key pair. Enter file in which to save the key (/home/philip/.ssh/id_ed25519): press Enter to accept default Enter passphrase (empty for no passphrase): recommended: enter a passphrase Enter same passphrase again: Your identification has been saved in /home/philip/.ssh/id_ed25519 Your public key has been saved in /home/philip/.ssh/id_ed25519.pub The key fingerprint is: SHA256:xyz789ABCdef123GHI456jklMNO philip@debian The key's randomart image is: +--[ED25519 256]--+ | .o+ | | . o.o | | . + + . | +----[SHA256]-----+
```

What the flags mean

- `-t ed25519` — use the Ed25519 algorithm (modern, fast, recommended).
- `-C "philip@debian"` — a comment appended to the public key. Helps identify which machine a key came from when you have several. Use anything descriptive: your email, hostname, or purpose.

Should I set a passphrase?

Yes — and here's why. Your private key is just a file. If someone gets access to your machine (stolen laptop, compromised account), they can use your private key to connect to every server it's authorised on. A passphrase encrypts the key file itself, so stealing the file alone is not enough.

The passphrase is only ever typed on your local machine — never sent over the network. Use the SSH agent (below) so you only type it once per login session, not on every connection.

PUBLIC KEY — SHARE FREELY

`~/.ssh/id_ed25519.pub`

Copy this to every server you want to access. It's safe to email, paste, or put in a GitHub profile. Begins with `ssh-ed25519 AAAA...`

PRIVATE KEY — NEVER SHARE

`~/.ssh/id_ed25519`

Stays on your machine only. If you ever paste this somewhere, treat it as compromised and generate a new pair immediately. Begins with `-----BEGIN OPENSSH PRIVATE KEY-----`

Step 2 — Copy Your Public Key to the Server

The easy way: ssh-copy-id

`ssh-copy-id` handles everything — it connects to the server, creates the `~/.ssh/` directory if needed, appends your public key to `authorized_keys`, and sets the correct permissions. This is one of the most useful commands in the SSH toolkit.

philip@debian — copy public key to server

```
philip@debian:~$ ssh-copy-id philip@192.168.1.100 /usr/bin/ssh-copy-id: INFO: Source of key(s) to be installed: "/home/philip/.ssh/id_ed25519.pub" /usr/bin/ssh-copy-id: INFO: attempting to log in with
```

```
the new key(s), to filter out any that are already installed /usr/bin/ssh-copy-id: INFO: 1 key(s)
remain to be installed -- if you are prompted now it is to install the new keys
philip@192.168.1.100's password: ← last time you need the password Number of key(s) added: 1 Now try
logging into the machine, with: "ssh 'philip@192.168.1.100'" and check to make sure that only the
key(s) you wanted were added.
```

If you need to specify a key file or a non-standard port:

philip@debian — ssh-copy-id with options

```
# Specify a key file explicitly philip@debian:~$ ssh-copy-id -i ~/.ssh/id_ed25519.pub philip@server
# Server on a non-standard port philip@debian:~$ ssh-copy-id -p 2222 philip@server
```

The manual way — if ssh-copy-id isn't available

On Windows (before OpenSSH was built in) or on minimal systems, `ssh-copy-id` may not be present. Do it manually:

philip@debian — manual key copy (one-liner)

```
# Pipe the public key into the server's authorized_keys in one command philip@debian:~$ cat
~/.ssh/id_ed25519.pub | ssh philip@server \ "mkdir -p ~/.ssh && chmod 700 ~/.ssh && \ cat >>
~/.ssh/authorized_keys && \ chmod 600 ~/.ssh/authorized_keys" philip@server's password:
```

From Windows PowerShell

PowerShell — copy public key to Linux server

```
PS C:\Users\Philip> type $env:USERPROFILE\.ssh\id_ed25519.pub | ssh philip@server "mkdir -p ~/.ssh
&& cat >> ~/.ssh/authorized_keys && chmod 600 ~/.ssh/authorized_keys"
```

Step 3 — Test Key Login

philip@debian — test key-based login

```
philip@debian:~$ ssh philip@192.168.1.100 (if key has a passphrase, you're prompted for it once —
not the server password) Enter passphrase for key '/home/philip/.ssh/id_ed25519': Welcome to Debian
GNU/Linux 12 (bookworm) philip@server:~$ ← in without typing the server password
```

If it still asks for the server password rather than the key passphrase, see the troubleshooting section at the end of this chapter.

The SSH Agent — Type Your Passphrase Once

With a passphrase-protected key, you'd need to type the passphrase every time you connect. The SSH agent solves this: it holds your decrypted key in memory for the duration of your login session, so you type the passphrase once and SSH silently uses it for every subsequent connection.

philip@debian — using the SSH agent

```
# Start the agent (usually already running on a Linux desktop) philip@debian:~$ eval "$(ssh-agent -
s)" Agent pid 12345 # Add your key — type the passphrase once here philip@debian:~$ ssh-add
~/.ssh/id_ed25519 Enter passphrase for /home/philip/.ssh/id_ed25519: Identity added:
/home/philip/.ssh/id_ed25519 (philip@debian) # All subsequent connections use the cached key — no
passphrase prompt philip@debian:~$ ssh philip@server philip@server:~$ ← straight in, no prompt #
List keys currently held by the agent philip@debian:~$ ssh-add -l 256 SHA256:xyz789...
/home/philip/.ssh/id_ed25519 (ED25519)
```

> **On most Linux desktops** (GNOME, KDE, XFCE) the SSH agent starts automatically when you log in and a keyring prompt handles the passphrase transparently. On macOS, keys are stored in the system keychain. On Windows, enable the "OpenSSH Authentication Agent" service in Services and set it to start automatically.

What authorized_keys Looks Like

The `authorized_keys` file on the server contains one public key per line. Multiple keys means multiple machines can connect — useful if you want both your laptop and desktop to access the same server.

```
# ~/.ssh/authorized_keys on the SERVER
# One public key per line — each grants access from the machine that holds the matching private ke

ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAI... philip@debian
ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAI... philip@windows-laptop
ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAI... philip@work-machine
```

You can also prefix a key with options to restrict what it can do:

```
# Restrict a key to only run one specific command (useful for automation)
command="backup.sh",no-pty,no-agent-forwarding ssh-ed25519 AAAAC3... backup@cronjob

# Restrict to connections from a specific IP only
from="192.168.1.50" ssh-ed25519 AAAAC3... philip@trusted-machine
```

Critical: File Permissions

SSH is paranoid about permissions — intentionally. If the `~/.ssh` directory or `authorized_keys` file is writable by anyone other than you, SSH refuses to use them. This prevents another user on the same system from injecting their own key into your file.

File / Directory	Required Permission	Owner	Why
`~/.ssh/`	`700`	you	Only you can read or write the directory
`~/.ssh/authorized_keys`	`600`	you	Only you can read or write the key list
`~/.ssh/id_ed25519`	`600`	you	Private key — only you should ever read it
`~/.ssh/id_ed25519.pub`	`644`	you	Public key — readable by others, no harm
`~/.ssh/config`	`600`	you	Config file — contains hostnames and settings
`~/.ssh/ group/world writable`	`755 or 777`	—	SSH will refuse to use authorized_keys entirely

philip@server — fix permissions if needed

```
philip@server:~$ chmod 700 ~/.ssh philip@server:~$ chmod 600 ~/.ssh/authorized_keys philip@server:~$ chmod 600 ~/.ssh/id_ed25519 # Verify philip@server:~$ ls -la ~/.ssh/ drwx------ 2 philip philip 4096 Jun 17 20:00 . -rw------- 1 philip philip 571 Jun 17 20:00 authorized_keys -rw------- 1 philip philip 411 Jun 17 20:00 id_ed25519 -rw-r--r-- 1 philip philip 96 Jun 17 20:00 id_ed25519.pub
```

Step 4 — Disable Password Authentication (Optional but Recommended)

Once key-based login works, disabling password authentication closes the door on brute-force attacks completely. An attacker would need your private key file — guessing passwords becomes irrelevant.

> **Confirm key login works from a second terminal before doing this.** Open a new terminal window and verify you can log in with your key. Only then edit sshd_config in your existing session. If you disable password auth and your key doesn't work, you'll be locked out.

philip@server — disable password authentication

```
philip@server:~$ sudo vi /etc/ssh/sshd_config
```

Find and set these three lines (search with `/PasswordAuth` in vi):

```
# /etc/ssh/sshd_config — find these lines and set them as shown

PasswordAuthentication no
PubkeyAuthentication   yes
AuthorizedKeysFile     .ssh/authorized_keys
```

philip@server — reload SSH daemon

```
# Check config is valid before reloading philip@server:~$ sudo sshd -t (no output = config is valid)
philip@server:~$ sudo systemctl reload sshd # Now test from a NEW terminal — password auth should be
rejected philip@debian:~$ ssh -o PreferredAuthentications=password philip@server philip@server:
Permission denied (publickey). ← correct — password auth is now disabled
```

Managing Multiple Keys

If you connect to many servers — work, home, cloud, Raspberry Pi — you might want separate key pairs for different purposes. Use `-b` to name them at generation time:

philip@debian — multiple named key pairs

```
# Generate a separate key for work servers philip@debian:~$ ssh-keygen -t ed25519 -C "philip-work" -
f ~/.ssh/id_work Generating public/private ed25519 key pair. Your identification has been saved in
/home/philip/.ssh/id_work Your public key has been saved in /home/philip/.ssh/id_work.pub # Generate
a key for a Pi project philip@debian:~$ ssh-keygen -t ed25519 -C "philip-pi" -f ~/.ssh/id_pi #
Connect using a specific key philip@debian:~$ ssh -i ~/.ssh/id_work philip@work-server.example.com
philip@debian:~$ ssh -i ~/.ssh/id_pi pi@raspberrypi.local
```

> **Chapter 4 (SSH config file)** makes managing multiple keys much cleaner — you can set which key to use per host automatically, so you never need to remember the `-i` flag.

Revoking Access — Removing a Key

To stop a machine connecting to a server, remove its public key from `authorized_keys` on the server. Each key is one line — delete the line for the machine you want to revoke:

philip@server — remove a specific key

