ADMIN:DOCS

Ubuntu and Debian: the APT package manager guide for beginners

Understand package management on Ubuntu and Debian from the ground up: how dpkg and APT work together, deb822 sources, APT 3.0 changes, keyrings, Snap coexistence and troubleshooting.

Category: Linux beginners **Published:** 21.10.2024 **Last reviewed:** 08.09.2026

Anyone taking the first steps on a freshly installed Ubuntu desktop or a Debian server quickly meets a conspicuous abbreviation: **APT** (**Advanced Package Tool**). Where other systems make you hunt websites for trusted installers or click through countless dialogs, Debian and Ubuntu need a short terminal command to set up highly complex software packages together with all required system libraries fully automatically.

Since the late 1990s the Debian package system has been the reliable foundation for millions of computers worldwide — from a home Raspberry Pi through school laptops to huge cloud server farms. Behind the seemingly simple command `apt install` sits an elaborate piece of software architecture: cryptographically secured software depots, a mathematical dependency checker and a tireless background process work together so that the system stays stable and conflict-free even after hundreds of software installations.

Debian and Ubuntu package management in technical detail:

The interplay between `dpkg` and APT, the visual and structural changes in APT 3.0, clean package cleanup as well as the modern deb822 format and coexistence with Snap form the foundation for stable production systems.

Source: https://admindocs.de/en/linux-beginners/ubuntu-debian-apt-package-manager-guide

> **apt vs. apt-get at a glance:** The `apt` command is the modern, human-oriented tool for interactive work in the console. It combines the most important functions from `apt-get` and `apt-cache` with colored output and progress displays. The long-standing `apt-get` still exists in parallel and is used mainly in shell scripts, because its output stays strictly stable and is never surprised by visual changes.

Architecture and fundamentals: how dpkg and APT work together

To interpret error messages correctly and operate the system with confidence, it helps to look at the two layers of package management. On Ubuntu and Debian the division of labor is clear:



1. The .deb format: the archive with a package insert

A file with the `.deb` extension is the atomic software package. Technically it is a standard Unix archive (`ar`) that encapsulates three elementary parts:

- **Debian-binary:** *A tiny text file that tells the system the format version of the package.*
- **control.tar.xz:** *The control center of the package. Here you find the name, the exact version number, maintainer information, the package description and above all the list of all dependencies (*"I only run if libc6 is at least version 2.39"*). Scripts (`preinst`, `postinst`, `prerm`, `postrm`) that stop services or create user accounts also live here.*
- **data.tar.xz:** *The actual payload. It holds all binaries, help texts, icons and configurations, structured exactly as they later have to be unpacked onto disk under `/usr/bin`, `/etc` or `/var`.*

2. dpkg: the local craftsman

The `dpkg` tool (*Debian Package*) is the low-level manager. It unpacks `.deb` files, copies the contents to the right place in the filesystem and runs the installation scripts. `dpkg` keeps the central registry of all packages present on disk in `/var/lib/dpkg/status`.

`dpkg` has neither a network function nor foresight, however: if you try to install a program with `sudo dpkg -i program.deb` that needs two further libraries, `dpkg` aborts helplessly. It simply does not know from which server it could download those libraries.

3. APT: the networked logistics lead

This is exactly where **APT** enters. Thanks to regularly updated package lists, APT knows all available programs on the remote servers. When you request a package, APT reads its control data, recursively determines all missing pieces, downloads all required `.deb` files over encrypted network connections, checks their digital signatures and hands the packages to `dpkg` in exactly the right order.

The modern generation: what sets APT 3.0 and current releases apart

With Debian 13 (Trixie) and modern Ubuntu releases (from Ubuntu 24.10 and Ubuntu 26.04 LTS) APT has taken a large evolutionary step: **APT 3.0** and the refined **APT 3.1**.

