LINUX SYSTEMS SERIES

Linux Package Managers

APT and Debian/Raspbian, RPM and RedHat/Fedora side by side, troubleshooting, snap/flatpak, and practical maintenance workflows. 10 complete chapters.

10 Chapters

osztromok.com

CHAPTER 1: PACKAGE MANAGEMENT CONCEPTS

CHAPTER 1

Package Management Concepts

What a package manager actually does, package vs source installation, and how dependency resolution works under the hood

Every piece of software installed on Linux beyond what shipped with the OS has to get onto the system somehow — package managers are the tool that handles that, consistently and safely, across the entire system. This course covers APT (Debian/Raspbian — Philip's actual machines) and RPM-based tools (RedHat/Fedora) side by side, but this chapter starts with the concepts shared by both, and most other package managers too.

What a Package Manager Actually Does

📦 Installs software

Downloads a pre-built package and places its files in the correct system locations — binaries, libraries, config files, documentation — all in one coordinated operation.

🔗 Resolves dependencies

Automatically identifies and installs whatever else a package needs to actually function — covered in depth below.

📋 Tracks what's installed

Maintains a database of every installed package, its version, and exactly which files belong to it — the foundation of clean removal and upgrades.

⬆️ Handles updates

Checks for newer versions and applies them, ideally without breaking whatever already depends on the older version.

🗑️ Enables clean removal

Because it tracked every file a package installed, removal can delete exactly those files — not a guess, a precise, reversible record.

Package Install vs Source Install

📦 Installing a Package

A pre-built, pre-compiled bundle (a `.deb` or `.rpm` file) — software ready to run immediately, tracked by the package manager's database, dependency-checked automatically, trivially removable later.

🛠️ Building from Source

Downloading raw source code and compiling it yourself (`./configure && make && make install`) — full control over build options, but the package manager has no idea this software exists, no automatic dependency tracking, and no clean way to uninstall it later.

SOFTWARE INSTALLED FROM SOURCE BECOMES GENUINELY INVISIBLE TO YOUR PACKAGE MANAGER

Nothing installed via `make install` shows up when checking installed packages, gets flagged for security updates, or can be removed with a simple uninstall command — it's just files copied onto the filesystem with no tracking at all. This is exactly why "compile it yourself" should be a last resort, reserved for software genuinely unavailable as a package, not a default habit.

Dependency Resolution — The Core Problem Package Managers Solve

Almost no piece of software is fully self-contained — it relies on shared libraries, other tools, specific versions of both. Installing something with unmet dependencies either fails outright or, worse, installs something broken that fails the moment it's actually run.

your-app declares exactly which other packages, and which minimum versions, it needs to actually run

How Resolution Actually Works

1. **Read the target package's declared dependencies** — every package includes metadata listing exactly what it needs.
2. **Check what's already installed and at what version.**
3. **Calculate the full set of packages needed** — including dependencies of dependencies, recursively, until everything is satisfied.

4. **Present the full install/upgrade plan** — most package managers show this list before actually proceeding, letting you review what's about to happen.

```
$ # A real apt example — note it lists everything that will actually be pulled in
$ sudo apt install nginx
The following additional packages will be installed:
libnginx-mod-http-geoip2 nginx-common nginx-core
0 upgraded, 4 newly installed, 0 to remove
```

Dependency Conflicts — When Resolution Can't Find a Clean Answer

Sometimes two packages require genuinely incompatible versions of the same shared dependency — package managers detect this and refuse to proceed with a silently broken install, instead reporting the conflict explicitly so it can be resolved deliberately (often by choosing one of the conflicting packages, or finding an alternative).

> **"DEPENDENCY HELL" IS A REAL, HISTORICAL TERM FOR A REAL, MOSTLY-SOLVED PROBLEM**
>
> Before modern package managers with proper automated resolution became standard, manually tracking down and installing the correct version of every dependency by hand — and the cascading conflicts that followed — was genuinely painful enough to earn that specific nickname. Modern apt/dnf largely automate this away, which is exactly why this course exists: understanding the tool that quietly solves a problem that used to be a major source of real frustration.

The Two Families This Course Covers

- **Debian-based (APT)** — Debian, Ubuntu, Raspbian/Raspberry Pi OS. Package format: `.deb`. Philip's actual Debian server and Raspberry Pi both live in this family — covered in depth across Chapters 2-4.
- **RedHat-based (RPM/dnf)** — Fedora, RHEL, CentOS/Rocky/Alma. Package format: `.rpm`. Covered for comparison and general understanding in Chapters 5-7, even without day-to-day use on these specific systems.

CHAPTER 1 QUICK REFERENCE

- **A package manager** installs, tracks, updates, and cleanly removes software — maintaining a real database of what's installed and which files belong to it
- **Package install** — pre-built, tracked, dependency-checked, cleanly removable
- **Source install** — full control, but completely invisible to the package manager — no tracking, no clean removal
- **Dependency resolution** — reads declared requirements, checks what's installed, calculates the full set needed, presents the plan before acting
- **Dependency conflicts** are reported explicitly rather than silently producing a broken install
- **This course covers:** APT/Debian family (Ch2-4, Philip's actual machines) and RPM/RedHat family (Ch5-7, for comparison)
- **Next chapter:** APT in depth — apt vs apt-get vs dpkg, installing/removing/upgrading

CHAPTER 2: APT IN DEPTH

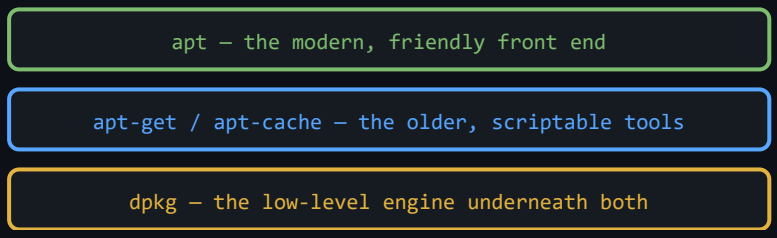
CHAPTER 2

APT in Depth

apt vs apt-get vs dpkg — three layered tools, and the everyday commands for installing, removing, and upgrading on Debian/Raspbian

This is the chapter most directly relevant to Philip's own machines — Debian on the server, Raspberry Pi OS (Raspbian) on the Pi. Three different tools all do package management on these systems, and understanding how they relate avoids real confusion about which one to actually reach for.

