Engineer

Package

code bundled with a description, ready to install

What it is

A package is the form in which software travels from the people who write it to the machines that run it: the code, as an archive or a directory, together with a manifest giving its name, its version and what else it needs in order to work. The plural is packages, and system documentation often says software package. Each ecosystem draws the edges in its own place. For npm, a package is a file or directory described by a package.json file, and the npm documentation separates it from a module, anything Node.js can load with require() or import; only modules that carry a package.json are packages. Python uses the word for two things, and the Python Packaging User Guide keeps them apart: an import package is a directory of modules that code imports, while a distribution package is the versioned archive a user downloads and installs. In Rust, a package bundles one or more crates with a Cargo.toml that says how to build them. On Debian, the unit is the binary package, the .deb file the system installs and tracks.

A package is a unit of distribution, and it is easiest to define against its neighbours. A library is code; a package is how that code is shipped, and one library can reach users as an npm package, a Python wheel and a Debian package at once. A package manager is the program that reads manifests, fetches what they list and installs it all in a workable order: npm, pip, Cargo and apt. A registry or archive is where packages wait to be fetched, such as the npm registry, PyPI, crates.io or a Debian mirror. Language package managers install code for one project, usually into a directory beside it. System package managers install software for the whole machine and record which package put each file where.

Use the concept in a decision

Put “What is a package?” to work in a real decision. We can help relate the concept to the system you are building or buying. The shape of a system worked out on paper first, so the parts fit and the cost is known before anyone writes code.

Explore our services Get in touch →

Test the assumptions behind the term

Use the explanation and linked work to frame questions for your project. We can examine the assumptions with you before a tool is chosen. One infrastructure plan for 14 departments — 14 research departments on one infrastructure plan

One infrastructure plan for 14 departments Get in touch →

Why it matters

Third-party code reaches a project in one of three ways: through a package manager, copied into the repository with its licence and a checksum, which is called vendoring, or built from source. A package manager is the usual choice for good reason. It resolves the whole tree of dependencies, updates them with one command and records what it installed. The cost is trust: the project now relies on every manifest in that tree and every registry it came from. Vendoring swaps that trust for work, since the copied code changes only when someone copies a new version in on purpose. Building from source gives the most control and asks for the most upkeep.

The manifest is where most of the trade-offs show. It usually names a range of versions rather than one; in npm, a caret range such as ^10.0.1 accepts any later release with the same major version. Ranges let fixes arrive without anyone editing the file, and they also mean two installs a week apart can produce different trees. A lock file closes that gap. npm writes package-lock.json to describe the exact tree it built so later installs reproduce it, Cargo writes Cargo.lock with the exact version of every dependency, and both expect the file to be committed. The manifest records what a project is willing to accept; the lock file records what it received.

Manifests also say how badly one package needs another. The Debian Policy Manual defines a ladder. Depends is absolute: the package is not configured until everything it depends on has been. Recommends is strong but not absolute, for packages that accompany this one in every installation except unusual ones. Suggests names packages that make it more useful, while installing without them is perfectly reasonable. An install that leaves out a recommended package is valid, and the software inside may still need that package for one of its jobs.

Every tool that reasons about packages trusts their metadata. A licence checker reads the declared licence, often an SPDX expression such as MIT OR Apache-2.0, where OR offers a choice and AND requires every licence at once. A vulnerability scanner matches declared names and versions against published advisories.

When a declaration is wrong, every tool that reads it is wrong in the same way, which is why the licence files inside a shipped package deserve a read of their own.

What it looks like in practice

A hardening review of a production server found that its intrusion-banning jail had never banned anyone, and one of three separate causes was a package relationship. The ban action called a packet-filter program that the packaging lists as recommended rather than required, and the server had been built without it. The jail started, counted failed logins and reached its decision every time, then broke at the final step of applying the ban. The review of three SSH protections found it by inspecting the running system instead of the configuration files, and left a rule that every security control is backed by an automated assertion, not a comment.

A Rust core linked into a shipped application brings its whole dependency graph into the product, 644 crates here, each with its own licence declaration. A dependency several layers down can move to a copyleft licence from one version to the next, and the only trace would be a changed line in a lock file few people read. The licence audit of 644 crates runs on every build: it evaluates each crate’s licence expression against an allow list, passes a choice of licences when any one is allowed and a combination only when all are, and stops the build on anything unmatched. To show the check can fire, it was broken deliberately twice, once by rewriting a crate’s licence expression and once by moving the pinned core revision without regenerating the audit. The page states the limit plainly: the audit reads declared metadata, which can be wrong, and it is not legal advice.

The company’s own git forge was installed from its vendor’s own package source as a single program that the operating system keeps running. Adding a vendor’s repository to a server is a decision in its own right, because from then on routine updates install whatever that vendor publishes and signs.

This site itself needs no Node to run. Its package.json is marked private, which tells npm to refuse to publish it, lists development tools only, with caret ranges, and sits beside a committed package-lock.json. Its pyproject.toml configures a linter, a spell checker and a type checker and has no [project] or [build-system] table, since it describes no package to build. The icons are vendored, copied in with the upstream licence and a SHA-256 checksum for every file.

Where it goes wrong

A feature reports healthy and does nothing

A service starts, logs that it is running and quietly skips one of its jobs every time. A program it calls belongs to a recommended or suggested package, which the install was entitled to leave out. When a feature depends on an external program, check for it at start-up and stop with a clear error if it is missing, declare it as a hard requirement in your own deployment, and test the feature end to end on a server built like production.

Two machines install different code from the same manifest

The build passes on a laptop and breaks in continuous integration, or the reverse, with no change to the project's own code. The manifest names version ranges, the lock file was never committed, and a dependency published a release between the two installs. Commit the lock file, install from it in automation with commands that refuse to change it, such as npm ci or cargo build –locked, and update it as a separate, reviewed change.

A licence changes inside a familiar dependency

A product that passed a licence review last year now ships code under terms nobody agreed to, while its list of dependencies looks the same. An existing dependency changed its licence between versions, and a routine update pulled the new version in through the lock file. Check declared licences against an explicit allow list on every build, so the change stops the build on the day it arrives.

Further reading

- npm Docs: About packages and modules — npm's own definition of a package, and why a module is not always one.
- Debian Policy Manual: Declaring relationships between packages — Depends, Recommends, Suggests and the other fields, and what a package manager does with each.
- Python Packaging User Guide: Writing your pyproject.toml — the three tables of pyproject.toml and the metadata a Python distribution package declares.
- The Cargo Book: Cargo.toml vs Cargo.lock — why the manifest describes what a project accepts, the lock file records exact versions, and when to commit it.

- SPDX Specification 2.3.1: License expressions — the syntax packages use to declare one licence, a choice of several or a set that applies together.

Glossary

Choose for your actual constraints

Describe the workflow, the existing interfaces and the users who need the change. We can define a testable delivery and a handover that lets your team maintain it.

See what it costs Get in touch →

Make the terminology practical

Ask how “What is a package?” applies to your organisation. A concrete example from your current system is enough to start the discussion.

Explore our services

welcome@engineer.company +45 50 10 61 84 Copenhagen, Denmark

Take this page with you

Contact card

Carefully Crafted With Love by D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Copyright © 2025 – present · Engineer ApS · All rights reserved.

Printed from https://engineer.company/glossary/package/