```
philip@server:~$ vi ~/.ssh/authorized_keys Delete the line containing the key you want to revoke,
save and exit No service restart needed — authorized_keys is read on every connection attempt
```

Troubleshooting Key Login

If SSH still asks for the server password after copying your key, work through these checks in order:

philip@debian — debug key authentication

```
# 1. Check which key SSH is trying to use philip@debian:~$ ssh -v philip@server 2>&1 | grep -E "Offering|Authenticating|refused" debug1: Offering public key: /home/philip/.ssh/id_ed25519 ED25519 debug1: Authentications that can continue: publickey,password # 2. Check permissions on the server philip@server:~$ ls -la ~/.ssh/ .ssh must be 700, authorized_keys must be 600 # 3. Check the public key is actually in authorized_keys philip@server:~$ cat ~/.ssh/authorized_keys Should contain a line starting with ssh-ed25519 # 4. Check the server's sshd_config allows key auth philip@server:~$ grep -E "PubkeyAuth|AuthorizedKeys" /etc/ssh/sshd_config PubkeyAuthentication yes AuthorizedKeysFile .ssh/authorized_keys # 5. Check the auth log on the server for the real error philip@server:~$ sudo journalctl -u sshd -n 20 Look for lines mentioning your IP — it will say exactly why the key was rejected
```

Quick Reference

Task	Command
Generate Ed25519 key pair	`ssh-keygen -t ed25519 -C "comment"`
Generate named key pair	`ssh-keygen -t ed25519 -f ~/.ssh/id_work`
Copy public key to server	`ssh-copy-id philip@server`
Copy key, custom port	`ssh-copy-id -p 2222 philip@server`
Start SSH agent	`eval "$(ssh-agent -s)"`
Add key to agent	`ssh-add ~/.ssh/id_ed25519`
List keys in agent	`ssh-add -l`
Remove all keys from agent	`ssh-add -D`
View your public key	`cat ~/.ssh/id_ed25519.pub`
Fix .ssh directory permissions	`chmod 700 ~/.ssh && chmod 600 ~/.ssh/authorized_keys`
Test with verbose output	`ssh -v philip@server`
Check sshd auth log	`sudo journalctl -u sshd -n 30`

Next — Chapter 4: The SSH Config File. Typing `ssh -p 2222 -i ~/.ssh/id_work philip@work-server.example.com` every time is tedious. The `~/.ssh/config` file lets you define aliases, set per-host options, and manage multiple keys automatically — so `ssh work` is all you need to type.

CHAPTER 4 — THE SSH CONFIG FILE

Chapter 4 — The SSH Config File

By now you know how to connect, use keys, and specify flags. But typing `ssh -p 2222 -i ~/.ssh/id_work philip@work-server.example.com` every time is nobody's idea of fun. The SSH config file — `~/.ssh/config` — lets you store all of that per host, so the full command above becomes simply `ssh work`. It also handles jump hosts, keepalives, agent forwarding, and anything else you'd normally pass as a flag.

> **Location:** `~/.ssh/config` on Linux and macOS. On Windows with OpenSSH it's `C:\Users\Philip\.ssh\config` — same format, same directives. Create the file if it doesn't exist; SSH reads it automatically on every connection.

Before and After

WITHOUT CONFIG — TYPE EVERYTHING

```
ssh -p 2222 -i ~/.ssh/id_work \
philip@work.example.com ssh -i ~/.ssh/id_pi \
pi@192.168.1.101 ssh -p 22 -i ~/.ssh/id_ed25519
\ philip@192.168.1.100
```

WITH CONFIG — SHORT ALIASES

```
ssh work ssh pi ssh server
```

Config File Structure

The file is a series of `Host` blocks. Each block starts with a `Host` line giving a nickname, followed by indented directives that apply to that host. The nickname is what you type in the terminal — it doesn't have to match the real hostname.

```
# ~/.ssh/config
# Lines starting with # are comments — ignored by SSH

Host server                    ← the nickname you type: ssh server
    HostName      192.168.1.100  ← real IP or hostname
    User          philip         ← login username
    Port          22             ← port (omit if 22)
    IdentityFile  ~/.ssh/id_ed25519

Host pi
    HostName      192.168.1.101
    User          pi
```

```
    IdentityFile  ~/.ssh/id_pi

Host work
    HostName      work.example.com
    User          philip
    Port          2222
    IdentityFile  ~/.ssh/id_work
```

philip@debian — using config aliases

```
# All of these use the settings from ~/.ssh/config philip@debian:~$ ssh server philip@server:~$
philip@debian:~$ ssh pi pi@raspberrypi:~$ philip@debian:~$ ssh work philip@work:~$
```

> **Config works everywhere SSH does** — the alias works with `scp`, `sftp`, and `rsync` too. Once `pi` is defined in your config, `rsync -av files/ pi:~/files/` just works.

The Global Default Block

A `Host *` block at the end of the file sets defaults that apply to every connection unless a more specific block overrides them. Put keepalives, compression, and agent forwarding defaults here so you don't repeat them in every host block.

```
# ~/.ssh/config — global defaults at the END of the file
# Host * must come LAST — specific blocks above take priority

Host server
    HostName  192.168.1.100
    User      philip

Host work
    HostName  work.example.com
    User      philip
    Port      2222

Host *                          ← applies to ALL connections
    ServerAliveInterval  60     ← keepalive every 60 seconds
    ServerAliveCountMax  3      ← drop after 3 missed keepalives
    AddKeysToAgent       yes    ← auto-add keys to ssh-agent
    IdentityFile         ~/.ssh/id_ed25519  ← default key
```

> **Order matters:** SSH reads the config top to bottom and uses the *first* matching value it finds for each directive. Put specific host blocks at the top and `Host *` defaults at the bottom — not the other way around.

All Useful Directives

Directive	What it does	Example value
`HostName`	The real hostname or IP to connect to	`192.168.1.100`
`User`	Login username on the remote machine	`philip`
`Port`	Port to connect on (default 22)	`2222`
`IdentityFile`	Path to the private key file to use	`~/.ssh/id_work`
`IdentitiesOnly`	Only use the key specified — don't try others from the agent	`yes`
`ServerAliveInterval`	Send a keepalive packet every N seconds	`60`
`ServerAliveCountMax`	Drop connection after N missed keepalive replies	`3`
`AddKeysToAgent`	Automatically add a used key to the SSH agent	`yes`
`ForwardAgent`	Forward your local SSH agent to the remote machine (use with care)	`yes`
`ProxyJump`	Connect via a jump host (see section below)	`jumphost`
`LocalForward`	Forward a local port to a remote address via the tunnel	`8080 localhost:80`
`Compression`	Enable compression — useful on slow connections	`yes`
`StrictHostKeyChecking`	How to handle unknown or changed host keys	`accept-new`
`LogLevel`	Verbosity of SSH output (QUIET, ERROR, INFO, VERBOSE, DEBUG)	`QUIET`
`ConnectTimeout`	Seconds to wait before giving up on connecting	`10`
`ControlMaster`	Reuse an existing connection for new sessions (multiplexing)	`auto`
`ControlPath`	Socket file for connection multiplexing	`~/.ssh/cm-%r@%h:%p`

Wildcards and Pattern Matching

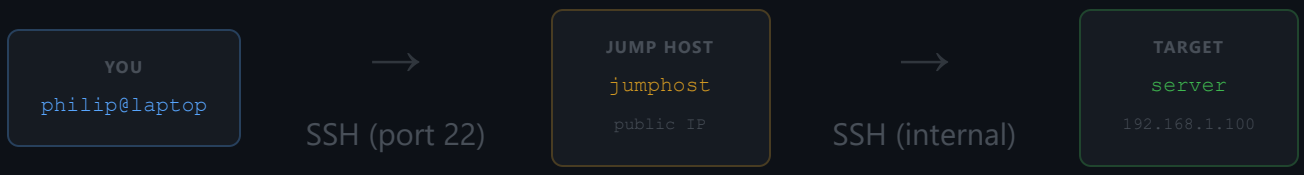
The `Host` keyword supports wildcards, which is useful for groups of similar hosts:

```
# Match all hosts ending in .example.com
Host *.example.com
    User          philip
    IdentityFile  ~/.ssh/id_work

# Match all hosts starting with pi
Host pi*
    User          pi
    IdentityFile  ~/.ssh/id_pi
```

```
# Match multiple names in one Host line (space-separated)
Host server homeserver debian-box
    HostName      192.168.1.100
    User          philip
```

ProxyJump — Connecting Through a Jump Host

A jump host (also called a bastion host) is an intermediate machine you connect through to reach a target that isn't directly accessible — for example, a server on a private network behind a firewall, or a Raspberry Pi on your home network when you're away from home.



Before `ProxyJump`, you had to SSH into the jump host first, then SSH again to the target — two hops, two prompts, two sessions to manage. `ProxyJump` makes SSH handle both hops transparently in a single command.

ProxyJump in the config file

```
# ~/.ssh/config

# The jump host — publicly accessible
Host jumphost
    HostName     jump.example.com
    User         philip
    IdentityFile ~/.ssh/id_ed25519

# The target — only reachable via the jump host
Host internal-server
    HostName     192.168.1.100
    User         philip
    IdentityFile ~/.ssh/id_ed25519
    ProxyJump    jumphost    ← connect via jumphost automatically
```

philip@laptop — jumping through to internal server

```
philip@laptop:~$ ssh internal-server (SSH connects to jumphost, then tunnels through to
192.168.1.100) philip@internal-server:~$ ← landed on the target, one command
```

ProxyJump inline (without editing config)

philip@laptop — ProxyJump as a flag

```
philip@laptop:~$ ssh -J philip@jumphost philip@192.168.1.100 # Chain multiple jump hosts
philip@laptop:~$ ssh -J hop1,hop2 philip@final-target
```

> **Your key stays on your machine.** With `ProxyJump`, your private key is only used by your local SSH client — it never touches the jump host. This is safer than the old approach of copying your key to the jump host and using agent forwarding. Always prefer `ProxyJump` over `ProxyCommand ssh -A` for this reason.

Connection Multiplexing — One Connection, Many Sessions

Every time you open a new terminal and SSH to the same server, the full handshake runs again. Multiplexing reuses the first connection as a shared channel — subsequent connections to the same host open in milliseconds and don't require re-authentication.

