



LINUX COURSE

Debian Development Machine Setup

Ten chapters on setting up a Debian 13 development desktop:
installation, graphics drivers, Python, Node.js, PHP and databases,
editors, a self-hosted Git server, and a reproducible rebuild.

10 Chapters

Philip Osztromok

Generated with Claude

Contents

Chapter 1	Planning the Machine & Installing Debian 13	2
Chapter 2	First Boot: Updates, sudo & the Essentials	10
Chapter 3	Knowing What's Installed	17
Chapter 4	The Graphics Card & Drivers	24
Chapter 5	Python the Debian Way	33
Chapter 6	Node.js & npm	41
Chapter 7	PHP, Composer & Databases	48
Chapter 8	Editors & IDEs on the Desktop	56
Chapter 9	Self-Hosting Git with Gitea	63
Chapter 10	Capstone: A Reproducible Dev Machine	71

Debian Development Machine Setup

Chapter 1 · Planning the Machine & Installing Debian 13

A development machine is different from the server you deploy to. It is where you install things, break things, try three versions of a tool, and put it all back. So it is worth planning it properly once, and writing down what you did, rather than letting it grow by accident. This course builds one such machine from an empty disk to a documented, repeatable setup. In this first chapter we choose the operating system, check the hardware, and get Debian installed.

TWO MACHINES, ONE LETTER APART

The machine we build is called `devserver`. Your existing web server is `debserver`. The names differ by a single letter, so check the prompt before you run anything that changes a system. Chapter 2 shows how to make the shell prompt say which machine you are on.

Why Debian 13 (“Trixie”)

Debian 13, codenamed **Trixie**, was released on **9 August 2025**. It ships with the Linux 6.12 LTS kernel and offers GNOME 48, KDE Plasma 6.3, Xfce 4.20, LXQt 2.1 and LXDE as desktop choices. At the time of writing the current point release is 13.7 (12 September 2026).

	DEBIAN 12 (BOOKWORM)	DEBIAN 13 (TRIXIE)
Released	June 2023	9 August 2025
Full support ends	11 July 2026 (already over)	9 August 2028
Long Term Support until	30 June 2028	30 June 2030
32-bit PC (i386)	Installable	No installer, no kernel

Bookworm is still safe to run, since Long Term Support keeps supplying security fixes until mid-2028, but a brand-new machine should start on the current release. Trixie gives you newer toolchains, which matters on a development box, and it keeps you in the Debian family, so everything you know from Setting Up a Web Server on Debian and Linux Package Managers 2 (APT in Depth) applies directly.

32-BIT IS GONE

Trixie has no installer or kernel for 32-bit x86. The i386 architecture survives only as a compatibility layer on a 64-bit system, so the new machine must be 64-bit (`amd64`). The Debian release notes advise anyone running a 32-bit system to reinstall as amd64 or retire the hardware instead of upgrading.

Checking the Hardware Before You Start

Debian's stated minimum for a desktop is modest (about 1 GB of RAM, with 2 GB recommended, and 10 GB of disk), and your machine is far above it. The questions that matter here are practical ones. Go through this list before you touch the installer.

CHECK	WHAT TO FIND OUT
Graphics card fit	The case is low-height. Measure the slot: card height (is it low-profile?), length, and cooler thickness. A 6 GB card may also need extra power connectors and a power supply with enough capacity, and a small work desktop often has neither.
RAM	64 GB of DDR4: confirm in the BIOS/UEFI screen that all of it is detected and that the sticks are in the right slots.
Storage	How big is the drive, and is it an SSD? Development means many small files, so an SSD makes a large difference. Note the size; you will need it for the partition plan below.
Network	A wired connection during installation is the simplest way to fetch packages and updates.
Boot mode	Is the firmware UEFI or legacy BIOS, and is Secure Boot switched on? Modern machines are usually UEFI.
Work ownership	The machine came from work. Ask whether it has a BIOS/UEFI password, management software, or a policy about wiping and reusing it, before you erase anything.

INSTALLING OVER AN EXISTING SYSTEM ERASES IT

Guided partitioning can wipe the whole disk. If the machine has anything on it that you might want, copy it somewhere safe first. Once the disk is repartitioned it cannot be recovered.

Choosing an Installation Image

Debian offers a few kinds of image for amd64. The two you are most likely to choose between are:

- **Netinst** — a small image that installs the base system and downloads the rest over the network. It needs a working internet connection during installation and always installs current packages.
- **DVD** — a large image containing many packages, so it can install without a network.

Official images have included the non-free firmware that some hardware needs since Debian 12. During installation Debian's installer checks which firmware your hardware wants and configures the `non-free-firmware` repository component when it is needed, so a Wi-Fi card or graphics chip is less likely to leave you stuck on a black screen than in earlier releases. Chapter 4 returns to the graphics card, which is a separate matter: the proprietary NVIDIA driver is not part of that firmware package set.

Which desktop?

The installer lets you choose a desktop environment. **GNOME** is the default and a fine choice. **KDE Plasma** suits people who like to configure things. **Xfce** is lighter, but with 64 GB of RAM that is not a concern for you. The rest of this course works on any of them, because the tools we install run the same regardless. Pick the one you would be happy to look at all day.

Verifying and Writing the Installer

Download the netinst image for amd64 from [debian.org](https://deb.debian.org/debian/cd/images/netinst), together with the `SHA512SUMS` and `SHA512SUMS.sign` files from the same directory. Then check the image before you use it. A corrupted or tampered installer is a bad way to start.

```
# 1. Check the file was not corrupted in transit
sha512sum -c SHA512SUMS --ignore-missing

# 2. Check the checksum file really came from Debian
# (import the signing key listed on debian.org/CD/verify first)
gpg --verify SHA512SUMS.sign SHA512SUMS
```

The first command confirms the image matches its listed checksum. The second confirms the checksum file carries a genuine Debian signature. Compare the key fingerprint gpg reports with

the fingerprints published on Debian's verification page.

Debian's images are *hybrid*, which means they can be copied straight onto a USB stick and booted. On Linux, find the stick with `lsblk`, then write the image to the **whole device**.

```
# Run lsblk before and after plugging the stick in; the new device is the stick
lsblk

# Replace sdX with your stick (for example sdb), NOT a partition like sdb1
sudo dd if=debian-13.7.0-amd64-netinst.iso of=/dev/sdX bs=4M status=progress conv=fsync
sync
```

DD WILL OVERWRITE THE WRONG DISK WITHOUT ASKING

`dd` does exactly what it is told. If `of=` points at your system disk instead of the USB stick, the disk is overwritten immediately. Read the device name twice, compare its size in `lsblk` with the stick you expect, and never write to a partition such as `sdb1`. Do not use tools such as UNetbootin, which modify the image instead of copying it. On Windows, use a tool that writes the ISO in raw ("DD image") mode.

The version number in the file name changes with each point release, so use whatever the current netinst file is called on the download page.

Planning the Partitions

Debian's guided partitioning offers several schemes: everything in one partition, a separate `/home`, or separate `/home`, `/var` and `/tmp`. It can also build an encrypted LVM setup, which puts everything except a small `/boot` inside an encrypted volume. For a development machine, a separate `/home` is the option worth taking. It keeps your projects, settings and downloads apart from the system, so you can reinstall or upgrade the operating system later without touching your work.

PARTITION	SUGGESTED SIZE	WHY
EFI System Partition	512 MB – 1 GB	Needed for booting a UEFI machine; the installer creates it for you.

PARTITION	SUGGESTED SIZE	WHY
<code>/</code> (ext4)	60 – 100 GB	The system, plus packages, tools and language runtimes you will install.
swap	8 – 16 GB	With 64 GB of RAM you will rarely swap. Make it at least as large as RAM only if you want hibernation.
<code>/home</code> (ext4)	Everything left	Projects, repositories, downloads, settings.

These sizes are this course's suggestion, not Debian's rule. Adjust them to your disk. Encrypting the disk is worth considering if the machine could be lost or stolen, or if company policy requires it. Ask when you check the work-ownership question above, because an encrypted disk cannot be recovered if you forget the passphrase.

/TMP LIVES IN RAM NOW

In Trixie the default is for `/tmp` to be a `tmpfs` filesystem held in memory, using up to 50% of RAM by default. On this machine that is a large, fast scratch area, which suits builds and compilers. It also means very large temporary files are stored in memory and vanish on reboot, and you can change the limit with `systemctl edit tmp.mount`.

Running the Installer

Boot from the USB stick (the boot menu key varies by manufacturer; it is often F12, F10 or Esc during start-up). Choose the graphical or text installer, then work through the questions. The choices that matter for this machine are:

1. **Language, location and keyboard** — choose these to match the physical keyboard you actually have.
2. **Hostname** — `devserver`. Leave the domain name blank unless you have one.
3. **Root password** — leave it empty. Debian then disables the root account and makes your first user an administrator who can use `sudo`, which is the safer everyday setup and the one this course assumes.
4. **Your user account** — a strong password; this is the account you will live in.
5. **Partitioning** — pick manual or guided with a separate `/home`, and review the summary carefully before you confirm the write to disk.

6. **Mirror** — choose a mirror in your own country for faster downloads.
7. **Software selection** — tick your chosen desktop environment and “standard system utilities”. It is worth also ticking “SSH server” so you can log in remotely (Chapter 2 revisits this). Leave the web server option off; you are not building the web server here.

TAKE NOTES AS YOU GO

Write down every non-default choice as you make it: the partition sizes, the desktop, the package groups. Chapter 10 turns these notes into a script that rebuilds the machine, and it is much easier to script something you recorded than something you have to remember.

First Checks After the Reboot

When the machine restarts into the installed system, confirm that what you built is what you meant to build. Open a terminal and run:

```
# Which release is this?
cat /etc/os-release

# 64-bit hardware and 64-bit packages? Expect x86_64 and amd64
uname -m
dpkg --print-architecture

# Is all the RAM visible? Expect about 64G
free -h

# Disks and mounted partitions
lsblk

# Is the graphics card detected at all?
lspci | grep -i vga
```

If `lspci` is not found, install it with `sudo apt install pciutils`. It is included in the standard system utilities on many installs but not all. If `free -h` shows much less than 64 GB, or `lspci` does not list a graphics card, stop here and sort it out; a stick loose in its slot is more common than a hardware fault. Everything after this chapter assumes the hardware is behaving.

Hands-On Exercises

EXERCISE 1

Write a one-page hardware and installation plan for `devserver`. Include: the measurements or checks you will do for the graphics card, the disk size you found, whether the firmware is UEFI, the answer to the work-ownership question, your chosen desktop and why, and the exact partition table you will create.

 [View solution](#)

EXERCISE 2

Practise verification without needing a Debian download. Create a small file, generate its checksum into a file with `sha512sum`, verify it with `sha512sum -c`, then change one character in the file and verify again. Explain what each result proves, and what the `gpg --verify` step adds that a checksum alone cannot.

 [View solution](#)

EXERCISE 3

A 1 TB SSD will hold `devserver`. Design the partition table (sizes and filesystems) and justify each choice. Then explain what Trixie's in-memory `/tmp` means for someone compiling a large project on this machine, and what you would do if a build ran out of space there.

 [View solution](#)

CHAPTER 1 QUICK REFERENCE

- **Debian 13 (Trixie)** was released 9 August 2025; full support runs to 9 August 2028 and Long Term Support to 30 June 2030
- Debian 12 (Bookworm) full support ended 11 July 2026; LTS continues until 30 June 2028
- Trixie has **no 32-bit x86 installer or kernel**; the new machine must be amd64
- Official images include non-free firmware since Debian 12; the NVIDIA driver is a separate matter (Chapter 4)

- Verify the download: `sha512sum -c SHA512SUMS --ignore-missing` then `gpg --verify SHA512SUMS.sign SHA512SUMS`
- Write the image to the whole device (`/dev/sdX` , not a partition) with `dd ... conv=fsync` , then `sync`
- A separate `/home` lets you reinstall the system without touching your work
- Trixie mounts `/tmp` as `tmpfs` in RAM, up to 50% of RAM by default
- Leave the root password empty in the installer to get a `sudo` -enabled first user
- After installing, check `/etc/os-release` , `uname -m` , `dpkg --print-architecture` , `free -h` , `lsblk` and `lspci`

Debian Development Machine Setup

Chapter 2 · First Boot: Updates, sudo & the Essentials

`devserver` has just booted into Debian 13 for the first time. Before you install any development tools, there is a short list of jobs worth doing once, in order: understand where your packages come from, bring the system up to date, check that `sudo` works, install a small set of tools every development machine needs, make the shell prompt tell you which machine you are on, and confirm you can log in remotely. None of it is glamorous, but everything in the rest of the course sits on top of it.

Where Your Packages Come From

On Debian 13, apt reads its package sources from the newer **deb822** format: files ending in `.sources` in `/etc/apt/sources.list.d/`, where each source is a stanza of `Key: value` lines. The old one-line-per-repository `/etc/apt/sources.list` format still works, but it is no longer what a fresh Trixie install uses. Have a look at yours:

```
ls /etc/apt/sources.list.d/
cat /etc/apt/sources.list.d/debian.sources
```

You should see something like this (yours will show the mirror you chose in the installer, and the exact lines can differ slightly):

```
Types: deb
URIs: https://deb.debian.org/debian
Suites: trixie trixie-updates
Components: main non-free-firmware
Signed-By: /usr/share/keyrings/debian-archive-keyring.gpg

Types: deb
URIs: https://security.debian.org/debian-security
Suites: trixie-security
Components: main non-free-firmware
Signed-By: /usr/share/keyrings/debian-archive-keyring.gpg
```

FIELD	WHAT IT MEANS
Types	<code>deb</code> is binary packages. <code>deb-src</code> would add source packages, which you only need if you plan to build Debian packages from source.
URIs	The mirror to download from. The second stanza is Debian's separate security archive.
Suites	<code>trixie</code> is the release itself. <code>trixie-updates</code> carries fixes that are not security-related but should not wait for the next point release. <code>trixie-security</code> carries security fixes.
Components	<code>main</code> is Debian's free software. <code>non-free-firmware</code> holds firmware for hardware such as network cards. <code>contrib</code> and <code>non-free</code> exist too but are not enabled here.
Signed-By	The keyring apt uses to check the packages really came from Debian.