The Three Layers



Each layer calls down into the one below it — apt and apt-get both ultimately use dpkg to actually install files

apt

Introduced specifically to give a cleaner, more consistent command set for everyday interactive use — a progress bar, colour output, and sensible defaults. The right default choice for almost everything in this chapter.

apt-get / apt-cache

The original tools, still fully functional and more commonly seen in older scripts and documentation — slightly more stable output formatting, which is why scripts sometimes still prefer apt-get over apt deliberately.

dpkg

Works directly with individual .deb files and the local package database — no repository awareness, no dependency resolution of its own. Genuinely useful for specific low-level tasks, covered in Chapter 4.

FOR INTERACTIVE, EVERYDAY USE, JUST USE APT — FOR SCRIPTS, APT-GET'S OUTPUT IS CONSIDERED MORE STABLE

The apt manual page itself notes that apt's command-line interface is allowed to change between versions in ways that could break a script depending on exact output formatting — apt-get's interface is guaranteed stable, which is why automated scripts conventionally still use apt-get even though apt is the better choice for a human typing commands directly.

Everyday Commands

```
$ # Refresh the local list of available packages and versions from configured repos
$ sudo apt update

$ # Install a package (resolves and installs dependencies automatically)
$ sudo apt install nginx

$ # Remove a package, keeping its configuration files
$ sudo apt remove nginx

$ # Remove a package AND its configuration files
$ sudo apt purge nginx

$ # Upgrade every installed package to its latest available version
$ sudo apt upgrade

$ # Search for a package by name or description
$ apt search "web server"

$ # Show detailed information about a package before installing
$ apt show nginx
```

APT UPDATE DOES NOT INSTALL ANYTHING — IT'S COMMONLY CONFUSED WITH UPGRADE

`apt update` only refreshes the local cache of what's available in the configured repositories — it changes nothing about what's actually installed on the system. `apt upgrade` is the command that actually applies new versions. The conventional, near-universal pairing is `sudo apt update && sudo apt upgrade` — update first so upgrade has current information to work from.

remove vs purge — A Distinction Worth Understanding

- **apt remove** — uninstalls the software itself, but deliberately leaves configuration files behind (typically in `/etc/`) — useful if you might reinstall later and want your settings preserved.
- **apt purge** — removes the software AND every configuration file associated with it — the genuinely complete uninstall, for when you're certain you won't need the package or its settings again.

autoremove — Cleaning Up Orphaned Dependencies

```
$ sudo apt autoremove
The following packages will be REMOVED:
libold-dependency1
```

When a package is removed, any dependency that was installed automatically *just* for it — and isn't needed by anything else still installed — becomes an "orphan." `autoremove` identifies and cleans these up, preventing a slow accumulation of unused packages over months of installing and removing software.

Full vs Minimal Upgrades

COMMAND	WHAT IT DOES
`apt upgrade`	Upgrades packages, but won't remove an existing package to satisfy a new dependency requirement
`apt full-upgrade`	Will remove packages if genuinely necessary to complete an upgrade — used for major version transitions
`apt dist-upgrade`	The older name for full-upgrade — same behaviour, still widely seen in documentation

CHAPTER 2 QUICK REFERENCE

- **apt** — the modern, friendly default for everyday interactive use
- **apt-get** — older, output-stable, conventionally still preferred in scripts
- **dpkg** — the low-level engine both call into; works with individual .deb files directly
- **apt update** — refreshes the package list only, installs/changes nothing; always pair with upgrade
- **apt install/remove/purge/upgrade** — the core everyday verbs
- **remove vs purge:** remove keeps config files; purge removes everything, including config
- **apt autoremove** — cleans up orphaned dependencies left behind by removed packages
- **apt full-upgrade/dist-upgrade** — will remove packages if necessary; plain upgrade never will
- **Next chapter:** managing repositories on Debian-based systems — sources.list, third-party repos, apt keys

CHAPTER 3

Managing Repositories on Debian-Based Systems

sources.list, adding third-party repositories safely, and how apt actually verifies what it downloads

Chapter 2's `apt update` refreshes the package list from somewhere — this chapter covers exactly where, how to add new sources beyond the defaults, and the verification mechanism that prevents apt from blindly trusting whatever a repository claims to offer.

sources.list — Where Repository Definitions Live

```
$ cat /etc/apt/sources.list
deb http://deb.debian.org/debian bookworm main contrib non-free
deb http://deb.debian.org/debian bookworm-updates main contrib non-free
deb http://security.debian.org/debian-security bookworm-security main contrib non-free
```

`deb`	Binary packages (deb-src would mean source packages instead)
`http://deb.debian.org/debian`	The repository's base URL
`bookworm`	The release codename — Debian 12 in this example; this is the part that changes between OS versions
`main contrib non-free`	Components — main (fully free software), contrib (free but depends on non-free), non-free (proprietary)

sources.list.d/ — The Modern, Preferred Location for Additions

Rather than editing the main `sources.list` file directly, individual `.list` files in `/etc/apt/sources.list.d/` let each added repository live in its own file — easier to remove cleanly later, and exactly the convention modern package-provided install instructions (Docker, VS Code, and similar) typically use.