The most important changes concern the user interface and the inner calculation logic:

- **Structured column layout:** Instead of messy walls of text, APT shows installation candidates and dependencies in clean table columns. Size, package name and target version can be taken in at a glance.
- **Intuitive color coding and deletion safety:** Terminal output uses targeted colors: components to be newly installed or updated appear in green. Package deletions now sit clearly at the end of the list and are unambiguously highlighted in red. That protects against accidentally confirmed uninstalls.
- **Smooth Unicode progress bars:** Progress displays while unpacking and configuring look modern and dynamic.
- **The new dependency solver (solver3):** Behind the scenes a new backtracking algorithm resolves even highly nested dependency conflicts that older heuristics failed on.
- **Modern cryptography with OpenSSL:** Older crypto helper libraries have been replaced; APT uses OpenSSL natively for TLS connections and cryptographic hash checks.
- **Automatic paging:** Long package listings are shown page by page, similar to Git, so output no longer races uncontrollably through the terminal window.

Package sources and repository architecture (deb822)

Ubuntu and Debian fetch their software from huge server archives (*repositories*). To keep order, stability and license rights, this pool is split into clearly defined segments.

Ubuntu's four software components

On an Ubuntu system software is divided into four categories:

- **main:** Fully free open-source software, supported directly by Canonical and supplied with guaranteed security updates for the entire lifetime of the distribution.
- **restricted:** Proprietary device drivers (for example for Nvidia graphics cards or certain Wi-Fi chips) that are indispensable for smooth hardware operation.
- **universe:** The huge community universe. Tens of thousands of free programs maintained by the worldwide Debian and Ubuntu community.
- **multiverse:** Software subject to legal restrictions or patent issues (such as special multimedia codecs or fonts).

The modern deb822 format in /etc/apt/sources.list.d/

For decades software sources were written as a single line (`deb http://archive.ubuntu.com/ubuntu noble main`). These one-liners in `/etc/apt/sources.list` were error-prone and hard to read.

Since Ubuntu 24.04 LTS the system defaults to the structured **deb822 format**. The primary system configuration now lives in `/etc/apt/sources.list.d/ubuntu.sources`:

```
Types: deb
URIs: http://archive.ubuntu.com/ubuntu/
Suites: noble noble-updates noble-backports
Components: main restricted universe multiverse
Signed-By: /usr/share/keyrings/ubuntu-archive-keyring.gpg

Types: deb
URIs: http://security.ubuntu.com/ubuntu/
Suites: noble-security
Components: main restricted universe multiverse
Signed-By: /usr/share/keyrings/ubuntu-archive-keyring.gpg
```

The advantages of this format are obvious: each field (`Types`, `URIs`, `Suites`, `Components`, `Signed-By`) sits on its own line. That makes reading easier for humans and error-free editing by automated scripts much simpler.

Enabling third-party sources safely (keyrings)

In the past, foreign sources were often added with `apt-key add`. That was a massive security risk, because a key imported that way could have manipulated any system package. The old `apt-key` is therefore fully deprecated.

The modern standard is that every GPG key lives in isolation under `/etc/apt/keyrings/` and is bound in a deb822 source file via `Signed-By` to exactly that one repository.

□□ Practical example:

We add the official Docker repository by the book through the modern deb822 procedure:

```
# 1. Ensure the key directory exists with correct permissions
sudo install -m 0755 -d /etc/apt/keyrings

# 2. Download the public GPG key and store it de-armored
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o
/etc/apt/keyrings/docker.gpg
sudo chmod a+r /etc/apt/keyrings/docker.gpg

# 3. Create the repository as a structured .sources file in deb822 format
sudo tee /etc/apt/sources.list.d/docker.sources <<EOF
Types: deb
URIs: https://download.docker.com/linux/ubuntu
Suites: $(lsb_release -cs)
Components: stable
Signed-By: /etc/apt/keyrings/docker.gpg
EOF

# 4. Update package indexes
sudo apt update
```

> ⚠ **Beware of outdated tutorials:** Sites that tell you to add keys with `apt-key add` or to use unencrypted URLs without `Signed-By` endanger the integrity of your system. Use only dedicated key files under `/etc/apt/keyrings/`.