OLD-FORMAT FILES

If you ever meet an old `.list` file, from a tutorial or a third-party installer, apt can convert it: `sudo apt modernize-sources` rewrites `.list` files to the `.sources` format and asks before it changes anything. Linux Package Managers 3 (Managing Repositories on Debian-Based Systems) covers repositories in more depth, though it predates the move to the newer format.

DON'T ENABLE NON-FREE YET

Chapter 4 needs the `non-free` component for the NVIDIA driver. Leave it off for now, and add it at that point, when there is a reason to.

Updating the System

Two apt commands do the work, and they do different jobs. `apt update` downloads the current list of available packages, and installs nothing. `apt full-upgrade` then actually installs newer versions. Always run them in that order.

```
sudo apt update
sudo apt full-upgrade
```

Why `full-upgrade` and not `upgrade`? `upgrade` only installs upgrades that do not need any existing package removed. `full-upgrade` does the same, and will also remove installed packages if that is needed to upgrade the system as a whole. On a stable release those removals

are rare and apt shows you exactly what it will do before asking you to confirm, so `full-upgrade` is the safer habit: it never leaves you half-updated. Read the summary before you type `y`.

If a new kernel was installed, restart the machine so it is the one running. You can compare what is running with what is installed:

```
# The kernel you are running now
uname -r

# The kernel packages that are installed
dpkg -l 'linux-image*' | grep ^ii
```

sudo: Your Administrator Account

In Chapter 1 you left the root password empty in the installer. Debian responded by disabling the root account and adding your first user to the `sudo` group. Check that this happened:

```
groups
sudo whoami
```

`groups` should list `sudo`, and `sudo whoami` (after your password) should print `root`. That single line is the whole idea: you are an ordinary user who borrows administrator rights for one command at a time, instead of staying logged in as an all-powerful account.

EDIT SUDO RULES ONLY WITH VISUDO

If you ever need to change who can use sudo, use `sudo visudo` (or a file under `/etc/sudoers.d/` edited with `sudo visudo -f`). It checks the syntax before saving. A typo in a plain-text edit of the sudoers file can lock you out of administrator access completely. Do not add `NOPASSWD` rules to save typing; a password prompt is a very cheap safety net.

A Starter Set of Tools

A fresh desktop install is deliberately small. These tools are worth installing right away because you will reach for them constantly, and later chapters assume they are present:

```
sudo apt install build-essential curl wget git vim htop tree unzip zip jq ripgrep \
ca-certificates gnupg pciutils
```

PACKAGE	WHY YOU WANT IT
build-essential	A C/C++ compiler and <code>make</code> . Many language packages compile small pieces of C when they install.
curl, wget	Download files and talk to web services from the command line.
git	Version control. Chapter 9 builds a Git server; you need the client before then.
vim	A proper editor for the terminal. The default <code>vi</code> is a cut-down cousin.
htop	A readable view of what the machine is doing.
tree, unzip, zip	Show directory structure; open and create archives.
jq, ripgrep	<code>jq</code> reads and filters JSON. <code>rg</code> is a very fast search through files.
ca-certificates, gnupg	Trusted certificates for HTTPS, and GPG for verifying keys. Later chapters add third-party repositories, which need both.
pciutils	Provides <code>lspci</code> , which Chapter 1 used to find the graphics card.

Some of these may already be installed. If so, apt tells you so and installs nothing extra. Chapter 3 shows how to check what is already on the system before you install, so you are not guessing.

A Prompt That Names the Machine

Debian's default bash prompt already shows `user@hostname`, but on two machines that look alike a plain prompt is easy to skim past. The fix is to give each machine its own colour, so a glance at the terminal tells you where you are. Confirm the hostname first:

```
hostnamectl
```

If it does not say `devserver`, set it with `sudo hostnamectl set-hostname devserver` and log out and in again. Then add this to the end of `~/.bashrc` on `devserver`:

```
# Prompt: user@host in bold cyan on devserver, working directory in blue
PS1='\[\e[1;36m\]\u@\h\[\e[0m\]:\[\e[1;34m\]\w\[\e[0m\]\$ '
```

On the web server, use a colour that means *be careful*. Red (`1;31`) does the job:

```
# On debserver: user@host in bold red
PS1='\[\e[1;31m\]\u@\h\[\e[0m\]:\[\e[1;34m\]\w\[\e[0m\]\$ '
```

Open a new terminal (or run `source ~/.bashrc`) to see the change. The `\[ ... \]` pairs tell bash the colour codes take no screen space, which stops long commands from wrapping strangely.

Logging In Remotely with SSH

You ticked “SSH server” in the installer, so `devserver` is already running the OpenSSH server. Trixie ships OpenSSH 10.0. On Debian the service is called `ssh`:

```
systemctl status ssh

# The machine's addresses; use one of these from another computer
ip -br address
```

From another computer on your network, connect with `ssh yourname@THE-ADDRESS`. Typing an address each time gets old fast, but the address is a reliable thing to fall back on, because a machine name only works if your network can resolve it.

Log in with a key, not a password

A password can be guessed. A key pair cannot in any practical sense. Generate one on the computer you will connect *from*, and copy the public half to `devserver`:

```
# On the computer you connect FROM
ssh-keygen -t ed25519
ssh-copy-id yourname@devserver-address
```

```
# Check that the key login works before changing anything else
ssh yourname@devserver-address
```

Once key login works, you can switch password logins off on `devserver`. Debian's SSH server reads extra settings from files in `/etc/ssh/sshd_config.d/`, so you do not need to edit the main file. Create `/etc/ssh/sshd_config.d/10-no-passwords.conf` containing:

```
PasswordAuthentication no
```

```
# Test the configuration for mistakes BEFORE applying it
sudo sshd -t

# Apply it without dropping existing connections
sudo systemctl reload ssh
```

KEEP A SECOND DOOR OPEN

Never switch off password logins until you have proved key login works from a *second* terminal, and keep the first session logged in while you test. If the change goes wrong you can still fix it from the open session. If you close everything and are locked out, you need physical access to the machine, which is manageable on your desk but not on a remote server. This course only covers the basics; the Remote Access With SSH course goes much further.

Hands-On Exercises

EXERCISE 1

Read your own `/etc/apt/sources.list.d/debian.sources`. For each stanza, explain what every field does. Then run `apt policy` and identify which of the archive lines correspond to the release, the updates suite and the security suite.

 [View solution](#)

EXERCISE 2

Before installing the starter tools, record which of them are already installed. Install the full list, then report which ones apt actually had to install and how you can tell. Finish by printing the version of each command-line tool.

 [View solution](#)

EXERCISE 3

Set up the coloured prompt on `devserver`, generate an ed25519 key on another computer, install it, and log in with it. Then write out the exact order of steps you would follow to disable password logins safely, including how you would undo the change if it went wrong.

 [View solution](#)

CHAPTER 2 QUICK REFERENCE

- Debian 13 keeps apt sources in `/etc/apt/sources.list.d/*.sources` (deb822 format); `apt modernize-sources` converts old `.list` files
- Stanza fields: **Types**, **URLs**, **Suites** (trixie, trixie-updates, trixie-security), **Components** (main, non-free-firmware), **Signed-By**
- `sudo apt update` refreshes the package list; `sudo apt full-upgrade` installs upgrades and may remove packages if needed
- Check `groups` for `sudo`; edit sudo rules only with `visudo`; avoid `NOPASSWD`
- Starter tools: `build-essential curl wget git vim htop tree unzip zip jq ripgrep ca-certificates gnupg pciutils`
- `hostnamectl` shows the hostname; a different `PS1` colour per machine prevents running commands on the wrong one
- The SSH service is `ssh` on Debian: `systemctl status ssh`
- Use `ssh-keygen -t ed25519` and `ssh-copy-id`; put server settings in `/etc/ssh/sshd_config.d/`
- Always `sudo sshd -t` before reloading, and keep a second session open while changing SSH settings

Debian Development Machine Setup

Chapter 3 · Knowing What's Installed

Most of the trouble on a development machine comes from a simple gap: you think a tool is installed, or think it is a particular version, and you are wrong. You install it a second time by a different route, and now two copies fight over which one runs. This chapter builds a habit that prevents that. Before you install anything in the rest of this course, you ask three questions.

QUESTION	TOOLS THAT ANSWER IT
1. Is it installed?	<code>dpkg -s</code> , <code>dpkg-query -W</code> , <code>command -v</code>
2. Which version?	<code>apt policy</code> , <code>dpkg-query -W</code> , <code>tool --version</code>
3. Where did it come from?	<code>dpkg -S</code> , <code>type -a</code> , <code>apt-mark showmanual</code>

Asking the Package System

Debian has two layers: `dpkg` is the low-level tool that knows what is installed, and `apt` sits on top of it, knows about the repositories, and works out dependencies. Linux Package Managers 2 (APT in Depth) and Linux Package Managers 4 (dpkg Deep Dive) cover both in detail. Here we only need the commands that answer questions.

Is this one package installed?

```
# Full status entry for one package
dpkg -s git

# Just the facts, in a format you choose
dpkg-query -W -f='${Package} ${Version} ${db:Status-Abbrev}\n' git
```

`dpkg -s git` prints a block that includes the line `Status: install ok installed` when the package is present. The `dpkg-query -W -f=...` form lets you print exactly the fields you want, so the answer is one short line: the package name, its version string, and `ii`. If the package is not installed, `dpkg -s` says so and returns a non-zero exit status, which matters when you put the check into a script.

Searching the list of installed packages

```
# List packages matching a pattern (installed or not)
dpkg -l 'python3*'

# Everything installed, from apt's point of view
apt list --installed 2>/dev/null | grep -i python

# What could be upgraded right now?
apt list --upgradable
```

Look at the first three characters of each line of `dpkg -l`. They are status letters: the first is what you asked for, the second is the package's actual state, and the third flags errors. The two you will meet most are:

SHOWN AS	MEANING
<code>ii</code>	Installed, and meant to be.
<code>rc</code>	Removed, but its configuration files were left behind. Not installed, even though it is listed.

LISTED DOES NOT MEAN INSTALLED

`dpkg -l 'python3*'` lists packages dpkg has any record of, including ones with status `rc`, and the pattern can also match the names of packages that are not on your machine. Always read the status letters, or use `dpkg -s`, which answers the exact question you asked.

PIPING APT'S OUTPUT

When you pipe `apt list` into another command, apt prints a warning that it does not have a stable command-line interface for scripts. That is why the example above adds `2>/dev/null`, which hides the warning. Apt's own advice is that scripts should use `dpkg-query` or `apt-get` instead; both are stable.

Which version is installed, and which is available?

```
apt policy git
apt show git
```

`apt policy` shows two lines that answer the version question directly: **Installed** (what you have, or `(none)`) and **Candidate** (what `apt install` would give you). If they differ, an upgrade is available. Below them it lists which repository each version comes from. `apt show` adds the description, dependencies and size.

Asking the Shell

The package system answers questions about *packages*. But what you actually type is a *command*, and the two names are not always the same. The shell can tell you whether a command exists and where it lives:

```
# Is the command available? (prints its path, or nothing)
command -v rg
command -v gcc

# Show EVERY copy of a command on the PATH, in the order the shell would find them
type -a python3

# What version is it?
rg --version
gcc --version
```

`command -v` is the reliable check: it prints the path when the command exists and prints nothing (with a non-zero exit status) when it does not. `type -a` is the one that catches the classic two-copies problem, because it lists every match on your `PATH`, and the first one listed is the one that runs.

THE PACKAGE NAME IS NOT ALWAYS THE COMMAND NAME

Several of the tools you installed in Chapter 2 do not share a name with their command: `ripgrep` gives you `rg`, `pciutils` gives you `lspci`, and `build-essential` gives you `gcc` and `make` but no command called `build-essential`. So `command -v ripgrep` finds nothing even though the package is installed. Ask the package system about package names, and the shell about command names.

From a file to its package, and back

Two tools connect the two views. Given a file, `dpkg -S` tells you which installed package owns it. Given a package, `dpkg -L` lists the files it installed:

```
# Which package owns this command?
dpkg -S "$(command -v lspci)"

# What did a package install?
dpkg -L ripgrep
```

`dpkg -S` only knows about *installed* packages. To find which package would provide a file you do not have yet, use `apt-file`. It downloads an index of every file in every package, so it needs one extra install and one extra update before it works:

```
sudo apt install apt-file
sudo apt-file update

# Which package contains a file called pg_config?
apt-file search pg_config
```

This is the tool to reach for when a build fails with “missing header file” or “command not found”. It tells you which package to install to fix it.

Not Everything Comes from apt

Here is the part that catches people. On a development machine, a lot of software never touches apt. Python packages come from pip, JavaScript tools from npm, and plenty of programs are just downloaded and dropped into a directory. The package system knows nothing about those, so

`dpkg -S` is a useful test:

```
# A file that apt installed
dpkg -S /usr/bin/git
# git: /usr/bin/git

# A file that something else put there (a made-up example)
dpkg -S /usr/local/bin/mytool
# dpkg-query: no path found matching pattern /usr/local/bin/mytool
```

"No path found matching pattern" is the tell: nothing managed by apt owns that file. Where a file lives is also a clue:

LOCATION	USUALLY MEANS
<code>/usr/bin</code>	A Debian package installed it.
<code>/usr/local/bin</code>	Installed by hand or by a script, outside apt.
<code>~/.local/bin</code>	Installed for your user only: often pip, pipx, or a downloaded tool.
Somewhere under your home directory	A version manager or a per-user tool, such as one you install in Chapters 5 and 6.