Adding a Third-Party Repository

```
$ # Modern approach — a dedicated file in sources.list.d/
$ echo "deb https://example.com/debian stable main" | sudo tee
/etc/apt/sources.list.d/example.list
$ sudo apt update
```

How apt Verifies What It Downloads — GPG Keys

Without verification, apt would have no way to confirm a downloaded package genuinely came from the repository it claims to, or that it hasn't been tampered with in transit — exactly the kind of supply-chain risk this mechanism exists to prevent.

apt only proceeds if the signature genuinely matches a public key it's been told to trust

```
$ # The modern, recommended way to add a repository's signing key
$ curl -fsSL https://example.com/key.gpg | sudo gpg --dearmor -o
/usr/share/keyrings/example.gpg

$ # Reference that specific key in the repository's source entry
$ echo "deb [signed-by=/usr/share/keyrings/example.gpg] https://example.com/debian
stable main" | sudo tee /etc/apt/sources.list.d/example.list
```

APT-KEY ADD IS DEPRECATED — SIGNED-BY IS THE MODERN REPLACEMENT

Older tutorials commonly show `sudo apt-key add key.gpg`, which adds a key trusted system-wide for *every* repository — genuinely too broad, and deprecated as of recent Debian/Ubuntu versions specifically for that reason. The `signed-by` approach shown above scopes a key to exactly the one repository that needs it, which is the correct, current practice.

PPAs — Ubuntu's Personal Package Archives

```
$ # Ubuntu-specific convenience tool for adding a PPA
$ sudo add-apt-repository ppa:someuser/somerepo
$ sudo apt update
```

PPAs are an Ubuntu-specific convention (not available on plain Debian or Raspberry Pi OS without extra setup) — individually maintained repositories, often for software not yet packaged in the official repos, or for newer versions than the official ones carry. `add-apt-repository` handles both adding the source entry and the signing key in one step.

ADDING A THIRD-PARTY REPOSITORY MEANS TRUSTING THAT MAINTAINER WITH ROOT ACCESS TO YOUR SYSTEM

Every package installed via apt typically runs install scripts with root privileges — a malicious or compromised third-party repository can run essentially anything on your machine. Only add repositories from sources you genuinely trust, and prefer official project-provided repositories over random PPAs or unofficial mirrors whenever possible.

Pinning — Controlling Which Repository Wins When Multiple Provide the Same Package

With multiple repositories configured, more than one might offer a package with the same name — APT pinning (`/etc/apt/preferences.d/`) lets you explicitly prioritise one source over another for specific packages, rather than relying purely on version-number comparison.

CHAPTER 3 QUICK REFERENCE

- **/etc/apt/sources.list** — the main repository definition file; codename (e.g. bookworm) is the part that changes per release
- **/etc/apt/sources.list.d/** — the modern, preferred place to add individual repositories in their own file
- **GPG signing** — verifies a package genuinely came from the claimed repository, unmodified
- **signed-by=** in the source line — the modern, properly scoped replacement for deprecated apt-key add
- **PPAs** — Ubuntu-specific personal repositories; add-apt-repository handles source + key in one step
- **Trusting a repository means trusting it with root access** — only add sources you genuinely trust
- **Pinning** — controls which repository wins when multiple provide the same package

- **Next chapter:** dpkg deep dive — querying installed packages, package contents, manual install/repair

CHAPTER 4: DPKG DEEP DIVE

CHAPTER 4

dpkg Deep Dive

Querying what's installed, inspecting package contents, and the manual install/repair operations apt alone can't do

Chapter 2 introduced dpkg as the low-level engine underneath apt. This chapter covers using it directly — genuinely useful for specific diagnostic and repair tasks that apt's higher-level interface doesn't expose at all.

Querying Installed Packages

```
$ # List every installed package
$ dpkg -l
ii nginx 1.22.1-9 amd64 small, powerful, scalable web server
ii openssh-server 1:9.2p1-2 amd64 secure shell (SSH) server

$ # Check if one specific package is installed
$ dpkg -l | grep nginx

$ # Detailed status of one specific package
$ dpkg -s nginx
Status: install ok installed
Version: 1.22.1-9
```

Decoding the Status Column

ii
Installed, correctly configured

rc
Removed, config files remain (Chapter 2's remove vs purge)

un
Unknown — never installed

iU
Installed but the configuration step failed — needs attention

Inspecting What Files a Package Actually Owns

```
$ # List every file installed by a package
$ dpkg -L nginx
/etc/nginx
/etc/nginx/nginx.conf
/usr/sbin/nginx

$ # The reverse — which package owns a specific file
$ dpkg -S /usr/sbin/nginx
nginx-core: /usr/sbin/nginx
```

DPKG -S IS GENUINELY USEFUL FOR "WHAT INSTALLED THIS FILE?" MYSTERIES

Tracking down which package is responsible for a specific binary or config file on a system you didn't fully set up yourself — or just confirming whether a file is genuinely part of a package or was placed there manually — is exactly what `dpkg -S` answers directly, rather than guessing.

Working with .deb Files Directly

```
$ # Install a standalone .deb file (e.g. downloaded directly from a vendor's site)
$ sudo dpkg -i some-package.deb

$ # If it fails due to missing dependencies, apt can resolve and finish the job
$ sudo apt --fix-broken install
```

DPKG -I DOES NOT RESOLVE DEPENDENCIES — THAT'S THE WHOLE REASON APT EXISTS ON TOP OF IT

Installing a .deb file directly with dpkg often fails partway through with unmet dependency errors, since dpkg has no repository awareness or dependency-resolution logic at all (Chapter 1) — that's exactly the gap apt fills. The standard recovery, shown above, hands the partially-installed state to apt, which can then pull in whatever dependencies are missing and complete the installation correctly.