```
# ~/.ssh/config — multiplexing settings

Host *
    ControlMaster  auto              ← reuse existing connections
    ControlPath    ~/.ssh/cm-%r@%h:%p ← socket file per connection
    ControlPersist 10m               ← keep master alive 10 min after you exit
```

With this in place, the second `ssh server` you open in a new tab connects almost instantly — no new handshake, no passphrase prompt, because it tunnels through the existing session.

A Complete Real-World Config

This is what Philip's `~/.ssh/config` might look like after setting up all the hosts covered in this course:

```
# ~/.ssh/config — Philip's full config

# ── Home server (Debian) ──────────────────────────
Host server
    HostName      192.168.1.100
    User          philip
    IdentityFile  ~/.ssh/id_ed25519

# ── Raspberry Pi ──────────────────────────────────
Host pi
    HostName      192.168.1.101
    User          pi
```

```
    IdentityFile  ~/.ssh/id_pi

# ── osztromok.com website server ──────────────────────
Host osztromok
    HostName      osztromok.com
    User          philip
    IdentityFile  ~/.ssh/id_ed25519
    Port          22

# ── Work server (non-standard port) ──────────────────
Host work
    HostName      work.example.com
    User          philip
    Port          2222
    IdentityFile  ~/.ssh/id_work

# ── Internal server via jump host ────────────────────
Host internal
    HostName      192.168.10.50
    User          philip
    ProxyJump     work
    IdentityFile  ~/.ssh/id_work

# ── Global defaults (must be LAST) ───────────────────
Host *
    ServerAliveInterval  60
    ServerAliveCountMax  3
    AddKeysToAgent       yes
    ControlMaster        auto
    ControlPath          ~/.ssh/cm-%r@%h:%p
    ControlPersist       10m
```

Creating and Editing the Config File

philip@debian — create and protect the config file

```
# Create it if it doesn't exist, then edit philip@debian:~$ touch ~/.ssh/config philip@debian:~$
chmod 600 ~/.ssh/config philip@debian:~$ vi ~/.ssh/config # Verify SSH reads it correctly (prints
the resolved settings for a host) philip@debian:~$ ssh -G server hostname 192.168.1.100 user philip
port 22 identityfile ~/.ssh/id_ed25519 serveraliveinterval 60 ...
```

> `ssh -G hostname` is the config file debugger — it prints every resolved setting for that host (including defaults inherited from `Host *`) without actually connecting. Use it to confirm your config is being read the way you expect.

StrictHostKeyChecking — Handle with Care

One directive worth a specific mention: `StrictHostKeyChecking` controls what happens when SSH sees an unknown or changed host key.

Value	Behaviour	When to use
`yes`	Refuse connection if key is unknown or changed. Never auto-add.	Maximum security — you manually manage known_hosts.
`accept-new`	Auto-add unknown hosts but refuse changed keys. Good balance.	Recommended for most uses — protects against MITM, allows new hosts.
`ask` *(default)*	Prompt for unknown hosts; refuse changed keys.	The default — fine for interactive use.
`no`	Accept any key silently — never check known_hosts.	Scripting against ephemeral hosts (containers, VMs that change keys constantly). Never use on real servers.

> **Don't put `StrictHostKeyChecking no` in your global `Host *` block.** It's tempting when you're tired of prompts, but it disables the protection that detects man-in-the-middle attacks. If you need it for scripting, set it only for the specific host or a specific IP range in its own `Host` block.

Quick Reference

Task	How to do it
Create the config file	`touch ~/.ssh/config && chmod 600 ~/.ssh/config`
Edit it	`vi ~/.ssh/config`
Debug / print resolved settings	`ssh -G server`
Connect using an alias	`ssh server`
Override a config setting inline	`ssh -o Port=2222 server`
Jump through a host (inline)	`ssh -J jumphost philip@target`
Use config alias with sftp	`sftp server`
Use config alias with rsync	`rsync -av files/ server:~/files/`

Next — Chapter 5: Port Forwarding & Tunnelling. SSH can do much more than give you a shell — it can create encrypted tunnels that forward traffic from a local port through the SSH connection to any address the server can reach. Chapter 5 covers local forwarding, remote forwarding, dynamic (SOCKS proxy) forwarding, and practical use cases for each.

CHAPTER 5 — PORT FORWARDING & TUNNELLING

Chapter 5 — Port Forwarding & Tunnelling

SSH is more than a remote shell. It can create encrypted tunnels that carry arbitrary TCP traffic between machines — forwarding ports across firewalls, exposing services on the internet, or routing your entire browser through a remote server. This chapter covers the three forwarding modes: local, remote, and dynamic.

> **What is port forwarding?** When SSH forwards a port, it listens on a port on one machine and transparently delivers traffic to a port on another machine — through the encrypted SSH connection. The application sending or receiving traffic doesn't know there's a tunnel involved.

The Three Types at a Glance

LOCAL FORWARDING

A port on *your machine* forwards to a port the server can reach.

```
ssh -L
local_port:target_host:target_po
server
```

Use when: you want to reach a remote service from your machine — a database, internal website, or admin panel that isn't exposed publicly.

REMOTE FORWARDING

A port on the *server* forwards back to a port your machine can reach.

```
ssh -R
remote_port:target_host:target_p
server
```

Use when: you want the server (or internet) to reach something on your local machine — exposing a dev server or a machine behind NAT.

DYNAMIC (SOCKS PROXY)

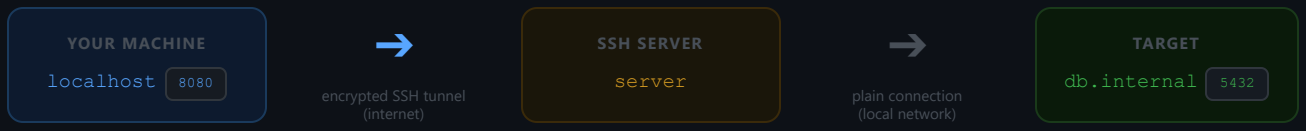
A local SOCKS proxy port routes *any* traffic through the server.

```
ssh -D local_port server
```

Use when: you want to browse the web (or run any app) as if you were on the server's network — bypassing geo-restrictions or local network filtering.

Local Port Forwarding (-L)

The most common use case: you want to reach something on the remote side that isn't directly accessible from your machine. Your laptop connects to `localhost:local_port`, and SSH silently delivers that traffic through the tunnel to `target_host:target_port` — as seen from the server.



```
# Syntax
ssh -L LOCAL_PORT:TARGET_HOST:TARGET_PORT SSH_SERVER

# Examples
# Access a remote MySQL database on your local port 3307
ssh -L 3307:localhost:3306 server

# Access an internal web service at db.internal:8080 via local port 8080
ssh -L 8080:db.internal:8080 server

# Use the config alias (server must be defined in ~/.ssh/config)
ssh -L 8080:db.internal:8080 server
```

philip@laptop — tunnel to remote MySQL

```
# Open the tunnel — SSH gives you a shell AND forwards the port philip@laptop:~$ ssh -L
3307:localhost:3306 server philip@server:~$ ← shell open on server, tunnel active # In a second
terminal on your laptop — connect to remote MySQL locally philip@laptop:~$ mysql -h 127.0.0.1 -P
3307 -u philip -p Welcome to the MySQL monitor. Commands end with ; or \g. ← you are connected to
the remote MySQL through the tunnel
```

Run the tunnel in the background (-f -N)

If you don't need an interactive shell — just the tunnel — add `-f` (background) and `-N` (no command / don't open a shell):

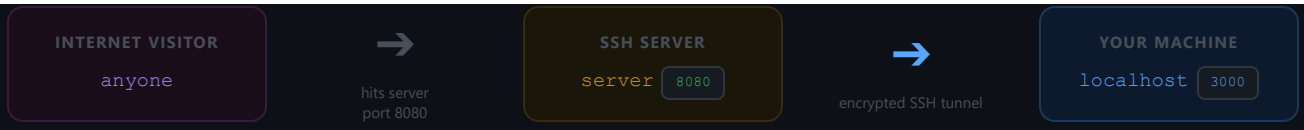
philip@laptop — background tunnel, no shell

```
# -f = fork to background after auth, -N = no shell, just forward the port philip@laptop:~$ ssh -fN
-L 3307:localhost:3306 server philip@laptop:~$ ← prompt returns immediately; tunnel running in
background # To find and kill the background tunnel later philip@laptop:~$ ps aux | grep ssh philip
12345 0.0 0.0 ssh -fN -L 3307:localhost:3306 server philip@laptop:~$ kill 12345
```

> **Bind to 0.0.0.0 to share the tunnel on your local network.** By default, the local port listens only on `127.0.0.1` (just your machine). Use `ssh -L 0.0.0.0:8080:target:80 server` to let other devices on your network use the tunnel too.

Remote Port Forwarding (-R)

The mirror image of local forwarding. A port opens on the *server* and traffic arriving there is forwarded back through the tunnel to a port your machine can reach. The canonical use case is exposing a local development server to the internet without any router configuration.