TWO COPIES, ONE WINNER

If `type -a python3` lists two paths, the shell runs the first. That may not be the one you meant to update. Chapter 5 (Python) and Chapter 6 (Node.js) return to this, because version managers work by putting their own copy first on the `PATH` on purpose.

Which Packages Did You Ask For?

When you run `apt install git`, apt also installs whatever git needs. Debian remembers the difference. Packages you asked for are marked **manual**, and packages pulled in as dependencies are marked **automatic**. An automatic package that nothing needs any more can be removed with `apt autoremove`.

```
# The packages that were installed on purpose
apt-mark showmanual

# How many?
apt-mark showmanual | wc -l
```

The manual list is the honest record of what you chose to put on the machine, which makes it the right starting point for rebuilding it. Chapter 10 exports it for exactly that purpose.

Putting It Together: Two Small Helper Functions

Typing three commands each time is slow. Two shell functions in `~/.bashrc` turn the checks into one word each. Add them below the prompt lines from Chapter 2:

```
# pkg_installed NAME : is this Debian package installed?
pkg_installed() {
    dpkg-query -W -f='${db:Status-Abbrev}' "$1" 2>/dev/null | grep -q '^ii'
}

# have NAME : does this command exist?
have() {
    command -v "$1" >/dev/null 2>&1
}

# Example use
if pkg_installed git; then echo "git package: yes"; fi
if have rg; then echo "rg command: yes"; fi
```

Note that the two functions ask different questions, which is the point of the earlier finding-box. `pkg_installed ripgrep` and `have rg` are both true on a machine with the ripgrep package installed, while `have ripgrep` is false.

Hands-On Exercises

EXERCISE 1

For each of `git`, `rg`, `lspci`, `gcc` and `python3`, work out: whether the command exists, its path, its version, and which Debian package owns it. Present the results as a table, and explain any case where the command name and package name differ.

 [View solution](#)

EXERCISE 2

Write a script, `check-tools.sh`, that takes a list of commands and prints, for each one, whether it is present, its path, and the first line of its version output. It must exit with a non-zero status if any are missing. Run it on the Chapter 2 starter tools.

 [View solution](#)

EXERCISE 3

Create a tiny script named `hello` in `~/local/bin` and show how apt's tools treat it. Then run `apt-mark showmanual`, pick three packages you do not recognise, and use `apt show` to find out what they are and why they might be on the machine.

 [View solution](#)

CHAPTER 3 QUICK REFERENCE

- Ask three questions before installing: **is it installed, which version, where did it come from**
- `dpkg -s NAME` and `dpkg-query -W -f='${Package} ${Version} ${db:Status-Abbrev}\n' NAME` check one package
- In `dpkg -l`, `ii` means installed; `rc` means removed with config files left behind
- `apt policy NAME` shows **Installed** vs **Candidate** versions and their repositories
- `command -v NAME` tests for a command; `type -a NAME` shows every copy on the PATH, first one wins
- Package names and command names differ: `ripgrep` → `rg`, `pciutils` → `lspci`
- `dpkg -S FILE` finds the owning package; “no path found matching pattern” means apt does not manage it
- `dpkg -L PACKAGE` lists a package's files; `apt-file search` finds packages for files you do not have yet
- `apt-mark showmanual` is the list of packages you chose to install
- Piping `apt list` triggers a warning; use `2>/dev/null`, or prefer `dpkg-query` in scripts

Debian Development Machine Setup

Chapter 4 · The Graphics Card & Drivers

`devserver` has an MSI GeForce RTX 3050 LP 6G OC, which is an NVIDIA card, and this chapter gets it working properly. NVIDIA drivers are the one place where a Debian install regularly needs extra work: the good driver is not part of Debian's free software, it has to be compiled against your kernel, and Secure Boot adds a further step. None of it is hard once you know the order, and it is also the chapter where the habit from Chapter 3 (check first, change second) pays off most.

Your Card

The RTX 3050 LP 6G OC is a **low-profile**, single-slot card built on NVIDIA's Ampere architecture, with 6 GB of GDDR6 memory. MSI lists it as a 70 W card that takes all its power from the PCIe slot, with no separate power connector. That settles the two worries from Chapter 1. It is designed for exactly the kind of low, small-form-factor case you have, and it should not need a bigger power supply or an extra power lead. You should still check that the box includes a bracket that matches your case, and make sure the card is seated firmly.

PLUG THE MONITOR INTO THE CARD

Connect your display to one of the card's outputs, not to the motherboard's own video port. If the motherboard output stays active, the desktop may keep running on the integrated graphics and the card will sit idle.

Two Drivers to Choose From

	NOUVEAU (BUILT IN)	NVIDIA PROPRIETARY DRIVER
Comes from	The Linux kernel and Mesa, developed by the community	NVIDIA, packaged by Debian in the <code>non-free</code> component
Setup	Works with nothing installed	Enable <code>non-free</code> , install packages, possibly enrol a Secure Boot key

	NOUVEAU (BUILT IN)	NVIDIA PROPRIETARY DRIVER
Performance	Noticeably lower, especially on newer cards	Full performance
Compute (CUDA and similar)	Not available	Available
Free software only?	Yes	No

A development desktop can run happily on nouveau, and if you only need a working screen, that is a fair choice. But you have a 3050 for a reason, and you may one day want CUDA or machine-learning tools, so this chapter installs the proprietary driver. Debian 13 packages the 550 series (550.163.01), which supports Maxwell-generation cards up to Ada and Hopper, so your Ampere card is covered.

NOT THE NEWEST DRIVER

The 550 series does not support the newest NVIDIA architecture (Blackwell). That does not affect an RTX 3050, but it is worth knowing if you ever swap the card for a very recent one: check driver support first.

Step 1: Check Before You Change

Confirm that Linux can see the card, and find out what it is using now. Also find out whether Secure Boot is on, because that decides whether you need Step 5.

```
# Is the card visible, and which driver is in use?
lspci -k | grep -A3 -i 'vga\|3d'

# Is Secure Boot on? (install mokutil first if the command is missing)
sudo apt install mokutil
mokutil --sb-state

# Which kernel are you running?
uname -r
```

In the `lspci -k` output, look for the line `Kernel driver in use:`. Right now it is probably `nouveau`. If `lspci` shows no NVIDIA device at all, stop: reseal the card, check the BIOS/UEFI

setting that chooses the primary display (it may be set to the integrated graphics), and try again. The rest of the chapter assumes the card is detected. `mokutil --sb-state` prints either `SecureBoot enabled` or `SecureBoot disabled`.

Step 2: Enable the non-free Component

Chapter 2 left `non-free` off on purpose. Now there is a reason to turn it on. Open `/etc/apt/sources.list.d/debian.sources` and add `contrib` and `non-free` to the `Components:` line of both stanzas, so each reads:

```
Components: main contrib non-free non-free-firmware
```

You can make the edit by hand with `sudo nano` (or `sudo vim`), or make a backup and let `sed` change both lines at once:

```
sudo cp /etc/apt/sources.list.d/debian.sources ~/debian.sources.bak
sudo sed -i 's/^Components: .*/Components: main contrib non-free non-free-firmware/' \
/etc/apt/sources.list.d/debian.sources
sudo apt update
```

Afterwards, `apt policy nvidia-driver` should show a **Candidate** version. Before this step apt could not find the package at all, because it lives in `non-free`. That is the Chapter 3 check in action.

Step 3: Install the Driver

```
sudo apt install linux-headers-amd64 nvidia-kernel-dkms nvidia-driver nvidia-smi
```

PACKAGE	WHAT IT IS
linux-headers-amd64	The kernel header files for the current kernel. The driver module is compiled against these.

PACKAGE	WHAT IT IS
<code>nvidia-kernel-dkms</code>	The driver's kernel module, as source that DKMS compiles on your machine. This is the <i>proprietary</i> flavour.
<code>nvidia-driver</code>	The metapackage that pulls in the user-space parts: OpenGL support, the X.org driver, video acceleration.
<code>nvidia-smi</code>	A command-line tool that reports what the GPU is doing. It is how you verify the driver later.

WHAT DKMS DOES

A kernel module has to match the kernel it runs in. DKMS (Dynamic Kernel Module Support) keeps the module's source and rebuilds it automatically every time a new kernel is installed, so the driver keeps working across updates. It needs the matching headers to do that, which is why `linux-headers-amd64` is in the list. Installing the driver will take a while because apt compiles the module as part of the install.

Choose the proprietary flavour, not the open one

Debian also offers an *open* flavour of the module (`nvidia-open-kernel-dkms`) for newer cards, and your Ampere card would be supported. But the Debian wiki reports that the open flavour of the 550 driver stopped compiling as of kernel 6.12.100, and recommends the proprietary flavour. That is why this chapter installs `nvidia-kernel-dkms`.

Step 4: Wayland Settings

GNOME on Debian 13 runs on Wayland by default. For NVIDIA cards to work well with Wayland, the driver's `nvidia-drm` module needs a setting called `modeset` switched on. Check whether it already is, after the first reboot:

```
cat /sys/module/nvidia_drm/parameters/modeset
```

`Y` means it is on and you need do nothing. If it prints `N`, the Debian wiki recommends creating `/etc/modprobe.d/nvidia-wayland.conf` with these two lines:

```
options nvidia-drm modeset=1
options nvidia-drm fbdev=1
```

Then run `sudo update-initramfs -u` and reboot, so the setting is applied early in start-up.

Step 5: Secure Boot and the Machine Owner Key

If `mokutil --sb-state` said `SecureBoot disabled`, skip this section. If it said `enabled`, read on, because the firmware will refuse to load a kernel module that is not signed by a trusted key. Debian signs its own kernel and modules, but a DKMS module is compiled on your machine, so Debian cannot sign it. Instead, DKMS signs it with a key of your own, called a **Machine Owner Key** (MOK), and you tell the firmware to trust that key once.

DKMS generates the key automatically the first time it builds a module, so the driver install in Step 3 has already created it. Enrol it like this:

```
# Ask the firmware to trust the key at the next boot; you choose a one-time password
sudo mokutil --import /var/lib/dkms/mok.pub

# Confirm the request is pending
sudo mokutil --list-new

sudo reboot
```

On the reboot, a blue **MOK Manager** screen appears before Debian starts. Follow its on-screen instructions to enrol the key and enter the one-time password you just chose. This step can only be done at the machine's own keyboard at boot time, which is deliberate: it stops malware running inside Debian from approving its own key. Afterwards you can confirm the key is trusted:

```
sudo mokutil --list-enrolled
```

DO NOT SKIP THE BLUE SCREEN

If you reboot and miss the MOK Manager prompt, the key is not enrolled, the NVIDIA module will not load, and the desktop may start without acceleration or show a black screen. That is recoverable:

switch to a text console with `Ctrl + Alt + F3`, log in, and run `sudo mokutil --import /var/lib/dkms/mok.pub` again, then reboot and complete the blue screen. Alternatively, you can turn Secure Boot off in the firmware settings, which avoids the whole step at the cost of that protection.

Step 6: Verify

```
# Was the module built and installed for the running kernel?
dkms status

# Is it loaded?
lsmod | grep nvidia

# Is the driver in use? Expect: Kernel driver in use: nvidia
lspci -k | grep -A3 -i 'vga\|3d'

# Ask the GPU directly
nvidia-smi
```

`dkms status` should list an `nvidia` entry for your kernel with the word `installed`. `nvidia-smi` prints a table showing the card's name, the driver version and its memory. If it prints the table, the driver is loaded and talking to the card. If it says it could not communicate with the driver, the module is not loaded: go back to `dkms status`, and to the Secure Boot step.

Keeping It Working After Kernel Updates

This is the part people forget. A driver that worked yesterday can fail after tomorrow's kernel update. DKMS rebuilds the module for each new kernel, but that build can fail if the driver's source is not yet compatible with the new kernel.

A REAL EXAMPLE, FROM THE DEBIAN WIKI

The Debian wiki records two cases where the 550 driver stopped compiling: the open flavour on kernel 6.12.100, and even the proprietary 550 driver on the 6.19 kernels that reached `trixie-backports`. The lesson is to stay on Trixie's default kernel, and not to install a newer kernel from backports on this machine unless you have checked that the NVIDIA driver supports it.

A simple routine avoids nasty surprises. After any `apt full-upgrade` that installs a new kernel, and *before* you reboot, check that the module was built for it:

```
# Which kernels are installed?
dpkg -l 'linux-image*' | grep ^ii

# Is there an nvidia entry marked installed for the NEW kernel?
dkms status
```

If the new kernel has no `installed` nvidia entry, do not reboot into it yet. The old kernel is still installed and will still work; keep using it until an updated driver arrives.

If Something Goes Wrong

SYMPTOM	LIKELY CAUSE AND FIRST STEP
Card not listed by <code>lspci</code>	Reseat the card. Check the BIOS/UEFI primary-display setting.
<code>apt policy nvidia-driver</code> cannot find the package	<code>non-free</code> is not enabled, or you forgot <code>apt update</code> . Recheck Step 2.
The install errors while building the module	The headers are missing or do not match the running kernel. Install <code>linux-headers-amd64</code> and reboot into the current kernel.
Black screen after rebooting	Secure Boot key not enrolled, or the module did not load. Use <code>Ctrl + Alt + F3</code> for a text console.
<code>nvidia-smi</code> : cannot communicate with the driver	The module is not loaded. Check <code>dkms status</code> , <code>lsmod</code> and the Secure Boot state.