Package management in daily work: search, find and install

Daily work with APT is straightforward and logically structured. Once you know the basic commands, you navigate the entire software catalog with purpose.

1. Finding packages and inspecting details

Before you install something you want to know the exact package name, which version is available and who maintains it:

□□ Practical example:

Search packages and inspect metadata:

```
# Search package names and descriptions for a term
apt search webserver

# Faster: restrict the search to package names
apt search --names-only nginx

# Show the full package insert of a package
apt show nginx
```

2. Root-cause analysis: why is a package installed?

You often hit installed packages and wonder: *“Who put this library on my system, and is it still needed?”*

Two practical tools help analyze the dependency tree:

□□ Practical example:

Trace package causes and dependency chains:

```
# Show all currently installed packages that depend on this library
apt-cache rdepends --installed libssl3

# Especially convenient: determine the causal chain up to the installed main program
sudo apt install aptitude
aptitude why libssl3
```

`aptitude why` gives a direct, understandable answer (e.g. *“nginx depends on libssl3”*). You immediately see whether a component can be deleted safely or is indispensable for business-critical services.

3. Finding missing commands with apt-file

You have certainly seen an error such as `dig: command not found` or `header file sqlite3.h is missing`. But which Debian package hides that file?

The practical tool `apt-file` helps here. It searches the file lists of all available packages without those packages having to be installed:

□□ Practical example:

Install `apt-file` and search for files:

```
# Install apt-file
sudo apt install apt-file

# Download the global file index (required once)
sudo apt-file update

# Find which package contains the dig tool
apt-file search bin/dig

# Find which package provides a given developer header
apt-file search /usr/include/sqlite3.h
```

`apt-file` reliably reports that you need the `bind9-dnsutils` package for `dig`.

4. Installing and simulating software

Installation uses the `install` subcommand. APT determines all dependencies and presents a preview of the transaction:

□□ Practical example:

Install programs or simulate actions safely:

```
# Install a single package
sudo apt install htop

# Install several packages in one shared run
sudo apt install git tmux vim curl

# Dry run: shows exactly what would happen without changing anything
apt install -s nginx

# Install a local .deb file (APT automatically resolves missing dependencies)
```

```
sudo apt install ./my-package.deb
```

Typical output of an APT transaction summary:

```
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
The following additional packages will be installed:
  libnginx-mod-http-image-filter nginx-common nginx-core
Suggested packages:
  nginx-doc
The following NEW packages will be installed:
  libnginx-mod-http-image-filter nginx nginx-common nginx-core
0 upgraded, 4 newly installed, 0 to remove and 0 not upgraded.
Need to get 820 kB of archives.
After this operation, 2,980 kB of additional disk space will be used.
Do you want to continue? [Y/n]
```

> **Tip for scripts:** When you run automation tasks and do not want to confirm the `[Y/n]` prompt by hand, append the `-y` parameter: `sudo apt install -y curl`.

5. Uninstalling software: remove vs. purge

Removing programs on Debian and Ubuntu is designed to be particularly careful. There are two stages:

Practical example:

Remove packages cleanly and wipe leftover configuration:

```
# Stage 1: delete binaries, but keep configurations in /etc/
sudo apt remove nginx

# Stage 2: radical cleanup – remove the program together with all configuration files
sudo apt purge nginx

# Remove orphaned dependencies that are no longer needed
sudo apt autoremove

# Thorough system hygiene: wipe orphaned packages including leftover configuration
sudo apt autoremove --purge
```

Action	Command	Effect on program files	Effect on configuration files
Normal removal	`sudo apt remove <package>`	Deleted	Remain for a possible later reinstall
Complete purge	`sudo apt purge <package>`	Deleted	Removed from the system without residue
Empty the orphanage	`sudo apt autoremove`	Deleted (if unused)	Configurations remain
Maximum hygiene	`sudo apt autoremove --purge`	Deleted	Fully purged

6. Using metapackages and tasks

You want to compile software from source or set up a complete working environment? Instead of collecting dozens of individual packages such as `gcc`, `make`, `libc-dev` and `dpkg-dev` by hand, you use **metapackages**:

□□ Practical example:

Install development environments and desktop tasks:

```
# Get essential compilers and libraries through a metapackage
sudo apt install build-essential

# Install whole task areas via the appended circumflex character
sudo apt install lamp-server^
```

The Ubuntu ecosystem: when APT, when Snap, when Flatpak?