```
# Syntax
ssh -R REMOTE_PORT:LOCAL_HOST:LOCAL_PORT SSH_SERVER

# Expose your local dev server (port 3000) on the server's port 8080
ssh -R 8080:localhost:3000 server

# Anyone hitting http://server:8080 now reaches your local port 3000
```

philip@laptop — expose local dev server to internet

```
# Your React/Node/Flask app is running on port 3000 philip@laptop:~$ npm start Local:
http://localhost:3000 # In another terminal — open reverse tunnel philip@laptop:~$ ssh -R
8080:localhost:3000 server philip@server:~$ # Your client can now open
http://server.example.com:8080 and see your app # No router changes, no port opening, no DynDNS —
just SSH
```

Allow external connections to the remote port

By default, the remote port binds only to `127.0.0.1` on the server — reachable from the server itself but not from outside. To allow external access, set `GatewayPorts yes` in the server's `/etc/ssh/sshd_config`, or specify the bind address explicitly:

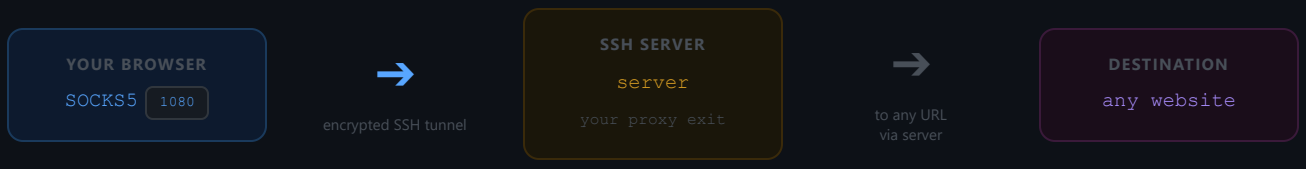
```
# Bind to all interfaces on the server — anyone can reach it
ssh -R 0.0.0.0:8080:localhost:3000 server

# On the server — enable GatewayPorts in sshd_config if needed
# /etc/ssh/sshd_config
GatewayPorts yes
# Then: sudo systemctl restart sshd
```

> **Remote forwarding opens your local machine to the internet** — only do it on a server you trust and control. Anyone who can reach the server's open port can connect to your local service. Always restrict access with firewall rules if exposing sensitive services.

Dynamic Port Forwarding — SOCKS Proxy (-D)

Instead of forwarding one specific port to one specific destination, dynamic forwarding creates a SOCKS5 proxy on a local port. Any SOCKS-aware application (browser, curl, git) can be pointed at it, and all its traffic will travel

through the SSH connection — leaving the server and appearing to originate from the server's IP address.




```
# Open a SOCKS5 proxy on local port 1080
ssh -D 1080 server

# Or background it so you keep your terminal
ssh -fN -D 1080 server
```

Then point your browser's proxy settings at `SOCKS5 127.0.0.1:1080`:

philip@laptop — SOCKS proxy via curl and Firefox

```
# Start the SOCKS proxy in the background philip@laptop:~$ ssh -fN -D 1080 server # Use it with curl — your traffic exits from server's IP philip@laptop:~$ curl --socks5 127.0.0.1:1080
https://ifconfig.me 203.0.113.42 ← server's IP address, not your laptop's # Use it with git — route all git traffic through the tunnel philip@laptop:~$ git config --global http.proxy
socks5://127.0.0.1:1080 # In Firefox: Settings → Network → Manual Proxy → SOCKS5 Host: 127.0.0.1 Port: 1080 # Enable "Proxy DNS when using SOCKS v5" to also tunnel DNS queries
```

Practical Use Cases

Access a remote database securely

MySQL/Postgres only listens on 127.0.0.1 on the server (correct). Tunnel it to your laptop so your GUI client (TablePlus, DBeaver) can connect.

```
ssh -fN -L 3307:localhost:3306 server
# Connect TablePlus to 127.0.0.1:3307
```

Browse an internal admin panel

Grafana / phpMyAdmin / Proxmox web UI is on an internal host not exposed to the internet.

```
ssh -fN -L 8080:grafana.internal:3000 server
# Open http://localhost:8080 in your browser
```

Show a client your local dev site

Your Node app is running on localhost:3000. Client wants to see it now, no time to deploy.

```
ssh -R 8080:localhost:3000 server
# Client opens http://server.example.com:8080
```

SSH into a machine behind NAT

Your home Raspberry Pi is behind a home router. You want SSH access from work without port forwarding the router.

```
# Run on the Pi at home:
ssh -fN -R 2222:localhost:22 server
# From work:
ssh -p 2222 pi@server
```

Browse the web via server's network

You're on a restrictive hotel/coffee-shop Wi-Fi and want to route traffic through your VPS.

```
ssh -fN -D 1080 server
# Set browser SOCKS5 proxy: 127.0.0.1:1080
```

Access entire remote subnet

Multiple services on a private network (192.168.10.x). Forward a different local port for each, or use SOCKS and configure the browser.

```
ssh -fN -D 1080 server
# Point browser proxy to 127.0.0.1:1080
# All 192.168.10.x addresses now reachable
```

Port Forwarding in the SSH Config File

Frequently used tunnels belong in `~/.ssh/config` — then you don't need to remember the flags, and the tunnel opens automatically when you connect to that host:

```
# ~/.ssh/config — permanent local tunnel for the database

Host server
    HostName         192.168.1.100
    User             philip
    IdentityFile     ~/.ssh/id_ed25519
    LocalForward     3307 localhost:3306   ← always tunnel MySQL
    LocalForward     8080 grafana.internal:3000  ← and Grafana

# Now ssh server automatically sets up both tunnels
```

```
# config directives for each forwarding type

LocalForward   local_port target_host:target_port   ← -L equivalent
RemoteForward  remote_port local_host:local_port    ← -R equivalent
DynamicForward local_port                           ← -D equivalent
```

Quick Reference

Flag / directive	What it does
`-L local:host:remote`	Local forward — local port → remote target (via server)
`-R remote:host:local`	Remote forward — server port → your local target
`-D port`	Dynamic SOCKS5 proxy on local port
`-N`	No shell — just set up the tunnel, no interactive session

Flag / directive	What it does
`-f`	Fork to background after authenticating
`-fN`	Background tunnel with no shell (use together for clean background tunnels)
`LocalForward`	Put `-L` equivalent in `~/.ssh/config`
`RemoteForward`	Put `-R` equivalent in `~/.ssh/config`
`DynamicForward`	Put `-D` equivalent in `~/.ssh/config`
`GatewayPorts yes`	Allow external connections to remote-forwarded ports (in sshd_config)
`curl --socks5 127.0.0.1:PORT url`	Use a SOCKS proxy with curl
`ps aux | grep ssh`	Find background SSH tunnel processes to kill them

Next — Chapter 6: SFTP. SSH can transfer files as well as open shells. SFTP (SSH File Transfer Protocol) runs over the same connection and gives you an interactive file browser — upload, download, rename, delete, and navigate directories on the remote machine. Chapter 6 covers the `sftp` command, its most useful subcommands, batch transfers, and GUI clients like FileZilla and WinSCP.

CHAPTER 6 — SFTP

Chapter 6 — SFTP

SFTP — SSH File Transfer Protocol — is the standard way to transfer files over an SSH connection. Unlike the old FTP protocol (which sends credentials in plain text and requires a separate data channel), SFTP runs entirely inside the encrypted SSH session. If you can SSH into a machine, you can SFTP into it without any extra configuration.

> **SFTP is not FTP over SSH.** Despite the similar name, SFTP is a completely separate protocol developed as part of SSH2. It has nothing in common with FTP or FTPS — it just happens to do the same job (file transfer) more securely.

Connecting

The `sftp` command takes the same arguments as `ssh` and supports all the same options — including aliases from your `~/.ssh/config`:

```
# Standard connection — same syntax as ssh
sftp user@hostname

# With a non-standard port
sftp -P 2222 user@hostname   ← note: -P not -p (capital P)

# Using a specific key
sftp -i ~/.ssh/id_work user@hostname

# Using a config alias (works exactly like ssh)
sftp server
sftp pi
sftp work
```

philip@laptop — connecting with sftp

```
philip@laptop:~$ sftp server Connected to server. sftp> ← you are now in the sftp prompt
```

Navigating — Local and Remote at the Same Time

The SFTP shell manages two locations simultaneously: one on the remote machine and one on your local machine. Most commands come in pairs — the plain version acts on the remote side, and the version prefixed with `l` acts locally.

sftp — navigating remote and local directories

```
# Remote side — where you are on the server sftp> pwd Remote working directory: /home/philip sftp> ls Desktop Documents Downloads projects www sftp> cd www/html sftp> ls -la drwxr-xr-x 2 philip www-data 4096 Jun 15 10:22 . -rw-r--r-- 1 philip www-data 2048 Jun 15 10:22 index.html # Local side — where you are on YOUR machine sftp> lpwd Local working directory: /home/philip sftp> lcd ~/projects/website sftp> lls index.html style.css app.js images/
```

Uploading and Downloading

Uploading to the server (put)

sftp — uploading files

```
# Upload a single file to the current remote directory sftp> put index.html Uploading index.html to /var/www/html/index.html index.html 100% 48KB 240.0KB/s 00:00 # Upload to a specific remote path sftp> put style.css /var/www/html/css/style.css # Upload an entire directory recursively sftp> put -r images/ Uploading images/ to /var/www/html/images images/logo.png 100% 128KB 512.0KB/s 00:00 images/hero.jpg 100% 384KB 768.0KB/s 00:00 # Upload multiple files using a glob pattern sftp> put *.html
```

Downloading from the server (get)

sftp — downloading files

```
# Download a file to current local directory sftp> get backup.tar.gz Fetching /home/philip/backup.tar.gz to backup.tar.gz backup.tar.gz 100% 1.2GB 4.8MB/s 04:12 # Download to a specific local path sftp> get nginx.conf ~/backups/nginx.conf.bak # Download an entire directory recursively sftp> get -r /var/log/nginx/ ~/logs/nginx/ # Download multiple files with glob sftp> get /var/log/nginx/access.log.*
```

Managing Files and Directories

sftp — file and directory operations

