npm and Package Publishing Course

Flavio Copes

Updated August 3, 2026

Contents

Create, test, package, publish, secure, version, and maintain a small npm package as a real consumer contract.

What you'll learn: Publish and verify a small npm package with intentional exports, dependency policy, consumer tests, a release gate, and a recovery plan.

Package foundations

Define the contract, metadata, exports, license, and packed artifact.

Decide what the package owns

Draw a narrow public boundary before adding metadata, builds, or publishing automation.

Before choosing a tool or writing more code, make the requirement concrete. The utility owns title normalization and slug creation, but it does not own routing, databases, or UI state.

A package is a versioned contract consumed by code you do not control, even when the first consumer is your own project. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

A common failure is to extract a grab bag of unrelated helpers because they happen to live in one folder. The happy path may still work, which makes the mistake easy to miss. write the package purpose, supported inputs, outputs, runtime assumptions, and explicit non-goals. Keep the first version small enough that every important input and output remains visible.

This supports the module goal: Turn a useful function into a small package whose public surface and published files are intentional. Capture the request, command, trace, screenshot, test result, or other evidence that proves the result.

Create a one-page contract and three usage examples before moving the function into its package directory. Change one condition on purpose, predict the result, and compare the prediction with what actually happened.

Write package.json deliberately

Set identity, version, module type, entry points, scripts, engines, and repository metadata with intent.

Start from the behavior you can observe. The package declares a scoped name, initial version, ESM type, supported Node range, license, repository, and test script.

`package.json` is both local project configuration and published package metadata, so each field affects tools or consumers. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

It is tempting to accept generated metadata without checking what consumers and the registry will see. That trades a clear decision for an assumption you will eventually have to debug. The stronger move is to keep only accurate fields, validate the package name and version, and document the supported runtime. Make the boundary explicit in code and in the project notes.

Tie the decision back to the larger job: Turn a useful function into a small package whose public surface and published files are intentional. Record enough evidence that another developer can repeat the result without relying on your memory.

Create the manifest by hand, run npm metadata commands, and explain every retained field in one sentence. Repeat it once with an invalid or hostile condition and write down the boundary the failure revealed.

Design package exports

Expose a small supported entry-point map and keep internal files private to consumers.

Consumers import `slugify-title` or `slugify-title/strict`; they do not reach into `dist/internal.js`. That contrast gives us something useful to test instead of a rule to memorize.

The `exports` field defines public import paths and can provide different targets for import, require, types, or environments. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

The happy path can hide a bad design: you can publish every generated file as an accidental public API and still get one successful demo. Push past the demo. define the smallest export map you can support and test each documented import from a separate consumer, then test the condition most likely to prove the choice wrong.

The module is moving toward one outcome: Turn a useful function into a small package whose public surface and published files are intentional. Save the before-and-after evidence now; it will make the final project review much more honest.

Add one main export and one intentional subpath, then prove an internal path is unavailable. Save the evidence, then explain which requirement would force you to choose a different design.

Inspect the package tarball

Use npm pack and file controls to see the exact artifact before it reaches the registry.

A dry-run shows compiled files, types, README, license, manifest, size, and accidental secrets or fixtures. This is a small part of a small `slugify-title` package that is tested locally, consumed from another project, and published safely, but the boundary it creates affects everything that follows.

The Git repository and the npm tarball are different products; only the tarball is installed by consumers. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

Watch for one shortcut in particular: trying to assume gitignore alone defines the published artifact. It makes later failures harder to locate. Instead, use a focused `files` list, run `npm pack --dry-run`, unpack the tarball, and inspect it as a consumer would. Prefer an implementation you can explain from the outside before optimizing it.

Keep the wider goal in view: Turn a useful function into a small package whose public surface and published files are intentional. A short observation with concrete evidence is more useful than a confident sentence with no reproduction path.

Generate the tarball, verify every included file, and make the check fail if a fixture or environment file appears. Hand the result to someone else and see whether the behavior is clear without an oral explanation.