Repairing a Broken Package State

COMMAND	USE WHEN
`sudo dpkg --configure -a`	A package was interrupted mid-install (e.g. power loss) and is stuck "half-configured"
`sudo apt --fix-broken install`	Dependencies are unmet or inconsistent after a manual dpkg -i
`sudo dpkg --remove --force-remove-reinstreq <pkg>`	A genuinely stuck package refuses normal removal — last resort, use cautiously

```
$ # The classic recovery sequence after an interrupted install
$ sudo dpkg --configure -a
$ sudo apt --fix-broken install
$ sudo apt update && sudo apt upgrade
```

Listing Files in a .deb Without Installing It

```
$ dpkg -c some-package.deb
drwxr-xr-x root/root 0 2026-01-01 ./etc/
-rw-r--r-- root/root 1234 2026-01-01 ./etc/myapp.conf
```

A genuinely useful inspection step before installing something from an unfamiliar source — seeing exactly what files a package would place, and where, without committing to the installation at all.

CHAPTER 4 QUICK REFERENCE

- **dpkg -l** — list all installed packages; **dpkg -s** — detailed status of one specific package
- **Status codes:** ii (installed OK), rc (removed, config remains), un (never installed), iU (failed configuration)
- **dpkg -L** — files a package owns; **dpkg -S** — which package owns a given file (reverse lookup)
- **dpkg -i** — installs a standalone .deb file directly, but does NOT resolve dependencies
- **apt --fix-broken install** — the standard follow-up to resolve dependencies after a dpkg -i failure
- **dpkg --configure -a** — fixes packages stuck mid-install after an interruption
- **dpkg -c** — inspect a .deb's file contents without actually installing it

- **Next chapter:** RPM-based systems — dnf/yum on RedHat/Fedora, installing/removing/upgrading

CHAPTER 5: RPM-BASED SYSTEMS

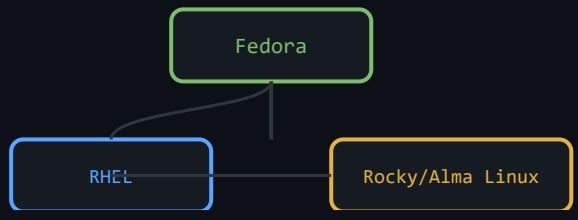
CHAPTER 5

RPM-Based Systems

dnf and yum on Fedora/RHEL — the other major package management family, covered for comparison and genuine understanding

Everything in Chapters 2-4 was Debian-family specific. Fedora, RHEL, and their derivatives use an entirely separate lineage — different package format, different tools, broadly equivalent concepts. This chapter maps the two families onto each other directly, since almost every apt concept already covered has a genuine RPM-world counterpart.

The RPM Family Tree



Fedora is the upstream, fast-moving testing ground; RHEL is the stable, commercially-supported downstream; Rocky/Alma are free, community-maintained RHEL-compatible rebuilds

The Direct Equivalents

DEBIAN/APT CONCEPT	RPM EQUIVALENT
`.deb file format`	.rpm file format
`apt (modern front end)`	dnf (modern front end)
`apt-get (older front end)`	yum (older front end — dnf is its modern replacement)
`dpkg (low-level engine)`	rpm (low-level engine)
`/etc/apt/sources.list`	/etc/yum.repos.d/*.repo files

DNF IS TO YUM EXACTLY WHAT APT IS TO APT-GET

Same relationship pattern as Chapter 2's apt/apt-get distinction — dnf is the newer, friendlier front end, yum is the older tool it replaced (and still works as an alias to dnf on modern Fedora/RHEL specifically for backward compatibility with existing scripts and habits).

Everyday Commands — Directly Mirroring Chapter 2

```
$ # Refresh package metadata (dnf does this automatically before most operations,
$ # unlike apt where update is a deliberate separate step)
$ sudo dnf check-update

$ # Install a package
$ sudo dnf install nginx

$ # Remove a package
$ sudo dnf remove nginx

$ # Upgrade everything installed
$ sudo dnf upgrade

$ # Search for a package
$ dnf search "web server"

$ # Show detailed package information
$ dnf info nginx

$ # List installed packages
$ dnf list installed
```

DNF REFRESHES METADATA AUTOMATICALLY — THERE'S NO SEPARATE, MANDATORY "UPDATE" STEP LIKE APT'S

Unlike apt, where forgetting `apt update` before `apt install` means working from a stale package list (Chapter 2's most-confused command), dnf automatically checks for fresher metadata as part of normal operations — `dnf check-update` exists mainly to preview what updates are available, not as a mandatory prerequisite step the way apt update functions.

autoremove — Same Concept, Same Name

```
$ sudo dnf autoremove
```

Identical purpose to Chapter 2's `apt autoremove` — cleans up dependencies that were only installed automatically for a now-removed package and aren't needed by anything else.

Where dnf/RPM Genuinely Differs from apt/dpkg

- **Transaction history.** `dnf history` shows every past package transaction with the ability to undo a specific one (`dnf history undo <id>`) — a more structured rollback mechanism than apt offers natively.
- **Built-in metadata refresh** as part of normal operation, rather than a separate explicit step.
- **Different repository file format** — individual `.repo` files in `/etc/yum.repos.d/`, with their own INI-style syntax, rather than apt's `deb ...` line format.