```
# Create a directory on the remote server sftp> mkdir uploads sftp> mkdir -p archives/2026/june # Remove a remote file sftp> rm oldfile.html # Remove a remote directory (must be empty) sftp> rmdir old_uploads # Rename / move a remote file sftp> rename draft.html index.html # Check permissions and ownership sftp> ls -la # Change permissions on a remote file sftp> chmod 644 index.html sftp> chmod
```

```
755 scripts/deploy.sh # Create a local directory without leaving sftp sftp> lmkdir
~/downloads/server-backup # Run a local shell command with ! sftp> !ls -la ~/downloads/ sftp> !tar -
tzf backup.tar.gz | head
```

Full Command Reference

Command	Local equivalent	What it does	Example
pwd REMOTE	lpwd	Print current working directory	pwd
ls REMOTE	lls	List directory contents (accepts flags: -la, -lh)	ls -la /var/www
cd REMOTE	lcd	Change directory	cd /var/www/html
mkdir REMOTE	lmkdir	Create a directory	mkdir uploads
put LOCAL→REMOTE	–	Upload file(s). `-r` for recursive	put -r dist/
get REMOTE→LOCAL	–	Download file(s). `-r` for recursive	get -r /var/log/
rm REMOTE	–	Delete a remote file	rm old.html
rmdir REMOTE	–	Remove an empty remote directory	rmdir tmp
rename REMOTE	–	Rename or move a remote file	rename draft.html index.html
chmod REMOTE	–	Change remote file permissions	chmod 644 file.php
chown REMOTE	–	Change remote file ownership (numeric UID)	chown 1000 file.txt
df REMOTE	–	Show remote disk space usage	df
! LOCAL	–	Run a local shell command without leaving sftp	!ls ~/downloads
help	–	Print all available sftp commands	help
bye / quit / exit	–	Close the SFTP session	bye

Batch Mode — Non-Interactive Transfers

For scripted or automated transfers, SFTP can read commands from a file with `-b`, avoiding the need for an interactive session. Combine with `-q` to suppress progress output:

```
# Create a batch file — one sftp command per line
# ~/deploy.sftp
lcd ~/projects/website/dist
cd /var/www/html
put index.html
put style.css
```

```
put -r assets/
bye
```

philip@laptop — running an sftp batch file

```
# -b batch_file runs commands non-interactively philip@laptop:~$ sftp -b ~/deploy.sftp server
Uploading index.html to /var/www/html/index.html Uploading style.css to /var/www/html/style.css
Uploading assets/ to /var/www/html/assets # -q = quiet: suppress progress output, errors still show
philip@laptop:~$ sftp -q -b ~/deploy.sftp server philip@laptop:~$ ← silent unless something fails #
Embed in a shell script for a deploy pipeline philip@laptop:~$ npm run build && sftp -q -b
~/deploy.sftp server
```

> **Batch mode aborts on the first error.** If a `put` fails (file not found, permission denied), SFTP stops and exits with a non-zero code. For more resilient deploys where you want to continue after errors, consider `rsync` instead — covered in the next two chapters.

One-Shot Transfers Without an Interactive Session

You can use `scp` for simple one-file transfers where the interactive SFTP shell would be overkill. But SFTP can do the same thing non-interactively too:

```
# Upload a single file inline without entering the sftp shell
sftp server:/var/www/html/index.html <<< ""   ← download trick (rarely used)

# The common one-liner for a single upload: still easiest with scp
scp index.html server:/var/www/html/
scp -r dist/ server:/var/www/html/
scp -P 2222 file.txt server:~/   ← capital P for port, same as sftp
```

SFTP vs SCP — Which to Use

SFTP — USE WHEN

- You need to browse the remote filesystem
- Uploading multiple files interactively
- You need to rename, move, or delete remote files
- Batch mode scripted deploys
- GUI clients (FileZilla, WinSCP use SFTP)

SCP — USE WHEN

- Quick one-shot copy of a known file or directory
- You already know the exact remote path
- Copying between two remote machines
  scp server1:file.txt server2:~/
- Simple scripts where rsync would be overkill

> **For anything beyond a one-off copy, use rsync instead of either.** rsync only transfers what changed, can resume interrupted transfers, and is far more efficient for directories. SFTP and SCP transfer everything every time. Chapters 7 and 8 cover rsync in detail.

GUI Clients

If you prefer a drag-and-drop interface to the command line, several excellent SFTP GUI clients are available. All of them connect using the same credentials and key files as the `sftp` command.

FileZilla

WINDOWS · MACOS · LINUX — FREE

The most widely used SFTP client. Two-panel layout (local left, remote right) with drag-and-drop. Import your private key via Edit → Settings → Connection → SFTP → Add key file. Set Protocol to "SFTP – SSH File Transfer Protocol" in the Site Manager.

Tip: Use Site Manager (File menu) to save named connections — it remembers port, user, and key path.

WinSCP

WINDOWS ONLY — FREE & OPEN SOURCE

Windows-native SFTP/SCP client with deep Windows Explorer integration. Can save sessions, run scripts, and sync folders. Supports PuTTY .ppk key files natively — convert OpenSSH keys with PuTTYgen if needed.

Tip: WinSCP can open a PuTTY terminal in the same session via the Open Terminal button.

Cyberduck

WINDOWS · MACOS — FREE (DONATION)

Clean, minimal SFTP client that also supports S3, Google Drive, Backblaze, and more. Good choice if you work with multiple storage providers alongside SFTP. Integrates with the macOS keychain for key passphrase storage.

Tip: Right-click any bookmark → Edit → SSH Private Key to point it at your existing key file.

VS Code — Remote SSH

WINDOWS · MACOS · LINUX — FREE

Not a classic SFTP client, but the Remote-SSH extension lets you open a remote directory in VS Code and edit files directly on the server. Saves the upload/download step entirely for editing workflows. Uses your `~/.ssh/config` aliases automatically.

Tip: Install "Remote - SSH" from the Extensions panel. Press F1 → "Remote-SSH: Connect to Host" and pick your config alias.

Practical Workflow — Deploying a Website

philip@laptop — full deploy workflow with sftp

```
# 1. Build the project locally philip@laptop:~$ cd ~/projects/website && npm run build ✓ Build complete → dist/ # 2. Connect to the server with sftp philip@laptop:~$ sftp server Connected to server. # 3. Navigate remote and local directories sftp> cd /var/www/html sftp> lcd ~/projects/website/dist # 4. Upload changed files sftp> put index.html index.html 100% 12KB 240.0KB/s 00:00 sftp> put -r assets/ assets/style.css 100% 24KB 480.0KB/s 00:00 assets/app.js 100% 88KB 880.0KB/s 00:00 # 5. Verify permissions and exit sftp> ls -la sftp> bye philip@laptop:~$
```

> **Tab completion works in sftp** — press Tab to autocomplete remote paths and filenames just like in your regular shell. A lifesaver when navigating deeply nested directories.

Quick Reference

Task	Command
Connect using a config alias	`sftp server`
Connect to a non-standard port	`sftp -P 2222 user@host`
Upload a file	`put file.html`
Upload a directory recursively	`put -r dist/`
Download a file	`get backup.tar.gz`
Download a directory recursively	`get -r /var/log/nginx/`
Change remote directory	`cd /var/www/html`
Change local directory	`lcd ~/projects/dist`
Create remote directory	`mkdir uploads`
Delete remote file	`rm old.html`
Rename remote file	`rename draft.html index.html`
Run local command from sftp	`!ls ~/downloads`
Batch/scripted transfer	`sftp -b commands.sftp server`
Quick single-file copy (scp)	`scp file.txt server:~/`
Exit	`bye`

Next — Chapter 7: rsync Basics. SFTP transfers whole files every time. rsync is smarter — it compares source and destination and only sends what changed. Chapter 7 covers how rsync works, local directory syncing, and the essential flags you'll use in every rsync command: `-a`, `-v`, `-z`, `--delete`, and `--dry-run`.

CHAPTER 7 — RSYNC BASICS

Chapter 7 — rsync Basics

SFTP and SCP copy files. rsync *synchronises* them. The difference is efficiency: rsync compares the source and destination before transferring anything, then sends only the bytes that actually changed. Copy a 500 MB directory after editing one small file — SFTP sends 500 MB, rsync sends a few kilobytes.

This chapter covers how rsync works, local directory syncing, and the flags you'll reach for in almost every rsync command. Remote transfers over SSH come in Chapter 8.

How rsync Works

1. **Build a file list**
   rsync walks the source directory and records every file's name, size, modification time, and permissions.
2. **Compare with the destination**
   It does the same for the destination and compares the two lists. Files that match on size *and* modification time are considered up to date — no transfer needed.
3. **Block-level delta transfer**
   For files that differ, rsync doesn't send the whole file. It splits the destination file into fixed-size blocks, checksums each block, and only sends the blocks from the source that don't already exist at the destination.
4. **Reconstruct on the destination**
   The destination reassembles the file from existing blocks (unchanged parts) and the newly received blocks (changed parts). The result is a perfect copy of the source.
5. **Set metadata**
   Permissions, ownership, and timestamps are applied to match the source — assuming you passed the right flags (archive mode handles this automatically).

> **Result:** on a first run, rsync behaves like a full copy. On every subsequent run over the same source and destination, it does the minimum possible work — which is why it's the standard tool for backups and deploys.

Basic Syntax