UNDOING IT

If you decide the proprietary driver is more trouble than it is worth, remove it with `sudo apt purge nvidia-kernel-dkms nvidia-driver nvidia-smi`, then `sudo apt autoremove`, and reboot. Read apt's list before you confirm, and the machine returns to nouveau. You will have lost nothing but the extra performance.

Hands-On Exercises

EXERCISE 1

Before changing anything, run the Step 1 checks on `devserver` and write down: the GPU as `lspci` reports it, the driver currently in use, the Secure Boot state, and the running kernel. From those four facts, list which of Steps 2–5 you will need to do and which you can skip.

 [View solution](#)

EXERCISE 2

Explain in your own words why a DKMS module cannot be signed by Debian, what a Machine Owner Key is, and why enrolling one requires you to be at the keyboard during boot. Then write the ordered steps to recover if you rebooted and skipped the blue MOK Manager screen.

 [View solution](#)

EXERCISE 3

Write a shell function `gpucheck` that prints the running kernel, the `dkms status` lines for nvidia, whether the nvidia module is loaded, and whether `nvidia-smi` works, and returns a non-zero status if the driver is not healthy. Explain when in your update routine you would run it.

 [View solution](#)

CHAPTER 4 QUICK REFERENCE

- The RTX 3050 LP 6G OC is a low-profile, single-slot Ampere card that draws power from the PCIe slot only (MSI: 70 W, no external connector)
- Plug the monitor into the card, not the motherboard
- Debian 13 packages NVIDIA driver **550.163.01**; it lives in `non-free`, so enable `contrib non-free` in `debian.sources`
- Install: `sudo apt install linux-headers-amd64 nvidia-kernel-dkms nvidia-driver nvidia-smi`
- Use the **proprietary** module flavour; the open flavour of 550 fails to compile on newer kernels
- DKMS recompiles the module for each new kernel and needs the matching headers
- Secure Boot on? `sudo mokutil --import /var/lib/dkms/mok.pub`, reboot, complete the blue MOK Manager screen

- Wayland: `cat /sys/module/nvidia_drm/parameters/modeset` should print `Y`
- Verify with `dkms status`, `lsmod | grep nvidia`, `lspci -k` and `nvidia-smi`
- After a kernel update, check `dkms status` *before* rebooting; stay on Trixie's default kernel, not backports

Debian Development Machine Setup

Chapter 5 · Python the Debian Way

Python is already on `devserver`, because Debian's own tools depend on it. That is exactly what makes it easy to get wrong. If you install packages into the Python that the operating system uses, you can break the operating system. Debian 12 and 13 refuse to let you do it by accident, and this chapter explains why, and gives you three clean ways of working: virtual environments for projects, `pipx` for command-line tools, and `pyenv` for when you need a different Python version. It builds directly on the “which copy runs?” habit from Chapter 3.

The Python You Already Have

Debian 13 ships Python **3.13** (the current package is 3.13.5). Check it with the tools from Chapter 3:

```
python3 --version
type -a python3
dpkg -S "$(command -v python3)"
apt policy python3
```

You should see one copy, `/usr/bin/python3`, owned by a Debian package. That is the **system Python**. Note there is no plain `python` command, only `python3`. (An optional package, `python-is-python3`, adds one, but you do not need it.)

A fresh desktop install often has the interpreter but not everything you need to develop. Install the extras now:

```
sudo apt install python3-pip python3-venv python3-dev pipx
```

PACKAGE	WHY
python3-pip	The <code>pip</code> installer.
python3-venv	Debian splits virtual-environment support into its own package, so <code>python3 -m venv</code> needs this.

PACKAGE	WHY
<code>python3-dev</code>	Header files for compiling Python extensions written in C. Many packages need it when they install.
<code>pipx</code>	Installs Python command-line tools each in their own isolated environment. Debian 13 has version 1.7.1.

Why `pip install` Refuses

Now try the thing every tutorial tells you to do:

```
pip install requests
```

On Debian 12 and 13 this stops with an error that begins `error: externally-managed-environment`, and explains that the environment is managed by the operating system. This is not a fault. Debian is following **PEP 668**, a Python standard that lets a distribution mark its Python as “externally managed”, so pip will not modify it. Debian marks the interpreter with a file named `EXTERNALLY-MANAGED` in the standard-library directory, and pip checks for it.

WHAT IT IS PROTECTING YOU FROM

Debian's own tools are written in Python and depend on particular versions of particular libraries. If `pip` installed or upgraded one of those libraries system-wide, a Debian tool could stop working, and the fault would appear somewhere unrelated to what you did. Keeping your packages out of the system Python removes that whole category of problem.

THE OVERRIDE EXISTS. DON'T USE IT AS A HABIT.

`pip install PACKAGE --break-system-packages` overrides the safeguard, and the Debian wiki mentions it. It puts you back in exactly the situation the rule prevents. Reserve it for a throwaway machine, never for `devserver`. Also avoid `sudo pip install`, which is the same mistake with more authority.

Which Tool for Which Job

Instead of overriding the rule, choose the right tool. The question to ask is: what am I installing?

YOU WANT TO...	USE	BECAUSE
Write a project that needs libraries	A virtual environment	Each project gets its own libraries and versions, separate from the system and from other projects.
Install a command-line tool written in Python	pipx	The tool gets its own private environment but is available as an ordinary command.
Get a Python version Debian does not ship	pyenv	Builds and switches between other Python versions in your home directory.
Get a library that an <i>apt-installed program</i> needs	apt (<code>python3-NAME</code>)	Debian packages many libraries as <code>python3-*</code> and keeps them consistent with the system.

Virtual Environments

A virtual environment (venv) is a folder containing its own copy of the Python interpreter link and its own place for packages. Create one inside each project, and activate it when you work there:

```
mkdir -p ~/projects/demo && cd ~/projects/demo
python3 -m venv .venv
source .venv/bin/activate

# Your prompt now shows (.venv). Check which copies the shell finds:
type -a python3
pip --version

pip install requests
pip list

deactivate
```

After `source .venv/bin/activate`, `type -a python3` lists the venv's copy *first*. That is the whole trick: activation puts the environment's `bin` directory at the front of your `PATH`, so `python3` and `pip` now mean the ones inside the venv. `pip install requests` works, and the package lands inside `.venv`, not in the system. `deactivate` puts the `PATH` back.

NEVER COMMIT THE VENV, AND NEVER MOVE IT

A venv is not portable, because scripts inside it contain absolute paths. If you move a project, recreate the environment rather than copying it. Since Python 3.13, `python3 -m venv` writes a `.gitignore` file into the environment so that Git ignores it automatically. What you *should* commit is a list of the packages the project needs, which is the next step.

Recording and recreating an environment

```
# With the venv active: write down exactly what is installed
pip freeze > requirements.txt

# Later, on any machine: build a fresh venv and install from the list
python3 -m venv .venv
source .venv/bin/activate
pip install -r requirements.txt
```

Because a venv is disposable, you can delete `.venv` and rebuild it from `requirements.txt` in seconds. That is what makes it safe to experiment.

pipx: Tools, Not Libraries

Some Python software is not a library for your project but a program you want to run: a code formatter, an HTTP client, a linter. `pipx` installs each one into its own private virtual environment and puts its command where you can run it. The Debian wiki recommends `pipx` as the easiest way to work with the externally-managed rule.

```
# One-time: make sure ~/.local/bin is on your PATH (log out and in afterwards)
pipx ensurepath

# Install a tool, and see what pipx manages
pipx install httpie
pipx list

http --version
command -v http

pipx upgrade httpie
pipx uninstall httpie
```

The `http` command works from anywhere, yet its libraries are isolated and cannot clash with anything else. The command itself sits in `~/.local/bin`, which Chapter 3 listed as a place where software installed for one user lives outside `apt`.

pyenv: Other Python Versions

Debian 13 gives you one Python, 3.13. That is fine most of the time. But a project may need an older version to match a server, or you may want to test on a newer one. `pyenv` builds extra Python versions from source into your home directory, and lets you choose per project.

Because it compiles Python, `pyenv` needs a set of build libraries first. The `pyenv` project's own suggested list for Debian is:

```
sudo apt update; sudo apt install make build-essential libssl-dev zlib1g-dev \
  libbz2-dev libreadline-dev libsqlite3-dev curl git \
  libncursesw5-dev xz-utils tk-dev libxml2-dev libxmlsec1-dev libffi-dev liblzma-dev lib
```

`pyenv`'s documentation offers a one-line installer that pipes a downloaded script into `bash`. It is worth knowing that running a downloaded script blindly is exactly what the verify-before-you-trust habit from Chapter 1 warns against, so this course uses the other documented method: clone the project and read what you are running.

```
git clone https://github.com/pyenv/pyenv.git ~/.pyenv
```

Then add these lines to `~/.bashrc` (and the same lines to `~/.profile`, as the `pyenv` documentation advises):

```
export PYENV_ROOT="$HOME/.pyenv"
[[ -d $PYENV_ROOT/bin ]] && export PATH="$PYENV_ROOT/bin:$PATH"
eval "$(pyenv init - bash)"
```

Open a new terminal, then use it:

```
pyenv install --list | grep -E '^s*3\.[12]' # see what is available
pyenv install 3.12.x                       # replace with a real version from the
```

```
pyenv versions
```

```
cd ~/projects/demo
```

```
pyenv local 3.12.x # writes a .python-version file in this folder
```

```
python --version
```

`pyenv local` writes a `.python-version` file, so a project always uses the version you chose for it, and the file can be committed to Git. Building takes a few minutes, that is normal.

CHAPTER 3'S WARNING, IN ACTION

pyenv works by putting its own *shims* directory at the front of your `PATH`. That means `type -a python3` now lists a pyenv shim before `/usr/bin/python3`, which is exactly the “two copies, first one wins” situation from Chapter 3, arranged on purpose. If you ever wonder which Python is running, `type -a python3` and `pyenv versions` answer it.

YOU MAY NOT NEED PYENV AT ALL

Do not install pyenv just because it exists. If 3.13 does everything you need, skip it: every extra tool on the `PATH` is one more thing to explain when something behaves oddly. Install it the day a real project needs a different version.

When apt Is the Right Answer

Debian packages many Python libraries as `python3-NAME` (for example `python3-requests`). Use `apt` when a library is needed by a program you installed with `apt`, or when you want a tool that is part of the operating system's own ecosystem. The catch is that the version is whatever Debian froze at release, often older than PyPI's latest, which is why projects use `venvs`. Check with the Chapter 3 tools before you choose:

```
apt policy python3-requests
```

```
apt search --names-only '^python3-flask$'
```

For deeper Python work, see the Python Fundamentals course. Here we only need the machine to be set up so that course, and every project after it, starts cleanly.

Hands-On Exercises

EXERCISE 1

Install the four packages from this chapter and verify them with the Chapter 3 tools. Then run `pip install requests` outside any virtual environment, copy the error, and explain in your own words what PEP 668 is and why Debian enforces it. Finally, do the same install correctly inside a venv.

 [View solution](#)

EXERCISE 2

Prove a virtual environment is disposable. In a new project folder, create a venv, install `requests`, freeze the requirements, delete the venv completely, recreate it from `requirements.txt`, and confirm the package is back. Also check what `type -a python3` reports before, during and after activation.

 [View solution](#)

EXERCISE 3

Use `pipx` to install a command-line tool, and show where its command lives and that its libraries are not visible to the system Python. Then, for each of five scenarios you invent or are given (a Flask project, a code formatter, a project that needs Python 3.11, a library used by an apt-installed program, a quick throwaway experiment), name the right tool and justify the choice.

 [View solution](#)

CHAPTER 5 QUICK REFERENCE

- Debian 13 ships **Python 3.13** as `python3` (no plain `python`); it is the **system Python**, used by Debian's own tools
- Install: `sudo apt install python3-pip python3-venv python3-dev pipx`
- `pip install` outside a venv fails with `externally-managed-environment` (PEP 668): a deliberate protection, not a fault
- Avoid `--break-system-packages` and `sudo pip install`
- Projects: `python3 -m venv .venv`, `source .venv/bin/activate`, `deactivate`

- Record and rebuild: `pip freeze > requirements.txt`, `pip install -r requirements.txt`
- Venvs are not portable and should not be committed; Python 3.13 adds a `.gitignore` inside automatically
- Command-line tools: `pipx install NAME`, `pipx list`, `pipx upgrade`, `pipx uninstall`, plus `pipx ensurepath` once
- Other Python versions: pyenv (clone to `~/.pyenv`, needs the build libraries; `pyenv local VERSION` writes `.python-version`)
- Activation and pyenv both work by putting their own directory first on `PATH`; `type -a python3` shows which copy wins
- Use apt (`python3-NAME`) for libraries needed by apt-installed programs

Debian Development Machine Setup

Chapter 6 · Node.js & npm

Node.js has the same shape of problem as Python in Chapter 5, with a different twist. Debian packages Node, but the packaged version is fixed at release time, and Node moves quickly. So the question for `devserver` is not just “how do I install Node?” but “which Node, and how do I change my mind later?” This chapter answers both, and finishes with npm, Node's package manager.

What Debian Gives You

Start with the Chapter 3 checks, before installing anything:

```
dpkg -s nodejs
command -v node
type -a node
apt policy nodejs npm
```

On a fresh desktop install Node is probably not there. `apt policy` shows what Debian offers. On Trixie the `nodejs` package is **20.19.2** (with Debian's security fixes applied), and `npm` is a *separate* package, version 9.2.0, with a long list of dependencies. The `nodejs` package does depend on `corepack`, a helper that can give you access to other JavaScript package managers.