CHAPTER 5 QUICK REFERENCE

- **dnf** ↔ apt; **yum** ↔ apt-get (older, dnf is its modern replacement); **rpm** ↔ dpkg
- **.rpm** ↔ .deb file format
- **dnf install/remove/upgrade/search/info** — directly mirror the apt verbs from Chapter 2
- **dnf refreshes metadata automatically** — no separate mandatory "update" step like apt's
- **dnf autoremove** — identical concept to apt autoremove
- **dnf history / dnf history undo** — a more structured built-in rollback mechanism than apt natively offers
- **/etc/yum.repos.d/*.repo** — the RPM-world equivalent of sources.list.d/
- **Next chapter:** managing repositories on RedHat-based systems — repo files, EPEL, third-party repos

CHAPTER 6: MANAGING REPOSITORIES - REDHAT

CHAPTER 6

Managing Repositories on RedHat-Based Systems

.repo files, enabling/disabling repositories, and EPEL — the RPM world's most important third-party repository

Chapter 3 covered apt's repository system in depth. This chapter covers the RPM-world equivalent — structurally similar in purpose, genuinely different in file format and conventions.

The .repo File Format

```
$ cat /etc/yum.repos.d/fedora.repo
[fedora]
name=Fedora $releasever - $basearch
baseurl=https://download.fedoraproject.org/pub/fedora/linux/releases/$releasever/Everyth
enabled=1
gpgcheck=1
gpgkey=file:///etc/pki/rpm-gpg/RPM-GPG-KEY-fedora-$releasever-$basearch
```

`[fedora]`	The repository's ID — an INI-style section header, used to reference this specific repo in other commands
`baseurl`	Where to actually fetch packages from — equivalent to apt's repository URL
`enabled`	1 or 0 — whether dnf actually uses this repository at all
`gpgcheck + gpgkey`	The RPM-world equivalent of apt's signature verification (Chapter 3) — should essentially always be enabled

$RELEASEVER AND $BASEARCH ARE GENUINELY CONVENIENT VARIABLES

These placeholders automatically resolve to the actual OS version and CPU architecture at runtime — the same repo file works correctly across different Fedora versions or CPU architectures without needing separate, hand-edited files for each.

Enabling and Disabling Repositories

```
$ # List all configured repositories and their enabled/disabled state
$ dnf repolist all

$ # Temporarily disable a repo for one specific command
$ sudo dnf install nginx --disablerepo=fedora-updates-testing

$ # Permanently enable/disable a repo
$ sudo dnf config-manager --set-enabled fedora-updates-testing
$ sudo dnf config-manager --set-disabled fedora-updates-testing
```

EPEL — Extra Packages for Enterprise Linux

EPEL is the single most important third-party repository in the RHEL-compatible world — a community-maintained collection of additional packages, filling gaps in RHEL/Rocky/Alma's deliberately conservative default package set, broadly analogous in role and reputation to Debian's `contrib` component or a well-trusted PPA.

```
$ # Installing EPEL on RHEL/Rocky/Alma
$ sudo dnf install epel-release
$ sudo dnf update
```

> **EPEL ITSELF IS JUST INSTALLED AS A REGULAR PACKAGE, NOT CONFIGURED BY HAND**
>
> Rather than manually writing a .repo file and importing a signing key (the manual process Chapter 3 covered for apt), EPEL ships as its own package that, once installed, sets up its own repository configuration and trusted key automatically — about as close to a one-command setup as third-party repositories get in this whole course.

Adding a Generic Third-Party Repository Manually

```
$ # Many vendors provide a one-liner to add their own .repo file directly
$ sudo dnf config-manager --add-repo https://example.com/myrepo.repo

$ # Or manually create the file with the structure shown earlier in this chapter
$ sudo nano /etc/yum.repos.d/myrepo.repo
```

THE SAME TRUST PRINCIPLE FROM CHAPTER 3 APPLIES IDENTICALLY HERE

Adding any third-party RPM repository means trusting that maintainer's packages run with root privileges during installation — exactly the same risk covered for apt's third-party repos. Stick to well-known, widely-trusted sources like EPEL, and verify `gpgcheck=1` is actually set for anything else added.

COMMAND	WHAT IT DOES
`dnf repolist all`	Lists every configured repo and its enabled state
`dnf config-manager --set-enabled/--set-disabled`	Permanently toggles a repo on or off
`dnf install epel-release`	Installs and auto-configures the EPEL repository
`dnf config-manager --add-repo <url>`	Adds a .repo file from a URL directly

CHAPTER 6 QUICK REFERENCE

- **.repo files** in /etc/yum.repos.d/ — INI-style, with [id], baseurl, enabled, gpgcheck/gpgkey fields
- **$releasever/$basearch** — variables resolved automatically, making one repo file work across versions/architectures
- **dnf repolist all** — see every configured repo and its state
- **dnf config-manager --set-enabled/--set-disabled** — permanently toggle a repo
- **EPEL** — the most important third-party repo in the RHEL world; installs and configures itself via one package
- **Same trust principle as apt's third-party repos** — only add sources genuinely trusted, verify gpgcheck is enabled
- **Next chapter:** rpm deep dive — querying, verifying, manual package operations

CHAPTER 7: RPM DEEP DIVE

CHAPTER 7

rpm Deep Dive

Querying, verifying, and manual package operations — the RPM-world counterpart to Chapter 4's dpkg deep dive

Just as dpkg sits beneath apt, `rpm` sits beneath dnf — the low-level engine for direct package inspection and manual operations. This chapter mirrors Chapter 4's structure closely, since the underlying concepts are genuinely the same.