```
# rsync [OPTIONS] SOURCE DESTINATION
rsync -av source/ destination/

# SOURCE and DESTINATION can be:
#   a local path:   /home/philip/docs/
#   a remote path:  server:/home/philip/docs/   (Chapter 8)
```

The Essential Flags

`-a`

`--archive`

The single most important flag. Enables recursive copy and preserves permissions, ownership, timestamps, symlinks, and device files. Equivalent to `-rlptgoD`. Use it on almost every rsync command.

`-v`

`--verbose`

Prints the name of each file transferred. Add a second `-v` (`-vv`) for even more detail. Essential when learning; drop it in cron jobs where the output goes to waste.

`-n`

`--dry-run`

Simulates the sync without actually transferring or deleting anything. Combine with `-v` to see exactly what *would* happen. Always run a dry-run before using `--delete` for the first time.

`-z`

`--compress`

Compresses data during transfer. Helpful on slow connections or for text-heavy files (HTML, CSS, JS, logs). Skip it for already-compressed formats like JPG, MP4, or zip — compression adds CPU overhead for no gain.

`--delete`

`(no short form)`

Deletes files at the destination that no longer exist at the source. What makes rsync a true mirror rather than just a copy. Without it, deleted source files accumulate at the destination forever.

`--progress`

`(no short form)`

Shows a per-file progress bar with transfer speed and time remaining. Useful for large files. For a single summary at the end instead, use `--info=progress2`.

`--exclude`

`(no short form)`

Skip files or directories matching a pattern. Accepts shell globs. Can be specified multiple times: `--exclude='*.log' --exclude='.git'`. Patterns match relative to the source root.

`-h`

`--human-readable`

Prints sizes in human-readable format (KB, MB, GB) rather than bytes. Combine with `--progress` or `--stats`.

The Trailing Slash Rule

rsync's most common source of confusion: whether you put a trailing slash on the *source* changes what gets synced.

WITH TRAILING SLASH — SYNC CONTENTS

`rsync -av source/ dest/` Copies the *contents* of source/ directly into dest/: dest/ ├── file1.txt ├── file2.txt └── subdir/

Think: "copy what's inside source"

WITHOUT TRAILING SLASH — SYNC DIRECTORY

`rsync -av source dest/` Copies the source *directory itself* inside dest/: dest/ └── source/ ├── file1.txt ├── file2.txt └── subdir/

Think: "copy the source folder"

> **The trailing slash only matters on the source, not the destination.** `rsync -av source/ dest` and `rsync -av source/ dest/` behave identically. The rule only applies to the left side.

Local Sync — Practical Examples

Mirror one directory to another

philip@debian — basic local sync

```
# Mirror ~/projects/website to ~/backups/website philip@debian:~$ rsync -av ~/projects/website/
~/backups/website/ sending incremental file list index.html style.css js/app.js images/logo.png sent
248,291 bytes received 89 bytes 496,760.00 bytes/sec total size is 247,820 speedup is 1.00
```

Second run — only changed files transfer

philip@debian — subsequent run after editing one file

```
# Only index.html was changed — only index.html transfers philip@debian:~$ rsync -av
~/projects/website/ ~/backups/website/ sending incremental file list index.html sent 2,341 bytes
received 35 bytes 4,752.00 bytes/sec total size is 247,820 speedup is 104.48 ↑ speedup factor: rsync
skipped 104× worth of data
```

Dry-run first — always before --delete

philip@debian — dry-run to preview changes

```
# -n = dry run. Shows exactly what WOULD happen without doing anything philip@debian:~$ rsync -avn -
-delete ~/projects/website/ ~/backups/website/ sending incremental file list index.html deleting
old-page.html sent 1,823 bytes received 27 bytes 3,700.00 bytes/sec total size is 247,820 speedup is
134.78 (DRY RUN) ← nothing was actually changed # Looks right — run for real philip@debian:~$ rsync
-av --delete ~/projects/website/ ~/backups/website/
```

> **--delete permanently removes files at the destination.** There is no recycle bin. If you accidentally flip source and destination, you could delete the files you meant to keep. Always `-n` dry-run first when using `--delete` until the pattern is muscle memory.

Reading rsync Output

With `-v` and `--itemize-changes` (`-i`), rsync prints an 11-character status code before each filename showing exactly what changed:

philip@debian — itemized output

```
philip@debian:~$ rsync -avi --delete source/ dest/ sending incremental file list >f+++++++++
newfile.txt ← new file, will be created >f.st...... index.html ← size and time changed .f.........
unchanged.css ← no change, skipped *deleting removed.html ← deleted from source, removed at dest
```

`>f` File transferred (sent to destination) `cd` Directory created `*d` Deletion (file or directory) `.f` File unchanged / skipped

Excluding Files and Directories

philip@debian — exclusion patterns

```
# Exclude a specific directory philip@debian:~$ rsync -av --exclude='.git' source/ dest/ # Exclude multiple patterns philip@debian:~$ rsync -av \ --exclude='.git' \ --exclude='node_modules/' \ --exclude='*.log' \ --exclude='*.tmp' \ source/ dest/ # Load exclusions from a file — one pattern per line philip@debian:~$ rsync -av --exclude-from='.rsyncignore' source/ dest/ # .rsyncignore file contents: .git node_modules/ *.log *.tmp .DS_Store
```

Patterns are matched against the path relative to the source root. `--exclude='logs/'` excludes any directory named `logs` anywhere in the tree. `--exclude='/logs/'` (with leading slash) only excludes a `logs` directory at the top level of the source.

Backup with Timestamps — a Simple Local Backup

A useful pattern: back up a directory to a timestamped folder so you keep a history of snapshots. rsync's `--backup` and `--backup-dir` flags move overwritten files to a separate location rather than just overwriting them.

philip@debian — timestamped backup

```
# Store changed/deleted files in a dated subfolder before overwriting philip@debian:~$ rsync -av \ --backup \ --backup-dir=~/backups/$(date +%Y-%m-%d) \ ~/projects/website/ \ ~/backups/current/ sending incremental file list index.html sent 2,341 bytes received 35 bytes 4,752.00 bytes/sec # Result: ~/backups/current/ has the latest version # ~/backups/2026-06-17/ has the previous version of index.html
```

Progress and Stats

philip@debian — watching progress

```
# --progress: per-file progress bar philip@debian:~$ rsync -avh --progress source/ dest/ large-video.mp4 384,000,000 100% 12.50MB/s 0:00:29 (xfer#3, to-check=0/12) # --info=progress2: single overall progress line (less noisy) philip@debian:~$ rsync -ah --info=progress2 source/ dest/ 248,291 100% 1.23MB/s 0:00:00 (xfer#4, to-check=0/12) # --stats: summary at the end philip@debian:~$ rsync -av --stats source/ dest/ Number of files: 12 (reg: 10, dir: 2) Number of created files: 0 Number of deleted files: 0 Number of regular files transferred: 1 Total file size: 247,820 bytes Total transferred file size: 2,341 bytes Literal data: 2,341 bytes Matched data: 0 bytes
```

Quick Reference

Command / flag	What it does
`rsync -av src/ dest/`	Archive + verbose — the standard starting point
`rsync -avn src/ dest/`	Dry run — preview without changing anything
`rsync -av --delete src/ dest/`	Mirror — delete at dest what's gone from src
`rsync -avz src/ dest/`	With compression — for slow links and text files
`rsync -avh --progress src/ dest/`	Human-readable sizes + per-file progress bar
`rsync -av --stats src/ dest/`	Summary stats at the end of the run
`--exclude='pattern'`	Skip files matching a glob pattern
`--exclude-from=file`	Load exclude patterns from a text file
`--backup --backup-dir=path`	Move overwritten files to a dated backup directory
`-i / --itemize-changes`	Show an 11-character status code for each file
`rsync --version`	Check installed rsync version
`rsync --help`	Full flag reference

Next — Chapter 8: rsync over SSH. Everything from this chapter applies directly to remote transfers — the only difference is the destination format. Chapter 8 covers push and pull syncs to a remote server, bandwidth limiting, the `--rsh` flag, excluding large directories, and practical one-liners for website deploys and server backups.

CHAPTER 8 — RSYNC OVER SSH

Chapter 8 — rsync over SSH

Everything from Chapter 7 applies here — the flags, the trailing-slash rule, the dry-run habit. The only difference is the destination format. Instead of a local path, you provide `user@host:/path` and rsync tunnels the transfer through SSH automatically. No extra configuration needed if you can already SSH into the machine.

Remote Path Syntax

```
rsync -avz ~/projects/website/ philip@server.example.com:/var/www/html/
```

flags | local source | user | hostname or IP | remote destination

> **Config aliases work here too.** If `server` is defined in `~/.ssh/config`, you can write `rsync -av src/ server:/var/www/html/` and rsync picks up the port, user, and key file from your config automatically.

Push vs Pull

PUSH — LOCAL → REMOTE

```
rsync -avz src/ server:/dest/
```

Run on your local machine. Sends files *to* the server. Used for deploys — pushing a built website or updated config to a server.

PULL — REMOTE → LOCAL

```
rsync -avz server:/src/ dest/
```

Run on your local machine. Fetches files *from* the server. Used for backups — pulling logs, databases, or a site snapshot to your machine.

Basic Push and Pull

philip@laptop — push to server

```
# Push local website build to the server's web root philip@laptop:~$ rsync -avz ~/projects/website/
server:/var/www/html/ sending incremental file list index.html css/style.css js/app.js sent 52,480
bytes received 131 bytes 21,044.40 bytes/sec total size is 247,820 speedup is 4.71 # Pull server
logs back to your machine for analysis philip@laptop:~$ rsync -avz server:/var/log/nginx/
```

```
~/logs/nginx/ sending incremental file list access.log error.log sent 1,823 bytes received 284,910
bytes 82,209.43 bytes/sec
```

Non-Standard Port — the --rsh Flag

When your SSH server runs on a port other than 22, you can't just add `-P` — that means something else to rsync. Instead, use `-e` (short for `--rsh`, remote shell) to pass the port to SSH directly:

```
# -e tells rsync which shell command to use for the connection
rsync -avz -e "ssh -p 2222" src/ server:/dest/

# With a specific key file too
rsync -avz -e "ssh -p 2222 -i ~/.ssh/id_work" src/ server:/dest/

# Or just use a config alias — zero extra flags needed
rsync -avz src/ work:/dest/   ← port and key come from ~/.ssh/config
```

> **The config alias approach is almost always cleaner.** Define the port, user, and key in `~/.ssh/config` once, and every tool that uses SSH — including rsync, sftp, and scp — inherits those settings from the alias.

Practical Scenarios

Deploy a website (with delete)

Mirror local build to server — removes files deleted from source