On a modern Ubuntu installation you will inevitably notice that another system exists besides APT: **Snap**. Many beginners are confused when they type `sudo apt install firefox` and APT suddenly reports that a Snap is being set up instead.

Why does this dualism exist?

Traditional `.deb` packages share common system libraries. That saves RAM and disk space, but brings challenges: if a modern web browser such as Firefox requires a brand-new library, while the operating system (for example an LTS version) freezes that library at an older state for five years for stability, a conflict arises.

Canonical solves this dilemma with **Snap**: Snaps bring all required libraries themselves and run isolated in a secured sandbox.

Criterion	APT (.deb)	Snap	Flatpak (Flathub)
Use	System tools, server services, CLI, kernel	Graphical desktop apps, modern CLI tools	Focus purely on desktop applications
Isolation	Runs directly in the operating system	Runs in an AppArmor sandbox	Runs in a Bubblewrap sandbox
Disk space	Extremely economical (shared libraries)	Larger (ships its own libraries)	Larger (uses shared runtimes)

Source: https://admindocs.de/en/linux-beginners/ubuntu-debian-apt-package-manager-guide

Criterion	APT (.deb)	Snap	Flatpak (Flathub)
Updates	Centrally controlled via system updates	Automatically in the background	Manually or via the desktop software center

> **Practical guidance for daily work:**
>
> - Use **APT** for everything that works close to the system: `htop`, `git`, `curl`, web servers (`nginx`, `apache2`), databases (`postgresql`) and system libraries.
> - Use **Snap or Flatpak** for complex graphical end-user programs such as browsers, Spotify, Discord or LibreOffice when you always need the very latest features.

System care, updates and kernel hygiene

A secure Linux system is based on regular care. On Debian and Ubuntu you strictly distinguish between updating the software catalog and actually installing new packages.

1. The pair: apt update and apt upgrade

A common misunderstanding among newcomers: `apt update` installs **not a single piece of software** on your computer! It only downloads the fresh tables of contents from the servers. Only `apt upgrade` actually applies the updates:

Practical example:

Refresh the software catalog and update installed packages:

```
# 1. Refresh local package indexes (update the catalog)
sudo apt update
```

```
# 2. Check which packages have new versions
apt list --upgradable

# 3. Update all installed packages conservatively
sudo apt upgrade
```

2. The difference: upgrade vs. full-upgrade

Debian and Ubuntu know two upgrade stages:

- **apt upgrade:** Updates packages, but strictly refuses to uninstall already installed packages or to pull in new packages that were not present before.
- **apt full-upgrade** (historically also `dist-upgrade`): Has permission to pull in new dependencies or remove obsolete conflicting packages. This is necessary especially for new Linux kernel versions.

> ⚠ **Why kernel updates sometimes stick:** When a new Linux kernel appears, APT must add a new kernel package. A plain `apt upgrade` holds that kernel back. Therefore run `sudo apt full-upgrade` regularly on servers and desktops to activate the current security kernel.

3. The riddle of phased updates (“packages have been kept back”)

Sometimes APT reports after an `apt upgrade`:

```
The following packages have been kept back:
  libglib2.0-0
```

Many newcomers wrongly suspect an error in the package database. The explanation is: **phased updates** (staged rollout).

Canonical rolls Ubuntu updates out in steps. At first only 10 % of machines receive the new package. If no crashes occur, the release is widened over several days to 20 %, 50 % and finally 100 % of all systems. If your machine is not yet in turn, APT holds the package back temporarily.