Direct dpkg ↔ rpm Command Equivalents

DPKG COMMAND (CHAPTER 4)	RPM EQUIVALENT	PURPOSE
`dpkg -l`	`rpm -qa`	List all installed packages
`dpkg -s <pkg>`	`rpm -qi <pkg>`	Detailed status/info of one package
`dpkg -L <pkg>`	`rpm -ql <pkg>`	List files a package owns
`dpkg -S <file>`	`rpm -qf <file>`	Which package owns a given file
`dpkg -i <file>.deb`	`rpm -i <file>.rpm`	Install a standalone package file directly
`dpkg -c <file>.deb`	`rpm -qlp <file>.rpm`	List a package's contents without installing it

Querying — In Practice

```
$ # List every installed package
$ rpm -qa

$ # Check if a specific package is installed
$ rpm -q nginx
nginx-1.22.1-9.fc39.x86_64

$ # Full detailed information
$ rpm -qi nginx
```

```
Name : nginx
Version : 1.22.1
License : BSD
```

Verifying Package Integrity

Beyond install-time GPG signature checks (Chapter 6), rpm can verify whether currently-installed files still match what the package originally provided — genuinely useful for detecting accidental or malicious modification after the fact.

```
$ rpm -V nginx
S.5....T. c /etc/nginx/nginx.conf
```

Each character flags a specific kind of discrepancy — `S` means file size differs, `5` means the checksum doesn't match, `T` means the modification time changed. A clean, unmodified package produces no output at all from this command.

> **RPM -V IS A GENUINE, LIGHTWEIGHT INTEGRITY-CHECK TOOL**
>
> Running this against a critical package (especially one like openssh-server or sudo) periodically, or specifically after suspecting tampering, gives a quick signal of whether anything's been modified outside the normal package-management process — a config file legitimately edited by hand will show up here too, which is expected, not necessarily alarming.

Working with Standalone .rpm Files

```
$ # Install a standalone .rpm — like dpkg, no dependency resolution of its own
$ sudo rpm -i some-package.rpm

$ # dnf's equivalent handles a local file WITH dependency resolution — generally preferred
$ sudo dnf install ./some-package.rpm
```

> **THE SAME DPKG-VS-APT RELATIONSHIP APPLIES IDENTICALLY HERE — PREFER DNF INSTALL OVER RAW RPM -I FOR LOCAL FILES**

Exactly Chapter 4's lesson, restated: `rpm -i` has no awareness of repositories or dependencies, identical to plain `dpkg -i`'s limitation. `dnf install ./file.rpm` (note the leading `./`, which tells dnf this is a local file path rather than a repository package name) gets the dependency resolution benefit while still installing a file you've downloaded directly.

Removing and Upgrading Directly with rpm

```
$ sudo rpm -e nginx # erase/remove — equivalent to dpkg --remove
$ sudo rpm -U some-package.rpm # upgrade to a newer version from a local file
```

CHAPTER 7 QUICK REFERENCE

- **rpm -qa** ↔ dpkg -l; **rpm -qi** ↔ dpkg -s; **rpm -ql** ↔ dpkg -L; **rpm -qf** ↔ dpkg -S
- **rpm -V** — verifies installed files still match what the package originally provided; flags size/checksum/time discrepancies
- **rpm -i** — installs a standalone file directly, with the same no-dependency-resolution limitation as dpkg -i
- **dnf install ./file.rpm** — preferred over raw rpm -i; gets dependency resolution for a local file
- **rpm -e** — remove; **rpm -U** — upgrade from a local file
- **Same underlying relationship as dpkg/apt** — rpm is the low-level engine, dnf is the dependency-aware front end
- **Next chapter:** checking dependencies & troubleshooting broken packages across both families

CHAPTER 8

Checking Dependencies & Troubleshooting Broken Packages

Diagnosing and fixing the most common real-world package management failures, on both Debian and RedHat-based systems

Every concept from Chapters 1-7 occasionally collides with reality — interrupted installs, conflicting dependencies, partially-removed packages. This chapter is the practical troubleshooting reference for when something genuinely goes wrong, covering both families side by side.

Common Failure Patterns

Interrupted install (power loss, killed terminal)

A package is left "half-configured" — installed but never finished its setup scripts.

`apt: dpkg --configure -a · dnf: dnf install --skip-broken, or rpm verification`

Unmet dependencies

A package needs something not currently available — wrong version installed, or no source provides it at all.

`apt: apt --fix-broken install · dnf: dnf install (auto-resolves) or check repo availability`

Held/conflicting packages

Two packages require incompatible versions of the same shared dependency — neither side can install cleanly.

`Identify the conflict explicitly, choose one side, or find an alternative package`

Locked package database

Another package manager process is still running, or crashed leaving a stale lock file.

`Wait for genuine processes to finish; only remove a stale lock if truly certain nothing is using it`

Debian/APT Troubleshooting in Practice

```
$ # Symptom: "dpkg was interrupted, you must manually run 'dpkg --configure -a'"
$ sudo dpkg --configure -a

$ # Symptom: "The following packages have unmet dependencies"
$ sudo apt --fix-broken install

$ # Symptom: held back, won't upgrade automatically
$ sudo apt install <package> # explicitly naming it often resolves a hold

$ # The full recovery sequence, in order, for a genuinely broken state
$ sudo dpkg --configure -a
$ sudo apt --fix-broken install
$ sudo apt update
$ sudo apt upgrade
```

E: Unable to locate package — a deceptively simple error

```
$ sudo apt install some-typo-name
E: Unable to locate package some-typo-name
```

Almost always either a genuine typo, a package that needs a repository not yet configured (Chapter 3), or simply running `apt install` without ever having run `apt update` on a fresh system — check the spelling first, then whether the right repository is actually enabled.

RedHat/dnf Troubleshooting in Practice

```
$ # Check for general consistency issues across the whole installed package set
$ sudo dnf check

$ # Clean cached metadata if it seems stale or corrupted
$ sudo dnf clean all
$ sudo dnf makecache

$ # Identify exactly what's blocking a specific install
$ sudo dnf install nginx -v # verbose output shows the actual resolution attempt

$ # If a package's history shows a problematic transaction, undo it specifically
```

```
$ dnf history
$ sudo dnf history undo <transaction-id>
```

DNF CLEAN ALL + MAKECACHE IS THE RPM-WORLD EQUIVALENT OF "JUST TRY REFRESHING EVERYTHING"

A surprising number of confusing dnf errors (packages that should clearly exist not being found, version mismatches that don't make sense) trace back to stale local metadata cache rather than a genuine system problem — clearing and rebuilding it is a cheap, safe first troubleshooting step before assuming something more serious is wrong.