Check your understanding: package foundations

Check package boundaries, names, metadata, entry points, files, licenses, and packed contents.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Versions and dependencies

Use semantic versions, ranges, lockfiles, dependency roles, and peer contracts.

Use semantic versioning

Connect major, minor, and patch releases to the compatibility promise made to consumers.

Before choosing a tool or writing more code, make the requirement concrete. A bug fix becomes a patch, a backward-compatible option becomes a minor, and removing a supported call becomes a major.

Semantic versioning communicates whether an upgrade should preserve the documented public API. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

A common failure is to choose version numbers from effort, marketing importance, or the calendar without documenting a different scheme. The happy path may still work, which makes the mistake easy to miss. classify changes from the consumer contract and record breaking behavior before selecting the next version. Keep the first version small enough that every important input and output remains visible.

This supports the module goal: Make dependency and version choices that communicate compatibility rather than merely making installation succeed. Capture the request, command, trace, screenshot, test result, or other evidence that proves the result.

Classify ten realistic changes to the slug utility and defend the version bump for each. Change one condition on purpose, predict the result, and compare the prediction with what actually happened.

Read version ranges

Predict which releases satisfy caret, tilde, exact, comparison, tag, and prerelease specifications.

Start from the behavior you can observe. The package deliberately allows compatible patch and minor updates for one dependency while pinning a fragile build tool.

A dependency range is an update policy encoded in the manifest, not a decorative prefix. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

It is tempting to copy ranges without checking what future versions they admit. That trades a clear decision for an assumption you will eventually have to debug. The stronger move is to expand each important range into its actual lower and upper bounds and choose it from compatibility risk. Make the boundary explicit in code and in the project notes.

Tie the decision back to the larger job: Make dependency and version choices that communicate compatibility rather than merely making installation succeed. Record enough evidence that another developer can repeat the result without relying on your memory.

Use npm tooling to test candidate versions against several ranges, including a zero-major and a prerelease. Repeat it once with an invalid or hostile condition and write down the boundary the failure revealed.

Separate dependency roles

Place runtime, development, optional, and peer dependencies according to who must install and control them.

A library plugin declares its host framework as a compatible peer instead of installing a second private copy. That contrast gives us something useful to test instead of a rule to memorize.

Dependency sections describe ownership: runtime code needs dependencies, maintainers need devDependencies, and host integrations often use peers. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

The happy path can hide a bad design: you can move packages between sections only to silence an installation warning and still get one successful demo. Push past the demo. trace every import into the published runtime and decide whether the package or consuming application owns that dependency, then test the condition most likely to prove the choice wrong.

The module is moving toward one outcome: Make dependency and version choices that communicate compatibility rather than merely making installation succeed. Save the before-and-after evidence now; it will make the final project review much more honest.

Audit the project after a production-only install and after packing; explain every dependency section entry. Save the evidence, then explain which requirement would force you to choose a different design.

Use lockfiles with purpose

Treat the lockfile as the reproducible resolution for development and CI while publishing compatible ranges for consumers.

CI uses a clean install from the committed lockfile, while library consumers resolve dependencies within the published ranges. This is a small part of a small `slugify-title` package that is tested locally, consumed from another project, and published safely, but the boundary it creates affects everything that follows.

The manifest declares allowed versions; the lockfile records one resolved graph with integrity data. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

Watch for one shortcut in particular: trying to delete the lockfile whenever an update behaves unexpectedly. It makes later failures harder to locate. Instead, commit and review lockfile changes, use clean installs in CI, and update dependencies in small evidence-backed batches. Prefer an implementation you can explain from the outside before optimizing it.

Keep the wider goal in view: Make dependency and version choices that communicate compatibility rather than merely making installation succeed. A short observation with concrete evidence is more useful than a confident sentence with no reproduction path.