You do not have to do anything in this case: wait one or two days, and the package will install itself on the next regular upgrade.

4. All-clear: what do the Ubuntu Pro / ESM messages mean?

When running `apt upgrade` on Ubuntu you often hit lines like these:

```
Get more security updates through Ubuntu Pro with 'esm-apps' enabled:
  imagemagick libimage-magick-perl
```

These are notices about the **Expanded Security Maintenance (ESM)** program. Ubuntu LTS releases receive five years of security updates for the `main` component by default. For programs from `universe`, Canonical offers additional patches through Ubuntu Pro for up to ten or twelve years. For private individuals Ubuntu Pro is completely free on up to five machines. This message is not an error and does not mean that your system is locked or unusable.

5. Automatic security patches with unattended-upgrades

A Linux server should close security holes on its own without you having to log in daily via SSH:

□□ Practical example:

Enable automated security updates:

```
# Install the package for unattended upgrades
sudo apt install unattended-upgrades

# Enable the service interactively
sudo dpkg-reconfigure --priority=low unattended-upgrades
```

The configuration is kept in `/etc/apt/apt.conf.d/50unattended-upgrades`:

```
Unattended-Upgrade::Allowed-Origins {
    "${distro_id}:${distro_codename}-security";
};
Unattended-Upgrade::AutoFixInterruptedDpkg "true";
Unattended-Upgrade::MinimalSteps "true";
Unattended-Upgrade::Remove-Unused-Kernel-Packages "true";
Unattended-Upgrade::Remove-Unused-Dependencies "true";
Unattended-Upgrade::Automatic-Reboot "false";
```

Advanced techniques: pinning and version freeze

In professional environments there are situations in which you must keep full control over version states.

1. Freezing packages at a given version (hold)

If a database such as PostgreSQL or a graphics driver is running stably and you want to rule out an accidental update during a system upgrade, you set a version lock:

□□ Practical example:

Protect packages from updates:

```
# Freeze a package version permanently
sudo apt-mark hold postgresql-16

# Inspect all currently held packages
apt-mark showhold

# Lift the lock again when an update is desired
sudo apt-mark unhold postgresql-16
```

2. APT pinning: prioritizing sources in a targeted way

With **APT pinning** you can use configuration files in `/etc/apt/preferences.d/` to specify which repository packages should preferably come from:

Practical example:

Define a pinning rule:

```
# /etc/apt/preferences.d/99-custom-pinning
Package: nginx*
Pin: release o=Ubuntu
Pin-Priority: 900

Package: *
Pin: origin packages.example.com
Pin-Priority: 100
```

- **Priority > 1000:** The version is forced, even if a downgrade is required.
- **Priority 500 to 990:** Default behavior for installed distributions.
- **Priority 100:** The package is only installed if no other source exists.

- **Priority < 0:** The package is completely excluded from installation.

Store hygiene and troubleshooting

Even the most robust system can stumble — for example after an unexpected power loss during an update. Here are the most important repair moves.

1. Emptying download caches

Every `.deb` file APT downloads goes into the cache directory `/var/cache/apt/archives/`. Over months this folder can occupy several gigabytes of disk space:

□□ Practical example:

Clean the local package cache:

```
# Show disk space used by the cache
du -sh /var/cache/apt/archives/

# Gentle: delete only old packages that no longer exist on the servers
sudo apt autoclean

# Radical: empty the entire download cache without residue
sudo apt clean
```

2. Resolving locked lock files

One of the most common scares for beginners is this console message: `E: Could not get lock /var/lib/dpkg/lock-frontend - open (11: Resource temporarily unavailable)`

```
┌──────────────────────────────────────────────────────────┐
│                TROUBLESHOOTING APT LOCKS                 │
├──────────────────────────────────────────────────────────┤
│                                                          │
│   Symptom: E: Could not get lock /var/lib/dpkg/lock-...  │
│                                                          │
│   1. Identify the blocking process:                      │
│      sudo lsof /var/lib/dpkg/lock-frontend               │
│      sudo fuser -v /var/lib/dpkg/lock-frontend           │
│                                                          │
│   2. Check: is unattended-upgrades or apt running?       │
│      ──> Wait until the active transaction ends!         │
│                                                          │
│   3. If the process died after a system crash:           │
│      sudo dpkg --configure -a                            │
│      sudo apt --fix-broken install                       │
│                                                          │
└──────────────────────────────────────────────────────────┘
```