Resolving a Genuine Dependency Conflict

1. **Read the actual error message carefully.** Both apt and dnf name the specific conflicting packages and versions explicitly — this is rarely a vague, unsolvable mystery.
2. **Check if one side has a newer version available** that resolves the conflict — sometimes a repository simply hasn't been refreshed (Chapter 2/5).
3. **Consider whether one of the conflicting packages is genuinely needed** — removing the less important one is often the simplest real resolution.
4. **As a last resort, search for whether this is a known, documented issue** — dependency conflicts in well-maintained distributions are usually already discussed publicly, with a known workaround.

FORCING AN INSTALL PAST A DEPENDENCY CONFLICT IS ALMOST ALWAYS THE WRONG MOVE

Both apt and dnf offer ways to forcibly ignore dependency checks entirely — genuinely tempting when frustrated, but this routinely produces a system with software that's installed yet doesn't actually work correctly, sometimes in ways that aren't obvious until much later. Understanding and resolving the actual conflict, even if slower, avoids trading a clear error message now for a confusing, hard-to-diagnose failure later.

CHAPTER 8 QUICK REFERENCE

- **Interrupted install:** dpkg --configure -a (apt) / verify with rpm -V (dnf)

- **Unmet dependencies:** apt --fix-broken install (apt) / dnf generally auto-resolves, dnf check for consistency (dnf)
- **Full apt recovery sequence:** dpkg --configure -a → apt --fix-broken install → apt update → apt upgrade
- **E: Unable to locate package** — check for typos first, then whether the right repository is enabled
- **dnf clean all + dnf makecache** — cheap first step for confusing dnf errors, often a stale metadata cache
- **dnf history undo <id>** — rolls back a specific problematic transaction
- **Resolving real conflicts:** read the error carefully, check for newer versions, consider removing the less critical package
- **Never force past a dependency conflict** — trades a clear error now for a confusing failure later
- **Next chapter:** other package managers overview — snap, flatpak, pacman

CHAPTER 9: OTHER PACKAGE MANAGERS OVERVIEW

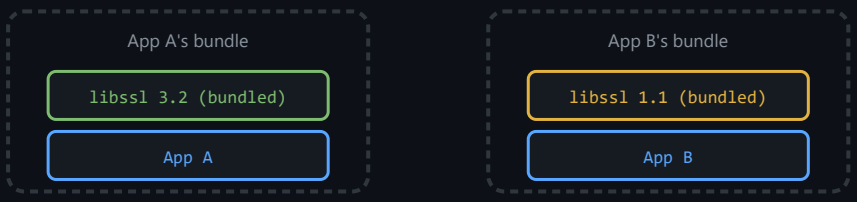
CHAPTER 9

Other Package Managers Overview

Snap, Flatpak, and pacman — a different philosophy of packaging, and when each genuinely makes sense over apt/dnf

Chapters 2-8 covered the traditional model — apt and dnf, both installing system packages that share libraries with the rest of the OS. A newer generation of tools takes a genuinely different approach: bundling an application with its own dependencies, isolated from the rest of the system.

The Sandboxed Application Model



Two genuinely incompatible versions of the same library, coexisting peacefully — exactly the dependency conflict scenario from Chapter 8, structurally impossible here

Snap (Canonical/Ubuntu)

```
$ sudo snap install spotify
$ snap list
$ sudo snap remove spotify
```

Developed by Canonical, deeply integrated into Ubuntu by default — automatic background updates, built-in sandboxing/permission system, and its own centralised store (the Snap Store).

Flatpak (Community/Red Hat-Backed)

```
$ # Typically needs Flathub added as a remote first
$ flatpak remote-add --if-not-exists flathub
https://flathub.org/repo/flathub.flatpakrepo
$ flatpak install flathub com.spotify.Client
$ flatpak list
$ flatpak uninstall com.spotify.Client
```

Distribution-agnostic by design — works essentially identically on Fedora, Debian, Ubuntu, Arch, and most other Linux distributions, with Flathub as the dominant central app store, similar in role to Snap's own store.

pacman (Arch Linux)

```
$ # Arch's own traditional (non-sandboxed) package manager — closer in spirit to
apt/dnf
$ sudo pacman -S nginx # install
$ sudo pacman -R nginx # remove
$ sudo pacman -Syu # sync repos and upgrade everything
```

PACMAN IS GENUINELY A DIFFERENT CATEGORY FROM SNAP/FLATPAK — INCLUDED HERE FOR COMPLETENESS

Unlike Snap and Flatpak's sandboxed-bundle philosophy, pacman is a traditional system package manager, conceptually much closer to apt/dnf — shared system libraries, no sandboxing by default. It's covered here purely because Arch is common enough to be worth recognising, not because it shares the sandboxing model the rest of this chapter focuses on.

Comparing the Sandboxed Approaches

📦 snap	📐 flatpak
Background auto-updates by default; strong Ubuntu integration; some reports of slower startup times for sandboxed apps due to the compression format used.	Distribution-agnostic; Flathub has a genuinely large catalogue; updates are user-triggered by default rather than automatic background updates.

When Sandboxed Packages Genuinely Make Sense

SITUATION	USE
A specific app's official repo package is genuinely outdated	Snap/Flatpak often has a much newer version maintained directly by the app's developer
Two apps need genuinely incompatible library versions	Sandboxing sidesteps the conflict structurally — exactly Chapter 8's conflict problem, avoided
Day-to-day system tools, services, server software	Stick with apt/dnf — better integration, no sandboxing overhead, the tools this whole course is built around
A desktop app you want tightly system-integrated	The native distro package, if available and current, usually integrates more smoothly than a sandboxed equivalent

SANDBOXED PACKAGES TRADE SOME INTEGRATION AND DISK SPACE FOR ISOLATION

Because each bundle includes its own copies of shared libraries rather than using the system's, total disk usage across several sandboxed apps is generally higher than the equivalent traditional packages — and desktop integration (themes, file dialogs, notification styling) is sometimes subtly less seamless. A reasonable, deliberate trade-off for specific apps, not a wholesale replacement for apt/dnf on a server or for core system tools.