THE PACKAGED VERSION IS ALREADY PAST UPSTREAM END-OF-LIFE

Node's own release schedule lists Node 20 with an end date of **30 April 2026**. Debian's package still receives security fixes from the Debian team, so it is not abandoned, but it is a release line that the Node project itself no longer maintains. That is a reasonable version to run a system tool on. It is not a good version to *start new projects on*, because the libraries you install will expect something newer.

Which Node? The Release Lines

Node publishes a new major version each year, and gives each an **LTS** (Long Term Support) phase. As of the end of September 2026, the schedule looks like this:

VERSION	STATUS NOW	MAINTENANCE PHASE BEGINS	END OF LIFE
Node 20 (Iron)	End of life	22 October 2024	30 April 2026
Node 22 (Jod)	Maintenance LTS	21 October 2025	30 April 2027
Node 24 (Krypton)	Active LTS	20 October 2026	30 April 2028
Node 26	Current (becomes LTS on 28 October 2026)	20 October 2027	30 April 2029

“Active LTS” is the sweet spot for real work: it is stable and still receiving improvements. Node 24 is the one to pick today, and it is only a few weeks from moving to the less active Maintenance phase, which is still fully supported. The Node project's own advice is that production applications should use Active or Maintenance LTS releases. Check `nodejs.org` before you decide, because these dates come from the project's published schedule and can be adjusted.

Three Ways to Get Node

METHOD	VERSION YOU GET	SWITCHING VERSIONS	TRADE-OFF
Debian package (<code>apt install nodejs npm</code>)	Fixed: Node 20 on Trixie	Not possible	Simple and integrated with apt, but old, and npm drags in a large set of packages.
A third-party apt repository (such as NodeSource)	The major version the repository provides	Only by changing the repository	Newer, still managed by apt, but you trust a third party's key and packages. Linux Package Managers 3 covers adding repositories safely.
nvm (a version manager)	Any version you choose	One command, or automatic per project	Lives in your home directory, outside apt. You manage it yourself.

For a development machine, the version manager is the best fit, for the same reason pyenv was the answer in Chapter 5: different projects want different Node versions, and you should be able to switch without touching the system. This chapter uses **nvm** (Node Version Manager). If you

install Node this way, do *not* also install the Debian `nodejs` package. Two Nodes on one machine is exactly the confusion Chapter 3 taught you to avoid.

Installing nvm

The nvm project's documentation gives a one-line installer that pipes a downloaded script into `bash`. As with pyenv, the safer route is to clone the repository at a specific release and add the shell lines yourself, so you know exactly what runs. The current release at the time of writing is **v0.40.8**; check the project page for a newer one.

```
git clone https://github.com/nvm-sh/nvm.git ~/.nvm
cd ~/.nvm && git checkout v0.40.8
```

Then add these lines to `~/.bashrc`:

```
export NVM_DIR="$HOME/.nvm"
[ -s "$NVM_DIR/nvm.sh" ] && \. "$NVM_DIR/nvm.sh"
[ -s "$NVM_DIR/bash_completion" ] && \. "$NVM_DIR/bash_completion"
```

Open a new terminal and check it worked. Note that `nvm` is a shell *function*, not a program, so the usual `which nvm` will find nothing. Use `command -v`:

```
command -v nvm      # prints: nvm
nvm --version
```

Installing and Switching Node Versions

```
# Install the latest LTS release and make it the default for new terminals
nvm install --lts
nvm alias default 'lts/*'

node --version
npm --version
type -a node
```

```
# See what is installed, and which is active
nvm ls
nvm current

# Install another line alongside it, and switch
nvm install 22
nvm use 22
node --version
```

`type -a node` now shows a path under `~/.nvm/versions/node/`. That is Chapter 3's “first copy wins” idea again: nvm puts the chosen version's directory at the front of your `PATH`. Switching with `nvm use` changes which directory is first. Each Node version also comes with its own npm.

Choosing a version per project

Put the version a project needs in a file called `.nvmrc` in the project's root, and commit it. It contains a version such as `24`, or `lts/*`. Then, from inside the project:

```
echo 24 > .nvmrc
nvm use          # reads .nvmrc; run 'nvm install' instead if that version is missing
```

MATCH THE SERVER

Whatever Node version `debserver` will run should be the version `.nvmrc` names, so what you test here is what you deploy there. Decide that before you start the project, not after the first surprise.

npm: Projects and Global Tools

npm does two different jobs, and confusing them causes most npm problems. You already know the equivalent split from Python (Chapter 5): libraries for a project, and tools you run.

Project dependencies

```
mkdir -p ~/projects/node-demo && cd ~/projects/node-demo
npm init -y                    # creates package.json
npm install lodash             # adds a dependency: node_modules/, package-lock.json
npm install -D prettier        # adds a development-only dependency
```

```
npm ls --depth=0          # what is installed
npm outdated              # what has newer versions
```

FILE / FOLDER	WHAT IT IS	COMMIT IT?
<code>package.json</code>	Your project's description and its list of dependencies.	Yes
<code>package-lock.json</code>	The exact version of every package installed, including dependencies of dependencies.	Yes
<code>node_modules/</code>	The installed packages themselves. Large, and rebuildable.	No (add to <code>.gitignore</code>)

This is the same idea as `requirements.txt` in Chapter 5. The lock file is the record, and `node_modules` is disposable. To rebuild exactly what the lock file describes, use `npm ci` (a clean install from the lock file), rather than `npm install`, which may update versions:

```
rm -rf node_modules
npm ci
```

Global tools

Use `npm install -g` for command-line tools you want available everywhere. With nvm this needs **no** `sudo`, because the tools install into the current Node version's own folder in your home directory:

```
npm install -g cowsay
npm ls -g --depth=0
command -v cowsay
cowsay "hello from devserver"
```

GLOBAL TOOLS BELONG TO ONE NODE VERSION

Each Node version installed by nvm keeps its own global packages. If you switch to another version with `nvm use`, a tool you installed globally under the first version can seem to vanish. It has not gone; it is simply installed under the other version. You can carry tools across with `nvm install --reinstall-packages-from=current NEW-VERSION`. For anything a project depends on, list it in that project's `package.json` instead of relying on a global copy.

RUNNING A TOOL ONCE

`npx` runs a package's command without installing it globally: `npx cowsay hi`. It is a good way to try something once without leaving anything behind.

BE CHOOSY ABOUT WHAT YOU INSTALL

npm packages can run scripts when they install, and a typical project pulls in many dependencies you never chose. Prefer well-known, actively maintained packages, look at what a package is before installing it, and keep the lock file in Git so you can see exactly what changed when it changes.

Which Tool for Which Job

YOU WANT TO...	USE
Pick or switch the Node version	nvm, with a <code>.nvmrc</code> per project
Add a library to a project	<code>npm install NAME</code> (locked in <code>package-lock.json</code>)
Rebuild a project exactly	<code>npm ci</code>
Install a command-line tool for yourself	<code>npm install -g NAME</code> (or <code>npx</code> for a one-off)
Run something Debian itself depends on	The apt <code>nodejs</code> package (only if you need it)

For the language itself, see the Node.js Fundamentals course. The next chapter does for PHP what these last two did for Python and Node.

Hands-On Exercises

EXERCISE 1

Before installing anything, run the Chapter 3 checks for Node on `devserver`. Then, using the release-line table and the Debian package facts, decide which Node version you will install and write a short justification. Say what would change your mind.

 [View solution](#)

EXERCISE 2

Install nvm and the current LTS, and record what `type -a node` reports. Then install a second Node major version, create two project folders each with its own `.nvmrc`, and show that `nvm use` selects the right version in each.

 [View solution](#)

EXERCISE 3

Create an npm project, install `lodash`, write a small script that uses it, then delete `node_modules` and restore it with `npm ci`. Next, install `cowsay` globally under one Node version, switch versions, and explain what you see and how to fix it.

 [View solution](#)

CHAPTER 6 QUICK REFERENCE

- Debian 13 packages **Node 20.19.2**; `npm` (9.2.0) is a separate, dependency-heavy package. Upstream Node 20 reached end of life on 30 April 2026
- As of September 2026: Node 24 is **Active LTS**, Node 22 is Maintenance LTS, Node 26 becomes LTS on 28 October 2026 (check nodejs.org)
- For development, use a version manager (**nvm**) rather than the Debian package; do not install both
- Install nvm at a release tag: `git clone https://github.com/nvm-sh/nvm.git ~/.nvm` then `git checkout v0.40.8`, and add three lines to `~/.bashrc`
- `command -v nvm` (it is a function, so `which` does not work)
- `nvm install --lts`, `nvm alias default 'lts/*'`, `nvm ls`, `nvm use VERSION`, `nvm current`
- A `.nvmrc` file pins a project's Node version; `nvm use` reads it
- Commit `package.json` and `package-lock.json`; never commit `node_modules`
- `npm ci` rebuilds exactly from the lock file; `npm install` may update versions
- With nvm, `npm install -g` needs no sudo, but global tools belong to one Node version; `npmx` runs a tool once

Debian Development Machine Setup

Chapter 7 · PHP, Composer & Databases

Your web projects run on PHP and MySQL, so `devserver` needs both. The plan follows the pattern from the last two chapters: check what is there, install the right thing, and keep a clear line between the development machine and the deployment target. There is one trap in the PHP package names that this chapter steers you around, and one surprise about “MySQL” on Debian.

Check First

```
command -v php composer mariadb mysql sqlite3
dpkg -l 'php*' | grep ^ii
apt policy php-cli composer mariadb-server
```

On a fresh desktop install none of these should be present. `apt policy` shows what Debian 13 offers: PHP **8.4** (the current build is 8.4.26), Composer **2.8.8**, and MariaDB **11.8.6**.

Installing PHP Without a Web Server

Here is the trap. The obvious command, `sudo apt install php`, installs a metapackage that depends on `php8.4`, and `php8.4` in turn requires *one* of a server-side interpreter: the Apache module, `php8.4-fpm` or `php8.4-cgi`. On a typical install, that means apt pulls in Apache and starts a web server you did not ask for. On a development machine where you will use PHP from the command line and its built-in test server, you do not want that.

DO NOT INSTALL THE `PHP` METAPACKAGE HERE

Installing `php` is designed for someone setting up a web server. Install `php-cli` and the extensions you need instead. Read apt's list of packages to be installed before you answer `y`: if it includes `apache2` or `libapache2-mod-php8.4`, you have installed the wrong thing. Chapter 3's `apt policy` and apt's own summary are how you catch this.

```
sudo apt install php-cli php-mbstring php-xml php-curl php-zip php-intl php-mysql php-sqlite3
```

PACKAGE	PROVIDES
php-cli	The <code>php</code> command, without any web server.
php-mbstring	Multi-byte string handling, needed for UTF-8 text including Japanese.
php-xml	XML and DOM support. Many frameworks and tools require it.
php-curl	HTTP requests to other services.
php-zip	Reading and writing zip archives. Composer uses it.
php-intl	Internationalisation support, needed by some libraries.
php-mysql	The <code>mysqli</code> and PDO MySQL drivers, for talking to MySQL and MariaDB.
php-sqlite3	The SQLite driver, for lightweight local databases.

Verify with the same checks you have used since Chapter 3:

```
php -v
php -m | grep -Ei 'mbstring|curl|mysqli|pdo_mysql|pdo_sqlite|zip|intl'
type -a php
```

`php -m` lists the loaded modules; the ones you asked for should all appear. `type -a php` should show a single `/usr/bin/php`.

Trying it without a web server

PHP has a small development web server built in. Create a folder with an `index.php` and serve it:

```
mkdir -p ~/projects/php-demo && cd ~/projects/php-demo
echo '<?php echo "Hello from PHP " . PHP_VERSION . " on " . gethostname();' > index.php
php -S localhost:8000
```

Open `http://localhost:8000` in a browser. Stop the server with `Ctrl + C`. The built-in server is only for development: it is single-purpose and not meant for real traffic, which is what the web server on `debserver` is for. Because it listens on `localhost`, nothing else on your network can reach it.

Matching the server

Your web server still runs an older Debian, so its PHP will differ from `devserver`'s 8.4. On `debserver`, run `php -v` and note the version. When you reinstall it with Trixie, the two will match. Until then, write code that runs on both, and test on the server before you trust it. If you ever genuinely need a different PHP version alongside 8.4, Debian's own repositories cannot supply it, and the usual options are a third-party repository (Linux Package Managers 3 explains how to add one safely and what you are trusting when you do) or a container, as in the Docker for Beginners course.

Composer

Composer is PHP's dependency manager, the equivalent of npm (Chapter 6) and pip (Chapter 5). Debian packages it, and installing it that way is simple and keeps it managed by apt:

```
sudo apt install composer git unzip
composer --version
```

Composer recommends `git` and `unzip`, which is why they are in the command: Composer downloads many packages as archives and works better with both present. You have already seen the pattern for project dependencies:

FILE / FOLDER	WHAT IT IS	COMMIT IT?
<code>composer.json</code>	The project's list of wanted packages and version ranges.	Yes
<code>composer.lock</code>	The exact version of every package installed.	Yes
<code>vendor/</code>	The installed packages, plus an autoloader.	No (add to <code>.gitignore</code>)

```
cd ~/projects/php-demo
composer init --no-interaction --name="me/php-demo"
composer require monolog/monolog
```

```
composer show

# Rebuild exactly what composer.lock describes
rm -rf vendor
composer install
```

To use an installed package in your code, include Composer's autoloader once at the top of your script: `require __DIR__ . '/vendor/autoload.php';`. As with `npm ci`, the important distinction is between `composer install` and `composer update`. **Install** reads `composer.lock` and installs exactly those versions, so it gives the same result everywhere. **Update** ignores the lock, fetches the newest versions that `composer.json` allows, and rewrites the lock. Use install to rebuild, and update only when you mean to upgrade.