> ⚠ **Never delete lock files in a hurry:** In 95 % of all cases the automatic security updater (`unattended-upgrades`) or the graphical software center is simply running in the background. If you delete the lock file by hand, you damage the package database! Wait a few minutes until the process has finished.

□□ Practical example:

Find the background process and repair the system after a crash:

```
# 1. Check which process holds the lock
sudo lsof /var/lib/dpkg/lock-frontend
```

```
# 2. After a system crash: repair incompletely configured packages
sudo dpkg --configure -a

# 3. Resolve broken dependencies
sudo apt --fix-broken install
```

3. Validating filesystem integrity with debsums

You suspect that system files were damaged or edited by accident? With `debsums` you check every file on the machine against the original checksums of the maintainers:

□□ Practical example:

Integrity check of the operating system:

```
# Install debsums
sudo apt install debsums

# Check all system files (shows only changed or faulty files)
sudo debsums -s
```

Command Reference (Cheatsheet)

Category	Command	Function and purpose
Catalog sync	`sudo apt update`	Synchronizes local tables of contents with the repositories
Search	`apt search <term>`	Searches package names and descriptions

Category	Command	Function and purpose
Package details	`apt show <package>`	Shows version, maintainer, size and dependencies
Cause search	`aptitude why <package>`	Explains why a package is installed and who needs it
File search	`apt-file search <path>`	Finds which package provides a given file
Install	`sudo apt install <package>`	Installs software including all required dependencies
Simulation	`apt install -s <package>`	Simulates the installation safely in a dry run
Uninstall	`sudo apt remove <package>`	Removes binaries, keeps configurations
Full purge	`sudo apt purge <package>`	Removes software together with all configuration files
System hygiene	`sudo apt autoremove --purge`	Deletes orphaned packages and leftover configuration
Update	`sudo apt upgrade`	Updates installed software packages conservatively
Full upgrade	`sudo apt full-upgrade`	Installs new kernel packages and resolves version conflicts
Hold version	`sudo apt-mark hold <package>`	Prevents automatic updates for this package
Release hold	`sudo apt-mark unhold <package>`	Frees a pinned package for upgrades again
Empty cache	`sudo apt clean`	Deletes all downloaded `.deb` files from the cache
Repair	`sudo apt --fix-broken install`	Corrects faulty or incomplete dependencies

Further Resources

Resource	Description	Type

Source: https://admindocs.de/en/linux-beginners/ubuntu-debian-apt-package-manager-guide

Official Ubuntu Server documentation	Full documentation on package management and system care	Official documentation
Debian APT handbook	Fundamental handbook on the architecture of the Advanced Package Tool	Official handbook
Debian Wiki: package management and pinning	Best practices for pinning, repositories and troubleshooting	Community wiki
Ubuntu Security Notices	Official security notices, CVE patches and release status	Security portal
Debian Security Tracker	Searchable database of all security advisories for Debian	Security portal

Conclusion

With APT, the Debian and Ubuntu ecosystem has one of the most mature and reliable package managers in the entire open-source world. The interplay of the low-level tool `dpkg` and the networked coordinator APT guarantees that software installations run transparently, traceably and with cryptographic protection.

Anyone who masters the modern standards — from structured **deb822 sources** through isolated **keyrings** to knowledge of phased updates and coexistence with Snap — steers the system safely through every version change.

> **Practical tip for daily work:** Get into the habit of running `apt list --upgradable` briefly before larger installations or updates. You then see in advance exactly which system components will receive an update, and you keep full control over changes to the operating system.