CHAPTER 9 QUICK REFERENCE

- **Snap/Flatpak** — bundle an app with its own dependencies, sandboxed, structurally avoiding Chapter 8's dependency-conflict problem
- **snap install/remove/list** — Canonical's tool, deep Ubuntu integration, auto-updates by default
- **flatpak install/uninstall/list** — distribution-agnostic, Flathub as the dominant store, user-triggered updates by default
- **pacman** — Arch's traditional (non-sandboxed) package manager; conceptually closer to apt/dnf than to snap/flatpak
- **Use sandboxed packages for:** outdated repo versions, genuine library conflicts; stick with apt/dnf for system tools and servers
- **Trade-off:** isolation and version freedom, at the cost of disk space and sometimes slightly less integration

- **Next chapter (final):** practical workflows — auditing installed packages, cleaning up unused dependencies, keeping systems updated safely

CHAPTER 10 · FINAL CHAPTER

Practical Workflows

Auditing what's installed, cleaning up unused dependencies, and keeping a system safely up to date — the routine maintenance habits worth having

This final chapter pulls together every command from the previous nine into a small set of habits genuinely worth running periodically — on Philip's actual Debian server and Raspberry Pi, and on any RPM-based system encountered along the way.

Auditing What's Actually Installed

```
$ # apt — full list with version numbers
$ apt list --installed

$ # dnf — equivalent
$ dnf list installed

$ # Manually installed packages only — excludes anything pulled in purely as a dependency
$ apt-mark showmanual
```

`apt-mark showmanual` is genuinely useful for understanding what you actually *chose* to install over time, versus the much larger set of dependencies pulled in automatically alongside those choices — a clearer picture of a system's real, deliberate software footprint.

Finding and Removing What's No Longer Needed

```
$ # Orphaned dependencies — Chapter 2/5's autoremove, worth running periodically
$ sudo apt autoremove
$ sudo dnf autoremove

$ # Clear cached .deb/.rpm files no longer needed after install
```

```
$ sudo apt clean
$ sudo dnf clean all
```

APT CLEAN VS APT AUTOREMOVE — GENUINELY DIFFERENT THINGS, OFTEN RUN TOGETHER

`autoremove` removes orphaned *packages* that are no longer needed at all. `clean` removes cached *download files* (the .deb files themselves, already installed, sitting in `/var/cache/apt/archives/`) that served their purpose and are just consuming disk space now. Running both periodically keeps a system genuinely lean rather than slowly accumulating cruft.

A Safe, Repeatable Update Routine

```
$ # A sensible weekly/monthly routine for a Debian-based server
$ sudo apt update
$ sudo apt upgrade
$ sudo apt autoremove
$ sudo apt clean
```

Unattended Security Updates — Worth Knowing About, Even If Not Always Used

```
$ sudo apt install unattended-upgrades
$ sudo dpkg-reconfigure unattended-upgrades
```

Configures security updates to apply automatically, without manual intervention — a real trade-off between staying patched against known vulnerabilities promptly versus the small risk of an automatic update introducing an unexpected problem on a system you weren't actively watching at the time.

ALWAYS CHECK WHAT'S ABOUT TO HAPPEN BEFORE A MAJOR VERSION UPGRADE (E.G. DEBIAN 12 → 13)

A routine `apt upgrade` is low-risk; a full distribution upgrade between major releases is a genuinely bigger event — back up critical data first, read the release notes for known breaking changes, and consider testing on a non-critical system or VM before applying it to something like a production server.

A Final, Complete Maintenance Checklist

- ☑ **Refresh package lists before any install/upgrade**
  apt update, or rely on dnf's automatic metadata refresh — Chapters 2 and 5.
- ☑ **Periodically audit what's actually installed**
  apt-mark showmanual / dnf list installed — understand your system's real footprint.
- ☑ **Clean up orphaned dependencies and cached download files regularly**
  autoremove + clean — keeps disk usage and package count from slowly creeping up.
- ☑ **Only add third-party repositories from genuinely trusted sources**
  Remember Chapter 3/6 — every added repository is effectively trusted with root access.
- ☑ **Know the troubleshooting sequence before something actually breaks**
  dpkg --configure -a → apt --fix-broken install, or dnf's equivalents — Chapter 8.

CHAPTER 10 QUICK REFERENCE — AND COURSE WRAP-UP

- **apt list --installed / dnf list installed** — full installed-package audit
- **apt-mark showmanual** — what you actually chose to install, excluding auto-pulled dependencies
- **autoremove** — removes orphaned packages; **clean** — removes cached download files; genuinely different, often run together
- **Sensible routine:** update → upgrade → autoremove → clean, run periodically
- **unattended-upgrades** — automates security patching, a real convenience-vs-control trade-off
- **Major version upgrades** deserve more caution than routine upgrades — back up, read release notes, test first
- **Course recap:** concepts (Ch1) → APT (Ch2) → Debian repos (Ch3) → dpkg (Ch4) → RPM/dnf (Ch5) → RedHat repos (Ch6) → rpm (Ch7) → troubleshooting (Ch8) → snap/flatpak/pacman (Ch9) → maintenance workflows (Ch10)
- **The throughline across both families:** a friendly front end (apt/dnf) for dependency-aware everyday use, a low-level engine (dpkg/rpm) underneath for direct inspection and repair