COMPOSER NEEDS PHP, AND PHP NEEDS THE RIGHT EXTENSIONS

If `composer require` complains that a package needs a PHP extension you have not installed, read the message: it names the extension. Install the matching `php-NAME` package with apt, then run the command again.

Databases: What “MySQL” Means on Debian

Here is the surprise. Oracle's MySQL is **not** in Debian's repositories. On Trixie there is no package called `mysql-server`. Instead there are metapackages called `default-mysql-server` and `default-mysql-client`, which install **MariaDB** (version 11.8.6), a fork of MySQL that is compatible for nearly everything a PHP project does. If you have been running a database on Debian and calling it MySQL, it is very likely MariaDB already. Check on `debserver` with `mysql --version` or `mariadb --version`: MariaDB announces itself in the output.

SAME ON BOTH MACHINES

Because both `devserver` and `debserver` are Debian, they will both use MariaDB from Debian's repositories, so using it on `devserver` keeps your development database as close as possible to your real one. Differences in the version number will shrink once the server is reinstalled with Trixie.

Installing and securing a local database

```
sudo apt install mariadb-server
systemctl status mariadb
mariadb --version

# Which addresses is it listening on? Expect 127.0.0.1 only
ss -tlnp | grep 3306
```

The server should listen only on `127.0.0.1`, meaning only programs on `devserver` can connect. Confirm that with `ss`, rather than assuming it. If a database is listening on `0.0.0.0`, it is open to your whole network, and that is not what you want on a development box.

On Debian, MariaDB's administrator account is tied to your operating-system login through a Unix socket, so `sudo mariadb` gets you in as the administrator without a database password. Older tutorials use `mysql` as the command name; on current MariaDB, `mariadb` is the command to use (a `mysql` name may also work through Debian's compatibility packages, but do not rely on it). Use the administrator account to create a database and a separate, limited user for your projects:

```
sudo mariadb

-- inside the MariaDB prompt:
CREATE DATABASE devdb CHARACTER SET utf8mb4;
CREATE USER 'dev'@'localhost' IDENTIFIED BY 'choose-a-password';
GRANT ALL PRIVILEGES ON devdb.* TO 'dev'@'localhost';
EXIT;
```

The `dev` user can do anything inside `devdb` and nothing anywhere else, which limits the damage a bug or a stray script can do. Test it from PHP using PDO, the database interface you will use in most projects. Save this as `db-test.php` and run `php db-test.php`:

```
<?php
$pdo = new PDO(
    'mysql:host=localhost;dbname=devdb;charset=utf8mb4',
    'dev',
    'choose-a-password',
    [PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION]
);

$pdo->exec('CREATE TABLE IF NOT EXISTS notes (id INT AUTO_INCREMENT PRIMARY KEY, body VA
```

```
$pdo->prepare('INSERT INTO notes (body) VALUES (?)')->execute(['hello from devserver']);

foreach ($pdo->query('SELECT id, body FROM notes') as $row) {
    echo $row['id'], ': ', $row['body'], PHP_EOL;
}
```

KEEP PASSWORDS OUT OF YOUR CODE AND OUT OF GIT

The password above is written in the script only so this example is easy to follow. In a real project, put credentials in a configuration file or environment variable that is listed in `.gitignore`, and never commit them. Notice the query uses a *prepared statement* (`prepare` and `execute`) with a placeholder instead of building SQL from text; that habit is what protects you from SQL injection. The Personal Catalogue: PHP & MySQL course builds a full application on this same pattern.

Starting and stopping it

A database that runs all day uses memory and starts on every boot. With 64 GB of RAM that is no burden, but you may prefer to run it only when needed:

```
# Stop it now, and stop it starting at boot
sudo systemctl disable --now mariadb

# Start it when you want it
sudo systemctl start mariadb
```

SQLite: a database with no server

For small tools and quick experiments, you do not need a database server at all. SQLite stores a whole database in a single file:

```
sudo apt install sqlite3
sqlite3 ~/projects/php-demo/dev.db "CREATE TABLE IF NOT EXISTS t (x TEXT); INSERT INTO t
```

PHP reaches it through the `php-sqlite3` extension you already installed, using PDO with a connection string like `sqlite:/path/to/dev.db`. Choose SQLite for something small and self-contained, and MariaDB when you need to match the server's database.

LOOKING AHEAD

Chapter 9 installs Gitea, which can store its data in SQLite or in a database server. The databases you set up here are what that choice will be made from.

Which Tool for Which Job

YOU WANT TO...	USE
Run PHP scripts and try a site locally	<code>php-cli</code> and <code>php -S localhost:8000</code>
Add PHP libraries to a project	<code>composer require</code> (recorded in <code>composer.lock</code>)
Rebuild a project's libraries exactly	<code>composer install</code>
Match the server's database	MariaDB (<code>mariadb-server</code>)
A small, serverless database	SQLite (<code>sqlite3</code> , <code>php-sqlite3</code>)
A different PHP version	A third-party repository or a container

Hands-On Exercises

EXERCISE 1

Install PHP the correct way (without the `php` metapackage). Verify the version and the loaded extensions, confirm no web server was installed, and serve a page with the built-in server. Explain why `apt install php` would have been the wrong command here.

 [View solution](#)

EXERCISE 2

Create a Composer project, require a package, write a script that uses it via the autoloader, then delete `vendor/` and restore it with `composer install`. Explain the difference between `composer install` and `composer update`, and say which files you would commit.

 [View solution](#)

EXERCISE 3

Install MariaDB, prove it listens only on localhost, create a database and a limited user, and write a PDO script that inserts and reads a row. Then record the PHP and database versions on `debserver` and say what differences you expect from `devserver` and how you would deal with them.

 [View solution](#)

CHAPTER 7 QUICK REFERENCE

- Debian 13: PHP **8.4** (8.4.26), Composer **2.8.8**, MariaDB **11.8.6**
- Do **not** install the `php` metapackage on a dev machine; it requires a server interpreter (Apache module, FPM or CGI)
- Install `php-cli php-mbstring php-xml php-curl php-zip php-intl php-mysql php-sqlite3`
- `php -v`, `php -m` to check; `php -S localhost:8000` is a development-only web server
- Composer: commit `composer.json` and `composer.lock`, not `vendor/`; `composer install` uses the lock, `composer update` rewrites it
- Oracle MySQL is not packaged in Debian; `default-mysql-server` installs **MariaDB**
- `sudo mariadb` gives administrator access through your Unix login; use the `mariadb` command, not `mysql`
- Check the database listens on `127.0.0.1` only: `ss -tlnp | grep 3306`
- Give each project a database and a limited user; use prepared statements; keep passwords out of Git
- SQLite (`sqlite3`, `php-sqlite3`) is a database in one file, with no server
- A different PHP version needs a third-party repository or a container

Debian Development Machine Setup

Chapter 8 · Editors & IDEs on the Desktop

The languages are installed, so now you need somewhere to write code. This chapter is about setting up editors on `devserver`, not about learning them: VS Code Essentials, Learning Vim and Text Editors & IDEs Survey already teach how to use the tools. Here we deal with the Debian-specific parts: where to get VS Code from, how to keep it updated, how to choose a default editor for the whole system, and how to record your extensions so a rebuild is painless.

What You Already Have

```
command -v nano vim nvim code
apt policy vim neovim code
readlink -f "$(command -v editor)"
```

A Debian desktop already has `nano`, and you installed `vim` in Chapter 2. The last command shows where Debian's `editor` command currently points. That is the system's default terminal editor, which programs like Git and `sudoedit` call when they need you to type something. `apt policy code` will show nothing yet: VS Code is not in Debian's own repositories.

EDITOR	TYPE	ON DEBIAN 13	GOOD FOR
nano	Terminal	Included	Quick edits to config files; the easy one to learn.
Vim	Terminal	Version 9.1 in Debian's repositories (<code>vim</code>)	Fast editing anywhere, including over SSH on <code>debserver</code> .
Neovim	Terminal	Version 0.10.4 in Debian's repositories (<code>neovim</code>)	A modern, extensible take on Vim.
VS Code	Graphical IDE	Not in Debian; from Microsoft's repository	Larger projects, debugging, extensions, integrated terminal and Git.

You do not have to choose one. A common pattern is VS Code for projects and a terminal editor for quick edits and for work over SSH. The rest of this chapter sets up that combination.

Installing VS Code from Microsoft's Repository

VS Code is Microsoft's program, not part of Debian, so you install it from Microsoft's own apt repository. That is a **third-party repository**: you are choosing to trust a company outside Debian to supply software that will run with your user's permissions, and to keep it updated. Linux Package Managers 3 explains what that trust involves, and Chapter 4 already showed you the same idea for the NVIDIA driver, though that one came from Debian. Do it deliberately, in three steps.

Step 1: Get Microsoft's signing key

```
sudo apt install wget gpg
wget -qO- https://packages.microsoft.com/keys/microsoft.asc | sudo gpg --dearmor -o /usr
```

This downloads Microsoft's public key and converts it into the binary format apt uses. Piping a downloaded *key* into `gpg` is not the same risk as piping a downloaded *script* into `bash`, because `gpg --dearmor` only converts data and does not run it. Still, apply the habit from Chapter 1 and look at what you have:

```
gpg --show-keys /usr/share/keyrings/microsoft.gpg
```

This prints the key's owner and fingerprint. Compare the fingerprint with the one published by Microsoft before you rely on it; a key you obtained over HTTPS from Microsoft's own site is reasonable to trust, but the check costs a minute.

Step 2: Add the repository

Create `/etc/apt/sources.list.d/vscode.sources` in the same deb822 format you met in Chapter 2:

```
Types: deb
URIs: https://packages.microsoft.com/repos/code
Suites: stable
Components: main
Architectures: amd64,arm64,armhf
Signed-By: /usr/share/keyrings/microsoft.gpg
```

Read the `Signed-By` line carefully. It tells apt that packages from *this* repository must be signed by *this* key, and that this key is trusted for this repository only. That keeps Microsoft's key from vouching for anything else.

Step 3: Install

```
sudo apt update
sudo apt install code
```

Then check it with the Chapter 3 tools:

```
apt policy code      # the repository line shows packages.microsoft.com
code --version
dpkg -S "$(command -v code)"
type -a code
```

THE INSTALLER SHORTCUT

Microsoft also offers a `.deb` file to download. Installing it offers to add the repository and key for you. That is convenient, but it does the repository setup on your behalf without showing you the steps. This course uses the manual method so that you can see what you have trusted, and it leaves a file (`vscode.sources`) that Chapter 10 can put in the rebuild script.

UPDATES NOW COME THROUGH APT

Because VS Code was installed from an apt repository, `sudo apt update && sudo apt full-upgrade` updates it along with everything else, and Chapter 2's routine covers it. Microsoft notes the repository can lag behind a release by up to three hours, so a brand-new version may take a little while to appear. The trade-off of a third-party repository is that you have added one more source of code to your machine, so keep it to the ones you actually need.

Extensions and Settings

VS Code is only as useful as its extensions. You can install them from the command line, which is what makes them recordable:

```
code --install-extension ms-python.python
code --install-extension bmewburn.vscode-intelephense-client
code --install-extension dbaeumer.vscode-eslint
code --install-extension esbenp.prettier-vscode
code --install-extension ms-vscode-remote.remote-ssh

code --list-extensions
```

EXTENSION	WHY
Python (<code>ms-python.python</code>)	Runs and debugs Python, and finds your project's virtual environment from Chapter 5.
PHP Intelephense (<code>bmewburn.vscode-intelephense-client</code>)	Code completion and error hints for PHP.
ESLint and Prettier	Checks and formats JavaScript, the tools you would install per project with npm from Chapter 6.
Remote - SSH (<code>ms-vscode-remote.remote-ssh</code>)	Opens a folder on another machine, such as <code>debserver</code> , over the SSH connection from Chapter 2.

Confirm each identifier in the extension marketplace before installing, because publishers rename things and this list is a starting point, not a guarantee. If a command reports that an extension was not found, search for it in VS Code's Extensions panel and copy the identifier from there.

EXTENSIONS CAN READ YOUR CODE

An extension runs with your permissions and can see your files. Install extensions from publishers you recognise, prefer ones with many users, and remove any you do not use. VS Code also sends usage data to Microsoft by default. If you would rather it did not, set `"telemetry.telemetryLevel": "off"` in your settings file, or search "telemetry" in the Settings screen.

Recording the list

Save the list now, and you can rebuild this part of the machine with one command later:

```
# Save
code --list-extensions > ~/vscode-extensions.txt
```

```
# Restore on a fresh machine
xargs -L1 code --install-extension < ~/vscode-extensions.txt
```

Choosing a Default Editor

Several programs, Git among them, open “the editor” when they need input from you. If that is not the editor you want, you will meet an unfamiliar interface at an awkward moment. Set it once. Debian keeps a system-wide choice, and you can override it for yourself:

```
# System-wide default (asks you to pick from installed editors)
sudo update-alternatives --config editor

# Just for your own account
select-editor

# Belt and braces: name it in your shell, and tell Git directly
echo 'export EDITOR=vim' >> ~/.bashrc
git config --global core.editor "vim"
```

Choose the editor you are most comfortable with in a terminal. If that is nano, use nano; the point is that it is a decision you made, not one that surprised you. Check it with `git config --global --get core.editor` and `echo $EDITOR`. If you would rather use VS Code for Git messages, `git config --global core.editor "code --wait"` works; the `--wait` makes Git pause until you close the file.

Which Tool for Which Job

YOU WANT TO...	USE
Work on a project	VS Code, with a few well-chosen extensions
Edit a config file, or work over SSH	Vim, Neovim or nano
Edit files on <code>debserver</code> from VS Code	Remote - SSH
Keep VS Code updated	<code>apt full-upgrade</code>
Rebuild your editor setup	<code>vscode.sources</code> and <code>vscode-extensions.txt</code>

Hands-On Exercises

EXERCISE 1

Add Microsoft's repository and install VS Code by the manual method. Before installing, inspect the key you downloaded. After installing, use the Chapter 3 tools to prove where the package came from. Explain what the `Signed-By` line does and what you have trusted.