```
rsync -avz --delete \
  --exclude='.git' \
  --exclude='node_modules/' \
  ~/projects/site/dist/ \
  server:/var/www/html/
```

Pull a full server backup

Download home directory from server; skip large cache dirs

```
rsync -avz \
  --exclude='.cache/' \
  --exclude='tmp/' \
  server:/home/philip/ \
  ~/backups/server-home/
```

Sync a Raspberry Pi project

Push code to the Pi, skip virtual environments

```
rsync -avz \
  --exclude='venv/' \
  --exclude='__pycache__/' \
  --exclude='*.pyc' \
  ~/projects/myapp/ \
  pi:~/myapp/
```

Backup MySQL dumps

Pull latest database dumps from server to local machine

```
rsync -avz \
  server:/var/backups/mysql/ \
  ~/backups/mysql/
```

Preview before deploying

Dry-run to confirm exactly what will change — always a good habit

```
rsync -avzn --delete \
  ~/projects/site/dist/ \
  server:/var/www/html/
```

Sync between two remote servers

Run from your local machine; route through it as relay

```
rsync -avz \
  server1:/var/www/html/ \
  server2:/var/www/html/
```

Bandwidth Limiting

On a shared or metered connection, rsync can be told to cap its transfer rate so it doesn't swamp the link — useful for background backups running during working hours:

philip@laptop — capping transfer speed

```
# --bwlimit takes kilobytes per second (KB/s) philip@laptop:~$ rsync -avz --bwlimit=5000
server:/home/philip/ ~/backups/server/ sending incremental file list large-archive.tar.gz
512,000,000 100% 4.88MB/s 0:01:44 ← capped near 5 MB/s # Common values # --bwlimit=1000 ≈ 1 MB/s
(cautious background transfer) # --bwlimit=5000 ≈ 5 MB/s (moderate) # --bwlimit=50000 ≈ 50 MB/s
(fast LAN, still capped)
```

Resuming Interrupted Transfers

If a large transfer is interrupted (connection drop, power cut), rsync can pick up where it left off rather than starting from scratch — but only for files that were partially transferred:

```
# --partial: keep partially transferred files at the destination
#   so the next run can resume from where it stopped
rsync -avz --partial server:/backups/large-archive.tar.gz ~/backups/

# --partial-dir: store partial files in a hidden staging directory
#   cleaner than leaving half-transferred files in the destination
rsync -avz --partial-dir=.rsync-partial server:/backups/ ~/backups/

# -P is shorthand for --partial --progress together
rsync -avzP server:/backups/large-archive.tar.gz ~/backups/
```

> `-P` **is the flag to remember** for large remote transfers — it gives you a progress bar *and* enables resume on interruption in a single letter.

Preserving Permissions Across Users

When pushing files to a server where Apache or Nginx serves them, you may need to set permissions correctly rather than copying your local user's permissions. Two approaches:

```
# --no-perms --no-owner --no-group: don't copy metadata — let the
# server's umask and ownership apply (simplest for web deploys)
rsync -rltvz --no-perms --no-owner --no-group \
  ~/projects/site/dist/ server:/var/www/html/

# --chmod: force specific permissions regardless of source
rsync -avz --chmod=D755,F644 \
  ~/projects/site/dist/ server:/var/www/html/
# D755 = directories get 755, F644 = files get 644

# --chown: force ownership (requires root or sudo on the server)
rsync -avz --chown=www-data:www-data \
  ~/projects/site/dist/ server:/var/www/html/
```

A Complete Deploy Script

Putting it all together — a simple shell script that builds and deploys a website, with a dry-run mode and a production mode:

```
#!/bin/bash
# deploy.sh — build and rsync to server
# Usage: ./deploy.sh         (dry run)
#        ./deploy.sh --live  (real deploy)

SERVER="server"                    # ssh config alias
REMOTE_DIR="/var/www/html/"
LOCAL_DIR="$HOME/projects/website/dist/"

RSYNC_OPTS="-avz --delete
  --exclude='.git'
  --exclude='node_modules/'
  --exclude='*.map'"

# Build first
echo "Building..."
npm run build || { echo "Build failed"; exit 1; }

# Deploy
if [[ "$1" == "--live" ]]; then
  echo "Deploying to $SERVER..."
  rsync $RSYNC_OPTS "$LOCAL_DIR" "$SERVER:$REMOTE_DIR"
```

```
  echo "Done."
else
  echo "Dry run (pass --live to deploy):"
  rsync $RSYNC_OPTS --dry-run "$LOCAL_DIR" "$SERVER:$REMOTE_DIR"
fi
```

philip@laptop — using the deploy script

```
# Preview what would be deployed philip@laptop:~$ ./deploy.sh Building... ✓ Built in 3.4s Dry run (pass --live to deploy): index.html css/style.css (DRY RUN) # Happy with the preview — deploy for real philip@laptop:~$ ./deploy.sh --live Building... ✓ Built in 3.4s Deploying to server... index.html css/style.css Done.
```

Common Errors and Fixes

troubleshooting rsync over SSH

```
# Error: rsync: failed to connect — wrong host or SSH not listening rsync: [Receiver] failed to connect to server.example.com: Connection refused → check: ssh server (does plain SSH work?) # Error: permission denied on the remote path rsync: [sender] change_dir "/var/www/html" failed: Permission denied (13) → fix: ensure your user owns or has write access to the remote path ssh server "sudo chown philip:philip /var/www/html" # Error: rsync not found on the remote machine bash: rsync: command not found rsync: connection unexpectedly closed → fix: install rsync on the server ssh server "sudo apt install rsync" # Warning: skipping non-regular file — symlink in source skipping non-regular file "current" → add -l to preserve symlinks, or --copy-links to follow them # Error: port 22 blocked by firewall at work → use -e to connect via port 443 if sshd listens there rsync -avz -e "ssh -p 443" src/ server:/dest/
```

Quick Reference

Command / flag	What it does
`rsync -avz src/ server:/dest/`	Push local to remote (archive + verbose + compress)
`rsync -avz server:/src/ dest/`	Pull remote to local
`rsync -avzn src/ server:/dest/`	Dry run — preview without transferring
`rsync -avz --delete src/ server:/dest/`	Mirror — delete remote files that no longer exist locally
`-e "ssh -p 2222"`	Use non-standard SSH port
`-e "ssh -p 22 -i ~/.ssh/id_work"`	Specify SSH key inline
`--bwlimit=5000`	Cap transfer at 5 MB/s (value in KB/s)
`-P`	Progress bar + resume partial transfers

Command / flag	What it does
`--partial-dir=.rsync-partial`	Store partial files in a staging directory
`--chmod=D755,F644`	Force directory/file permissions at destination
`--no-perms --no-owner --no-group`	Don't copy metadata — let server's umask apply
`--exclude='node_modules/'`	Skip a directory by name
`--exclude-from=.rsyncignore`	Load exclude patterns from a file

Next — Chapter 9: Automating rsync. Running rsync by hand is fine for one-off transfers. For backups and deploys you want them to happen automatically — on a schedule, without prompts, and with some kind of rotation so you don't fill your disk. Chapter 9 covers cron jobs, unattended key-based auth, backup rotation strategies, and building a reliable home-server backup routine.

CHAPTER 9 — AUTOMATING RSYNC

Chapter 9 — Automating rsync

Running rsync by hand works for one-off transfers. For backups and deploys, you want them to happen automatically — on a schedule, without any human involvement, and with some kind of rotation so old snapshots don't fill your disk. This chapter ties everything together: cron scheduling, passphrase-free key auth for unattended runs, backup rotation strategies, and a complete home-server backup routine.

Passphrase-Free Keys for Unattended rsync

A cron job can't type a passphrase. For automated rsync over SSH, you need a dedicated key pair with **no passphrase** — used only for backup automation, with its permissions locked down on the server.

philip@debian — creating a passphrase-free backup key

```
# Generate a dedicated key — press Enter twice (no passphrase) philip@debian:~$ ssh-keygen -t
ed25519 -C "backup-cron" -f ~/.ssh/id_backup Generating public/private ed25519 key pair. Enter
passphrase (empty for no passphrase): ← press Enter Enter same passphrase again: ← press Enter again
Your identification has been saved in /home/philip/.ssh/id_backup Your public key has been saved in
/home/philip/.ssh/id_backup.pub # Copy it to the remote server philip@debian:~$ ssh-copy-id -i
~/.ssh/id_backup.pub server Number of key(s) added: 1 # Test — should connect without any prompt
philip@debian:~$ ssh -i ~/.ssh/id_backup server "echo ok" ok
```

A passphrase-free key is powerful — lock it down. On the server, in `~/.ssh/authorized_keys`, prefix the backup key entry with `command=` and `no-pty` restrictions so that even if the key is stolen, it can only run rsync — nothing else:

```
# ~/.ssh/authorized_keys on the SERVER
# The backup key is restricted: can only run rsync, cannot open a shell

command="rsync --server --sender -logDtpre.iLsfxCIvu . /",no-pty,no-agent-forwarding,no-port-forwa

# Your normal key sits below it without restrictions
ssh-ed25519 AAAA... philip@laptop
```

Separate keys for separate jobs. Your daily-use key has a strong passphrase and full shell access. The backup key has no passphrase and restricted access. If the backup key leaks, the attacker can only pull files — they can't get a shell,

> forward ports, or use your agent.

Adding the Backup Key to ~/.ssh/config

```
# ~/.ssh/config — add a dedicated stanza for automated backup

Host server-backup
    HostName         192.168.1.100
    User             philip
    IdentityFile     ~/.ssh/id_backup
    IdentitiesOnly   yes
    StrictHostKeyChecking yes   ← fail loudly if host key changes
    ConnectTimeout   10

# rsync commands in cron use server-backup as the alias
# rsync -az server-backup:/home/philip/ ~/backups/server/
```

Cron — Scheduling the Job

Cron runs commands on a schedule. Each line in your crontab is a rule specifying when to run a command:

```
30 2 * * * /home/philip/scripts/backup.sh >> /var/log/backup.log 2>&1
```

MIN	HOUR	DAY	MONTH	WEEKDAY	COMMAND TO RUN
0–59	0–23	1–31	1–12	0–7	

```
# Common cron schedule examples
30 2 * * *      /path/to/backup.sh    # every day at 02:30
0  3 * * 0      /path/to/backup.sh    # every Sunday at 03:00
0  1 1 * *      /path/to/backup.sh    # 1st of every month at 01:00
*/15 * * * *    /path/to/script.sh    # every 15 minutes
@daily          /path/to/backup.sh    # shorthand: once a day at midnight
@weekly         /path/to/backup.sh    # shorthand: once a week (Sunday)
@reboot         /path/to/startup.sh   # run once at system boot
```

philip@debian — editing the crontab

```
# Open your personal crontab for editing (uses $EDITOR — set it to vi) philip@debian:~$ crontab -e #
List current cron jobs philip@debian:~$ crontab -l 30 2 * * * /home/philip/scripts/backup.sh >>
/var/log/backup.log 2>&1 # Check cron ran (look for entries from cron daemon) philip@debian:~$ grep
```

```
CRON /var/log/syslog | tail -5 Jun 17 02:30:01 debian CRON[12345]: (philip) CMD
(/home/philip/scripts/backup.sh)
```

> **Cron has a minimal environment — full paths everywhere.** Cron doesn't load your `.bashrc` or `.profile`. The `PATH` is bare (`/usr/bin:/bin`). Always use absolute paths in cron scripts: `/usr/bin/rsync`, not just `rsync`. Find the path with `which rsync`.

Backup Rotation Strategies

A single rsync mirror is useful but fragile — if you accidentally delete a file and the next backup runs, the file is gone from both places. Rotation keeps multiple snapshots so you can go back in time.

STRATEGY 1

Dated snapshots

Each backup run creates a new timestamped directory. Simple but uses a lot of disk space — every file is duplicated each time, even unchanged ones.

STRATEGY 2

Hard-link rotation

Each snapshot directory hard-links unchanged files from the previous snapshot — they share the same inode on disk. Looks like a full copy but uses only the space of changed files.

STRATEGY 3

Grandfather-Father-Son

Keep daily backups for 7 days, weekly for 4 weeks, monthly for 12 months. Classic tape rotation scheme — adapted easily for rsync with a rotation script.

Hard-link rotation with --link-dest

rsync's `--link-dest` flag points to a previous snapshot. Files that haven't changed are hard-linked (not copied) — the new snapshot appears complete but uses almost no extra disk space:

```
# Create today's snapshot, hard-linking unchanged files from yesterday
TODAY=$(date +%Y-%m-%d)
YESTERDAY=$(date -d yesterday +%Y-%m-%d)
BACKUP_DIR="/mnt/backup"

rsync -az \
  --link-dest="$BACKUP_DIR/$YESTERDAY" \
  server-backup:/home/philip/ \
  "$BACKUP_DIR/$TODAY/"

# Result in /mnt/backup/:
#   2026-06-15/   full snapshot (first run — all files copied)
#   2026-06-16/   only changed files actually stored; rest are hard links
#   2026-06-17/   same — looks like a complete copy, costs almost nothing
```

A Complete Backup Script

This is a production-ready backup script that pulls a server's home directory to a local machine, uses hard-link rotation, and deletes snapshots older than 30 days:

```
#!/bin/bash
# /home/philip/scripts/backup.sh
# Pulls philip's home dir from server, keeps 30 days of snapshots

## ── Config ──────────────────────────────────────────────
SOURCE="server-backup:/home/philip/"   # ssh config alias + remote path
DEST="/mnt/backup/server"              # local backup root
KEEP_DAYS=30                           # delete snapshots older than this
LOG="/var/log/backup-server.log"

## ── Setup ───────────────────────────────────────────────
TODAY=$(date +%Y-%m-%d)
LATEST="$DEST/latest"                # symlink to most recent snapshot
SNAPSHOT="$DEST/$TODAY"

## ── Run ─────────────────────────────────────────────────
echo "[$(date)] Starting backup" >> "$LOG"

/usr/bin/rsync -az \
  --link-dest="$LATEST" \
  --exclude='.cache/' \
  --exclude='tmp/' \
  --exclude='*.log' \
  "$SOURCE" "$SNAPSHOT/" \
  >> "$LOG" 2>&1

if [ $? -eq 0 ]; then
  # Update the 'latest' symlink to point to today's snapshot
  ln -snf "$SNAPSHOT" "$LATEST"
  echo "[$(date)] Backup OK → $SNAPSHOT" >> "$LOG"
else
  echo "[$(date)] Backup FAILED" >> "$LOG"
fi

## ── Rotate ──────────────────────────────────────────────
# Delete snapshot directories older than $KEEP_DAYS days
find "$DEST" -maxdepth 1 -type d -name '????-??-??' \
  -mtime +"$KEEP_DAYS" -exec rm -rf {} \;

echo "[$(date)] Rotation complete" >> "$LOG"
```

philip@debian — installing the backup script and cron job

```
# Make the script executable philip@debian:~$ chmod +x ~/scripts/backup.sh # Test it manually first
— always do this before scheduling philip@debian:~$ ~/scripts/backup.sh philip@debian:~$ cat
/var/log/backup-server.log [2026-06-17 14:23:01] Starting backup [2026-06-17 14:23:08] Backup OK →
/mnt/backup/server/2026-06-17 [2026-06-17 14:23:08] Rotation complete # Schedule it: every day at
02:30 philip@debian:~$ crontab -e # Add this line: 30 2 * * * /home/philip/scripts/backup.sh #
```

```
Verify the snapshot structure philip@debian:~$ ls -la /mnt/backup/server/ drwxr-xr-x 2026-06-15/
drwxr-xr-x 2026-06-16/ drwxr-xr-x 2026-06-17/ lrwxrwxrwx latest -> /mnt/backup/server/2026-06-17
```

Verifying Backups Actually Work

A backup you've never tested is not a backup. Build verification into your routine:

philip@debian — spot-checking the backup

```
# Check the latest snapshot has recent files philip@debian:~$ ls -lt /mnt/backup/server/latest/ |
head -10 -rw-r--r-- 1 philip philip 2048 Jun 17 10:22 index.html -rw-r--r-- 1 philip philip 8192 Jun
17 09:14 notes.txt # Confirm hard-linking is working — inode count should be 2+ for unchanged files
philip@debian:~$ stat /mnt/backup/server/2026-06-16/notes.txt File: notes.txt Size: 8192 Blocks: 16
IO Block: 4096 Links: 3 ← hard-linked across 3 snapshots # Check how much disk space the backup
actually uses philip@debian:~$ du -sh /mnt/backup/server/ 1.2G /mnt/backup/server/ philip@debian:~$
du -sh /mnt/backup/server/*/ 1.1G /mnt/backup/server/2026-06-15/ ← first run: full copy 4.2M
/mnt/backup/server/2026-06-16/ ← only changed files 2.1M /mnt/backup/server/2026-06-17/ ← only
changed files
```

Getting Notified on Failure

A cron job that silently fails is worse than no backup at all. Three ways to get alerts:

```
# Option 1 — cron's built-in MAILTO (sends output by email on any output)
MAILTO=emubantam@gmail.com
30 2 * * * /home/philip/scripts/backup.sh

# Option 2 — send a notification only on failure (in the script)
if [ $? -ne 0 ]; then
  echo "Backup failed on $(hostname) at $(date)" | \
    mail -s "BACKUP FAILED" emubantam@gmail.com
fi

# Option 3 — use healthchecks.io (free service, pings a URL on success)
# The service alerts you if it doesn't receive a ping on schedule
30 2 * * * /home/philip/scripts/backup.sh && \
  curl -fsS --retry 3 https://hc-ping.com/YOUR-UUID > /dev/null
```

Automation Checklist

- ✓ Dedicated passphrase-free key for cron jobs, separate from your daily-use key

- ✓ Key restricted in `authorized_keys` with `command=` and `no-pty`
- ✓ SSH config alias pointing to the backup key (`IdentitiesOnly yes`)
- ✓ Absolute paths in the script (`/usr/bin/rsync` not `rsync`)
- ✓ Script tested manually before scheduling — check the log output
- ✓ Exit code checked in the script (`if [ $? -eq 0 ]`)
- ✓ Rotation in place — old snapshots deleted automatically
- ✓ Failure notification configured
- ✓ Periodic manual spot-check that recent files are actually in the snapshot
- ✓ Disk space monitored — backups silently stop when the disk is full

Quick Reference

Command / concept	What it does
`ssh-keygen -f ~/.ssh/id_backup`	Generate a dedicated backup key (no passphrase)
`ssh-copy-id -i id_backup.pub server`	Install the backup key on the server
`crontab -e`	Edit your personal crontab
`crontab -l`	List current cron jobs
`30 2 * * *`	Run daily at 02:30
`--link-dest=/path/to/prev`	Hard-link unchanged files from a previous snapshot
`ln -snf new latest`	Update the "latest" symlink to the newest snapshot
`find … -mtime +30 -exec rm -rf {} \;`	Delete directories older than 30 days
`du -sh snapshots/*/`	Show disk usage per snapshot (hard links make this tiny)
`stat file`	Show inode link count — confirms hard-linking is working
`grep CRON /var/log/syslog`	Check cron ran the job

Course Complete — SSH, SFTP & rsync

All 9 chapters finished. From TCP handshakes to automated nightly backups.

Ch 1 — How SSH Works · Ch 2 — Connecting · Ch 3 — Key-Based Auth · Ch 4 — SSH Config File · Ch 5 — Port Forwarding · Ch 6 — SFTP · Ch 7 — rsync Basics · Ch 8 — rsync over SSH

Ch 9 — Automating rsync

Courses 2 and 3 continue the remote access series: Remote Desktop (RDP & VNC) and VPN.