 [View solution](#)

EXERCISE 2

Set your default editor at every level (system, user, shell and Git), then prove each setting took effect. Trigger Git's editor with a test commit and confirm the right editor opens.

 [View solution](#)

EXERCISE 3

Install three VS Code extensions from the command line, save the list, remove one extension, and restore it from the saved file. Then describe how you would use Remote - SSH to edit a file on `debserver`, and one risk of doing so.

 [View solution](#)

CHAPTER 8 QUICK REFERENCE

- Debian 13 has Vim 9.1, Neovim 0.10.4 and nano; **VS Code is not in Debian** and comes from Microsoft's repository
- Key: `wget -qO- https://packages.microsoft.com/keys/microsoft.asc | sudo gpg --dearmor -o /usr/share/keyrings/microsoft.gpg`; inspect with `gpg --show-keys`
- Repository file `/etc/apt/sources.list.d/vscode.sources` with `Signed-By: /usr/share/keyrings/microsoft.gpg`
- `sudo apt update && sudo apt install code`; afterwards `apt full-upgrade` keeps it current (repository can lag up to three hours)
- A third-party repository means trusting that publisher; keep them to the ones you need

- Extensions: `code --install-extension ID`, `code --list-extensions`; verify IDs in the marketplace
- Save and restore: `code --list-extensions > file`; `xargs -L1 code --install-extension < file`
- Default editor: `sudo update-alternatives --config editor`, `select-editor`, `EDITOR`, `git config --global core.editor`
- Remote - SSH edits files on another machine over SSH; be careful when the machine is `debserver`

Debian Development Machine Setup

Chapter 9 · Self-Hosting Git with Gitea

Chapter 2 installed the Git *client*. This chapter installs a Git *server*: a place on your own network where repositories live, with a web page for browsing them, issues, pull requests and user accounts. We use **Gitea**, a lightweight self-hosted Git service that ships as a single program. Git and GitHub Foundations teaches the Git commands themselves; here the job is running the service properly on Debian.

WHY BOTHER, WHEN GITHUB EXISTS?

Your code stays on machines you control, there is no account or limit to worry about, and you learn to run a real service: a dedicated user, a systemd unit, a database, backups. Those skills carry over to everything else you will host. The cost is that *you* are now responsible for updates and backups, which is why Chapter 10 covers backing this up.

Check First

```
command -v gitea
git --version
getent passwd git
ss -tlnp | grep -E ':3000|:22'
```

You want no `gitea` command, a Git of at least 2.0 (Gitea's documented minimum; Debian 13's is far newer), no existing `git` user, and nothing else using the ports. If `getent` prints a line, a `git` user already exists and you should look at it before continuing.

Choosing a Database

Gitea stores users, issues and settings in a database, while the repositories themselves are ordinary directories on disk. Chapter 7 gave you two candidates.

CHOICE	FOR	AGAINST
SQLite	Nothing extra to run or secure; one file to back up; ideal for one person or a small team.	Gitea's own documentation warns it does not scale, so choose something else if the instance may grow.
MariaDB	Already installed from Chapter 7; copes with more users; a familiar tool.	One more service that Gitea needs running before it starts, and one more thing to back up.

For a personal development machine, **SQLite is the sensible default**, and this chapter uses it. The MariaDB steps are given afterwards in case you prefer them. Gitea's documentation lists MariaDB 10.4 or newer as supported, and Debian 13 has 11.8.

Installing Gitea from the Binary

Gitea publishes a ready-built program for Linux, which is the installation method used here. Everything below follows Gitea's official binary-installation guide. Read the version number on the [download page](#) first; the examples use 1.27.3, which was the version in the documentation when this chapter was written, so substitute the current one.

Step 1: Download and verify

```
VER=1.27.3
cd ~/Downloads
wget -O gitea https://dl.gitea.com/gitea/$VER/gitea-$VER-linux-amd64
wget -O gitea.asc https://dl.gitea.com/gitea/$VER/gitea-$VER-linux-amd64.asc
chmod +x gitea

gpg --keyserver htps://keys.openpgp.org --recv 7C9E68152594688862D62AF62D9AE806EC1592E2
gpg --verify gitea.asc gitea
```

Look for `Good signature from "Teabot <teabot@gitea.io>"`. This is the same verify-before-trust habit as Chapter 1 and Chapter 8: you are about to run a downloaded program as a service, so confirm it is the one the project signed. A warning that the key is not certified with a trusted signature is normal for a key you have just fetched; a *bad* signature is not, and means you should delete the file and download again.

Step 2: A dedicated user

```
sudo adduser --system --shell /bin/bash --gecos 'Git Version Control' \
  --group --disabled-password --home /home/git git
```

This creates a `git` account that cannot log in with a password and exists only to own Gitea's files and run its process. If Gitea is ever compromised, the attacker has that account's permissions and not yours.

Step 3: Directories and the program

```
sudo mkdir -p /var/lib/gitea/{custom,data,log}
sudo chown -R git:git /var/lib/gitea/
sudo chmod -R 750 /var/lib/gitea/
sudo mkdir /etc/gitea
sudo chown root:git /etc/gitea
sudo chmod 770 /etc/gitea

sudo cp gitea /usr/local/bin/gitea
```

Data lives under `/var/lib/gitea`, configuration under `/etc/gitea`, and the program in `/usr/local/bin`, the place Chapter 3 described for software that did not come from apt. Note that apt knows nothing about this installation, so **apt will never update it**; upgrades are a manual job you will do in Chapter 10's routine.

THE CONFIG DIRECTORY IS DELIBERATELY LOOSE, FOR NOW

`/etc/gitea` is writable by the `git` group at this point so the web installer can save its settings. Once setup is finished you tighten it (below). Do not leave it at 770.

Step 4: The systemd service

Gitea provides a sample unit file. Create `/etc/systemd/system/gitea.service`, keeping to the project's sample and enabling only what you need:

```
[Unit]
Description=Gitea (Git with a cup of tea)
After=network.target
# If you choose MariaDB, uncomment these two lines so it starts first:
```

```
#Wants=mariadb.service
#After=mariadb.service

[Service]
RestartSec=2s
Type=simple
User=git
Group=git
WorkingDirectory=/var/lib/gitea/
ExecStart=/usr/local/bin/gitea web --config /etc/gitea/app.ini
Restart=always
Environment=USER=git HOME=/home/git GITEA_WORK_DIR=/var/lib/gitea

[Install]
WantedBy=multi-user.target
```

Read it as three facts: it runs as `git`, not root; it starts Gitea with the configuration file in `/etc/gitea`; and `Restart=always` brings it back if it crashes. The commented lines matter only for a database that runs as its own service.

```
sudo systemctl enable gitea --now
systemctl status gitea
journalctl -u gitea -n 30
ss -tlnp | grep gitea
```

`enable --now` starts it immediately and at every boot. The status and journal commands are the same ones you would use for any service (systemd in Depth covers them fully), and `ss` shows which address and port it is listening on.

First-Run Setup

Open the web address of the machine in your browser on port 3000 (Gitea's own documentation refers to port 3000; use whatever `ss` shows if yours differs), for example `http://localhost:3000`. The first visit shows an installation page. The fields that matter:

SETTING	SUGGESTED VALUE
Database type	SQLite3 (the path is offered for you), or MySQL for MariaDB with the details below

SETTING	SUGGESTED VALUE
Server domain	<code>devserver</code> (or its IP address, which is more reliable if names do not resolve on your network)
Base URL	<code>http://devserver:3000/</code> , matching how you will actually reach it, because Gitea builds clone links from this
Administrator account	Create one here, with a strong password. If you skip it, the first account you register becomes the administrator.

IF YOU CHOOSE MARIADB

Create the database and user before the installer, using the statements from Gitea's documentation, with the user restricted to `localhost` (Chapter 7's habit) rather than the `'%'` wildcard the documentation's example uses for remote access:

```
CREATE USER 'gitea'@'localhost' IDENTIFIED BY 'choose-a-password';
CREATE DATABASE giteadb CHARACTER SET 'utf8mb4' COLLATE 'utf8mb4_bin';
GRANT ALL PRIVILEGES ON giteadb.* TO 'gitea'@'localhost';
FLUSH PRIVILEGES;
```

In the installer choose MySQL, host `127.0.0.1:3306`, and those names. Uncomment the two `mariadb.service` lines in the unit file.

Lock the configuration down

Once the installer has finished and you can see the Gitea home page, tighten the permissions as the official guide advises:

```
sudo chmod 750 /etc/gitea
sudo chmod 640 /etc/gitea/app.ini
sudo systemctl restart gitea
```

`app.ini` contains secrets (and a database password if you used MariaDB), so it should be readable only by root and the `git` group.

WHO CAN REACH IT?

Gitea listens for web connections on a port anyone on your network can try. Check with `ss -tlnp | grep gitea`: if it shows `127.0.0.1:3000` it is reachable only from this machine; `0.0.0.0:3000` or

`*:3000` means the whole network. That is fine on a home network you trust, and worth a second thought anywhere else. Also disable open registration (Site Administration → Configuration, or the installer's optional settings) unless you want strangers making accounts. HTTPS in front of it is outside this course; the HTTPS/TLS Fundamentals and Nginx In Depth courses cover that.

Working With It

SSH keys

Pushing over SSH avoids typing a password each time. In the web page open your profile → Settings → SSH / GPG Keys, and paste the public key (`~/.ssh/id_ed25519.pub`, the pair you made in Chapter 2). Gitea's documentation describes a built-in SSH server and also the option of using the system's OpenSSH; because Chapter 2 already runs OpenSSH, ask the installer page about SSH settings and read what it offers before changing anything, and keep to the default it proposes. HTTP pushing, described next, needs none of this and is the simplest way to start.

Create a repository and push an existing project

Create an empty repository in the web page (no README, so the histories do not clash), then:

```
cd ~/projects/composer-demo
git status                # is it already a repository?
git branch -M main
git remote add origin http://devserver:3000/YOURNAME/composer-demo.git
git remote -v
git push -u origin main
```

`git remote add` records where the server is; `push -u` uploads and remembers the link so later a plain `git push` works. For a project that already has a GitHub remote, use `git remote rename origin github` first, or use a second name such as `gitea`, so both remain available. Remember the rule from earlier chapters: commit `.gitignore`, never `venv/`, `node_modules/` or `vendor/`.

Using it from other machines

Clone URLs use the Base URL you entered, so the name must resolve from the other computer. If `devserver` cannot find `devserver`, use the IP address or add a line to its `/etc/hosts`. Then `git clone http://devserver:3000/YOURNAME/project.git` works from there, and VS Code from Chapter 8 can use the repository like any other remote.

Keeping It Healthy

TASK	HOW
Is it running?	<code>systemctl status gitea</code>
What went wrong?	<code>journalctl -u gitea</code> , plus the log directory under <code>/var/lib/gitea/log</code>
Restart after a config change	<code>sudo systemctl restart gitea</code>
Upgrade	Download and verify the new binary, stop the service, copy it over <code>/usr/local/bin/gitea</code> , start it. Back up first.
Back up	Chapter 10. Gitea has its own dump command; check <code>gitea dump --help</code> for the options in your version.

Hands-On Exercises

EXERCISE 1

Install Gitea from the binary: check first, download and verify the signature, create the `git` user and directories, install the systemd unit, and start it. Show three separate checks that it is running as the `git` user and listening where you expect.

 [View solution](#)

EXERCISE 2

Complete first-run setup with SQLite, tighten the permissions on `/etc/gitea` and `app.ini`, and confirm the service still works afterwards. Explain why the tightening is done after setup and not before.

 [View solution](#)

EXERCISE 3

Push one of your earlier projects (the Composer demo, or a small Python or Node project) to a new Gitea repository. Then clone it into a second directory and prove the copy is identical and that the ignored folders were not uploaded.

CHAPTER 9 QUICK REFERENCE

- Gitea = self-hosted Git service; repositories are directories, everything else is in a database
- **SQLite** for a personal machine; MariaDB (11.8 on Trixie, 10.4+ supported) if you prefer
- Download the binary, verify with `gpg --verify` (key `7C9E68152594688862D62AF62D9AE806EC1592E2`), copy to `/usr/local/bin/gitea`
- Dedicated `git` user via `adduser --system ... git`; data in `/var/lib/gitea`, config in `/etc/gitea`
- Unit file `/etc/systemd/system/gitea.service`; `systemctl enable gitea --now`
- First run at port 3000: set Base URL to how you reach the machine; then `chmod 750 /etc/gitea` and `chmod 640 /etc/gitea/app.ini`
- Not installed by apt, so apt never updates it: upgrades are manual
- Push: `git remote add origin URL`, `git push -u origin main`
- Check who can reach it with `ss -tlnp | grep gitea`; consider disabling open registration

Debian Development Machine Setup

Chapter 10 · Capstone: A Reproducible Dev Machine

You now have a working `devserver`: Debian 13, a graphics driver, Python, Node.js, PHP, MariaDB, editors, and your own Git server. The last question is the one that decides whether all that work is safe: **if the disk died tomorrow, how long would it take to get back here?** This chapter turns what you built into something you can repeat and recover: a list of what is installed, a script that rebuilds it, your settings stored in Git, and backups you have actually tested.

THE IDEA RUNNING THROUGH THE WHOLE COURSE

Every chapter ended with a record rather than a memory: a lock file for a project, a version pinned in a file, a list of extensions. A machine is the same. What you can rebuild from a file is safe; what only exists in your head or on one disk is not.

Step 1: Audit What Is There

Start from facts, using the checking habits from Chapter 3. Keep everything in one folder so it can go into a Git repository:

```
mkdir -p ~/machine-setup && cd ~/machine-setup

# Packages you (or the installer) chose, not their dependencies
apt-mark showmanual | sort > packages-manual.txt

# Versions of the main tools
{ lsb_release -ds; uname -r; python3 --version; php --version | head -1;
  composer --version; mariadb --version; git --version;
  code --version | head -1; nvidia-smi --query-gpu=driver_version --format=csv,noheader;

# Things that did not come from apt
pipx list > pipx.txt
code --list-extensions > vscode-extensions.txt
ls ~/.nvm ~/.pyenv /usr/local/bin > non-apt.txt 2>&1
```

Node's version is not in that list because it comes from `nvm`, so record it separately: `.`
`~/ .nvm/nvm.sh && nvm current`. Read each file. `packages-manual.txt` is longer than you might expect, because the Debian installer and the desktop environment marked many packages as manual too; that is normal, and it is why the script below installs a deliberate list of your own instead of replaying the whole file.

DO NOT REPLAY THE WHOLE LIST BLINDLY

Feeding `packages-manual.txt` to `apt install` on a different install can fail, because packages get renamed or removed between releases, and it drags in installer choices you did not make on purpose. Use it as a *reference* to build your own short list, and to spot things you forgot you had.

Fill in the server side

Chapter 7 promised a comparison with `debserver`. Run the same version commands there and keep both files:

```
ssh debserver 'lsb_release -ds; php --version | head -1; mysql --version || mariadb --ve
```

When `debserver` is reinstalled with Debian 13, run it again and the two files should agree.

Step 2: A Provisioning Script

A provisioning script is a shell script that does the setup for you. The important property is that it is **safe to run twice**: each step checks whether the work is already done, exactly the check-before-change habit from Chapter 3, using the same `have` and `pkg_installed` helpers. Run it as your normal user, not as root; it calls `sudo` where it needs to.

```
#!/usr/bin/env bash
# bootstrap.sh - rebuild devserver's software on a fresh Debian 13 install.
# Run as your normal user. Safe to run again: every step checks first.
set -euo pipefail

have() { command -v "$1" >/dev/null 2>&1; }
pkg_installed() { dpkg-query -W -f='${Status}' "$1" 2>/dev/null | grep -q 'install ok in
apt_install() {
```

```

local missing=()
for p in "$@"; do pkg_installed "$p" || missing+=("$p"); done
if [ "${#missing[@]}" -gt 0 ]; then
    sudo apt-get install -y "${missing[@]}"
else
    echo "already installed: $"
fi
}

echo "== 1. Sources: add contrib and non-free (needed for the NVIDIA driver) =="
SRC=/etc/apt/sources.list.d/debian.sources
if ! grep -Eq '^Components:.*[[:space:]]non-free([[:space:]]|$)' "$SRC"; then
    sudo sed -i '/^Components:\/ s$/ contrib non-free/' "$SRC" # check the result afterwards
fi
sudo apt-get update

echo "== 2. Everyday tools =="
apt_install build-essential curl wget git vim htop tree unzip zip jq ripgrep \
    ca-certificates gnupg pciutils

echo "== 3. NVIDIA driver =="
apt_install linux-headers-amd64 nvidia-kernel-dkms nvidia-driver nvidia-smi
# Secure Boot: the MOK enrolment from Chapter 4 is still a manual step.

echo "== 4. Python =="
apt_install python3-pip python3-venv python3-dev pipx

echo "== 5. PHP, Composer and databases (php-cli, never the 'php' metapackage) =="
apt_install php-cli php-mbstring php-xml php-curl php-zip php-intl php-mysql php-sqlite3
    composer mariadb-server sqlite3

echo "== 6. Node.js through nvm =="
if [ ! -d "$HOME/.nvm" ]; then
    git clone https://github.com/nvm-sh/nvm.git "$HOME/.nvm"
    git -C "$HOME/.nvm" checkout v0.40.8
fi
set +u; . "$HOME/.nvm/nvm.sh"; nvm install --lts; set -u
# The shell lines that load nvm at login are in Chapter 6; add them to ~/.bashrc.

echo "== 7. VS Code from Microsoft's repository =="
if ! pkg_installed code; then
    apt_install wget gpg
    if [ ! -f /usr/share/keyrings/microsoft.gpg ]; then
        wget -qO- https://packages.microsoft.com/keys/microsoft.asc | sudo gpg --dearmor -o /usr/share/keyrings/microsoft.gpg
    fi

```

```

sudo tee /etc/apt/sources.list.d/vscode.sources >/dev/null <<'EOF'
Types: deb
URIs: https://packages.microsoft.com/repos/code
Suites: stable
Components: main
Architectures: amd64,arm64,armhf
Signed-By: /usr/share/keyrings/microsoft.gpg
EOF
sudo apt-get update
apt_install code
fi
if [ -f "$HOME/machine-setup/vscode-extensions.txt" ]; then
    xargs -L1 code --install-extension < "$HOME/machine-setup/vscode-extensions.txt"
fi

echo
echo "Done. Still manual: Secure Boot key (Ch. 4), SSH keys and ssh config (Ch. 2),"
echo "pyenv (Ch. 5), Gitea install and restore (Ch. 9 and below), dotfiles."

```

Read it as a table of contents for the whole course: every section is a chapter, and every chapter's choices appear as a line. Things it deliberately does *not* do are worth noticing. It never stores a password. It does not restore Gitea, because Gitea's data comes from a backup, not from a script. And it leaves a few steps manual and says so, because a script that silently guesses (a Secure Boot key, an SSH key) is worse than one that tells you what is left.

TEST IT BEFORE YOU NEED IT

Check the syntax with `bash -n bootstrap.sh`, and run it a second time on `devserver` itself: it should print "already installed" for everything and change nothing. That proves the checks work. The real proof is a fresh Debian 13 virtual machine, if you have one to spare. Read the `sed` line's effect on `debian.sources` afterwards, as it is the one step that edits a system file.

Step 3: Your Settings, in Git

Installed software is half of a machine. The other half is your configuration: the files in your home directory whose names start with a dot. Keep the ones you care about in a repository on your own Gitea (the payoff of Chapter 9):

```

mkdir -p ~/dotfiles && cd ~/dotfiles
mv ~/.bashrc bashrc      && ln -s ~/dotfiles/bashrc ~/.bashrc
mv ~/.gitconfig gitconfig && ln -s ~/dotfiles/gitconfig ~/.gitconfig

cp -r ~/machine-setup .      # bootstrap.sh and the lists live here too
git init -b main && git add . && git commit -m "Dotfiles and machine setup"
git remote add origin http://devserver:3000/YOURNAME/dotfiles.git
git push -u origin main

# On a rebuilt machine:
# git clone http://devserver:3000/YOURNAME/dotfiles.git ~/dotfiles
# ln -sf ~/dotfiles/bashrc ~/.bashrc

```

A symbolic link means the real file lives in the repository, so editing `~/.bashrc` edits the tracked copy, and `git diff` shows what you changed.

NEVER PUT SECRETS IN THE REPOSITORY

Private SSH keys, the Gitea password, database passwords, API tokens and `.env` files do not belong in it, even on your own server. Store the list of *which* secrets you need, not the secrets. Also remember that the repository lives on `devserver` itself: a copy that shares a disk with the original is not a backup, which brings us to the next step.

Step 4: Backups You Have Tested

Decide what must survive, and where a second copy lives.

WHAT	WHERE IT LIVES	HOW TO PROTECT IT
Your files and projects	<code>/home/you</code>	Copy to another disk, leaving out what can be rebuilt (<code>node_modules</code> , virtual environments, <code>vendor</code>)
Gitea data	<code>/var/lib/gitea</code> , <code>/etc/gitea</code> , and the database if you chose MariaDB	Stop or dump consistently, then copy
Databases you use for development	MariaDB	<code>mariadb-dump</code> for the ones that matter

WHAT	WHERE IT LIVES	HOW TO PROTECT IT
Software and settings	Package lists, <code>bootstrap.sh</code> , dotfiles	In Git, and Git copied elsewhere
Secrets and keys	<code>~/.ssh</code>	Encrypted copy in a safe place, never in Git

Files

```
# Preview first: -n changes nothing and -v lists what would happen
rsync -avn --delete --exclude node_modules --exclude .venv --exclude venv --exclude vendor
~/ /mnt/backup/home/

# Then for real (remove the n)
rsync -av --delete --exclude node_modules --exclude .venv --exclude venv --exclude vendor
~/ /mnt/backup/home/
```

`--delete` makes the destination an exact mirror, including removing files you deleted. That is what you want for a mirror and dangerous when you get the two paths the wrong way round or the disk is not mounted, so always do the preview first and check that `/mnt/backup` is really your backup disk (`findmnt /mnt/backup`). If `rsync` is missing, `sudo apt install rsync`.

Gitea

Gitea's own `dump` command bundles its repositories, database and configuration into one archive. Run `/usr/local/bin/gitea dump --help` first to see the options for your version, and then run it as the `git` user so that file ownership matches the running service:

```
cd /tmp
sudo -u git /usr/local/bin/gitea dump --help
sudo -u git /usr/local/bin/gitea dump -c /etc/gitea/app.ini
ls -lh /tmp/gitea-dump-*.zip
```

If the command complains about a working directory or a permission, read its message rather than working around it; the `--help` output tells you where the archive is written. Then copy the archive to the backup disk. Keep a few dated dumps, not just the latest.

A BACKUP YOU HAVE NOT RESTORED IS A HOPE

Test on something harmless: restore a single file from the rsync copy into a scratch folder and open it, and unpack the Gitea archive somewhere temporary and look inside. A full Gitea restore onto a spare machine is the real test, and worth doing once, before the day you need it. Chapter 10's exercises walk through the small versions.

Upgrading Gitea, the manual routine

Because apt does not manage Gitea, updating it is your job. Do it in this order: **dump first**, then download the new binary and verify its signature (Chapter 9), then `sudo systemctl stop gitea`, copy the new file over `/usr/local/bin/gitea`, `sudo systemctl start gitea`, and check `journalctl -u gitea -n 30`. Read the release notes for the versions you are skipping, since a jump across several versions can include changes you need to know about.

The Regular Routine

HOW OFTEN	DO THIS
Weekly	<code>sudo apt update && sudo apt full-upgrade</code> ; reboot if the kernel changed (NVIDIA driver check, Chapter 4); run the file backup
Monthly	Gitea dump, MariaDB dump, refresh <code>packages-manual.txt</code> and <code>vscode-extensions.txt</code> , commit and push the setup repository, look at the Gitea release page
When you install something new	Add it to <code>bootstrap.sh</code> in the same sitting
Twice a year	Restore something from backup on purpose; re-check Node's and Debian's support dates

Closing Checklist

Each line points back to the chapter that built it.

CHAPTER	YOU CAN NOW...
1 Planning & Installing	Install Debian 13 with sensible choices, and know what you chose

CHAPTER	YOU CAN NOW...
2 First Boot	Keep the system updated, use sudo, log in over SSH with keys, and tell <code>devserver</code> from <code>debserver</code> at a glance
3 What's Installed	Check whether something is installed, which version, and where it came from
4 Graphics Drivers	Install and verify the NVIDIA driver, handle Secure Boot, and survive kernel updates
5 Python	Use venvs, pipx and pyenv instead of fighting the system Python
6 Node.js	Run several Node versions with nvm and keep project and global installs apart
7 PHP & Databases	Install PHP without a web server, use Composer, run MariaDB with a limited user, and compare with the server
8 Editors	Install VS Code safely, set a default editor, and record extensions
9 Gitea	Run a Git server as a proper service and push to it
10 This chapter	Rebuild the machine from a script, keep your settings in Git, and restore from a backup you have tested

WHERE TO GO NEXT

Docker for Beginners is the natural next addition to this machine, and it would go into `bootstrap.sh` as an eleventh section. When `debserver` is reinstalled with Debian 13, much of this course, and this very script, applies to it too, with the web server courses (Setting Up a Web Server on Debian, and Nginx In Depth) covering the parts that differ.

Hands-On Exercises

EXERCISE 1

Run the audit: create `~/machine-setup` with the package list, versions, extension list and non-apt tool list, plus the `debserver` versions file. Then find three packages in `packages-manual.txt` that you did not install on purpose, and say what each is.

 View solution

EXERCISE 2

Save the provisioning script as `bootstrap.sh`, syntax-check it, and run it twice on `devserver`. Show from the output that the second run changed nothing. Then add one new tool of your choice to it, correctly, using the helper functions.

 [View solution](#)

EXERCISE 3

Put your dotfiles and setup folder into a Gitea repository. Take a file backup with an `rsync` preview first and then for real, and a Gitea dump. Prove both work by restoring one file from the `rsync` copy and listing the contents of the Gitea archive.

 [View solution](#)

CHAPTER 10 QUICK REFERENCE

- Audit: `apt-mark showmanual`, version commands, `pipx list`, `code --list-extensions`, `nvm current`
- The manual-package list is a **reference**, not something to replay blindly
- A provisioning script is **re-runnable**: every step checks first (`have`, `pkg_installed`)
- Install `php-cli` plus extensions, never the `php` metapackage; add `contrib non-free` before the NVIDIA packages
- The script never holds passwords and leaves risky steps (Secure Boot, SSH keys, Gitea restore) manual
- Dotfiles: real files in a Git repository, symbolic links from the home directory, no secrets
- Backups: `rsync -avn` preview then run; `gitea dump` (check `--help`); a copy on the same disk is not a backup
- Upgrade Gitea by hand: dump, verify, stop, copy, start, check the journal
- A backup is only real once you have restored from it