Change one dependency, inspect the lockfile diff, reproduce the install in a clean directory, and record the tested graph. Hand the result to someone else and see whether the behavior is clear without an oral explanation.

Check your understanding: versions and dependencies

Check semantic versions, ranges, lockfiles, dependency types, peers, and update evidence.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Local package development

Build, test, consume, and develop packages through scripts and workspaces.

Design package scripts

Give common development, test, build, lint, and release checks stable commands.

Before choosing a tool or writing more code, make the requirement concrete. The same `npm test` and `npm run build` commands work locally, in CI, and just before publishing.

Scripts are the package interface for maintainers and CI; they should describe outcomes and compose predictably. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

A common failure is to hide essential setup in an undocumented personal shell command. The happy path may still work, which makes the mistake easy to miss. create a small script vocabulary, keep platform-specific shell tricks out, and make failures stop the pipeline. Keep the first version small enough that every important input and output remains visible.

This supports the module goal: Develop the package against a real consumer and make build and test behavior reproducible from a clean checkout. Capture the request, command, trace, screenshot, test result, or other evidence that proves the result.

Delete generated output, install cleanly, and run the documented scripts in the order a new maintainer would. Change one condition on purpose, predict the result, and compare the prediction with what actually happened.

Test from a real consumer

Install the packed artifact in another project so source-tree shortcuts cannot hide packaging mistakes.

Start from the behavior you can observe. A fixture application installs the tarball and imports every documented entry point using the supported runtime.

Tests inside the package can pass while exports, filenames, declarations, or runtime dependencies fail after installation. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

It is tempting to test consumers through relative paths into the package source. That trades a clear decision for an assumption you will eventually have to debug. The stronger move is to pack the package, install that artifact in an isolated consumer, and run public API examples there. Make the boundary explicit in code and in the project notes.

Tie the decision back to the larger job: Develop the package against a real consumer and make build and test behavior reproducible from a clean checkout. Record enough evidence that another

developer can repeat the result without relying on your memory.

Create CommonJS or ESM consumers as appropriate and verify runtime imports, types, errors, and documented examples. Repeat it once with an invalid or hostile condition and write down the boundary the failure revealed.

Use workspaces with clear boundaries

Develop related packages together without erasing the fact that each package is published and versioned separately.

The repository contains the package and a demo app; the app depends on the workspace package by its package name. That contrast gives us something useful to test instead of a rule to memorize.

Workspaces coordinate installation and scripts, but each workspace still needs correct dependencies, exports, and packed contents. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

The happy path can hide a bad design: you can rely on hoisted dependencies that the package never declares and still get one successful demo. Push past the demo. declare dependencies in the workspace that imports them and test packed packages outside the workspace layout, then test the condition most likely to prove the choice wrong.

The module is moving toward one outcome: Develop the package against a real consumer and make build and test behavior reproducible from a clean checkout. Save the before-and-after evidence now; it will make the final project review much more honest.

Run a production install for each workspace and locate one dependency that was accidentally available only through hoisting. Save the evidence, then explain which requirement would force you to choose a different design.

Publish types and build output

Compile only when necessary and align JavaScript entry points, source maps, declarations, and package exports.

The TypeScript version emits JavaScript and declarations into matching export targets, then verifies both from the consumer fixture. This is a small part of a small `slugify-title` package that is tested locally, consumed from another project, and published safely, but the boundary it creates affects everything that follows.

A build exists to produce a supported consumer artifact, not because every package needs a dist directory. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

Watch for one shortcut in particular: trying to point exports at source files that the supported runtime cannot execute. It makes later failures harder to locate. Instead, map every export to an existing built file and declaration, clean stale output, and test the packed artifact. Prefer an implementation you can explain from the outside before optimizing it.

Keep the wider goal in view: Develop the package against a real consumer and make build and test behavior reproducible from a clean checkout. A short observation with concrete evidence is more useful than a confident sentence with no reproduction path.

Break one declaration path intentionally, observe the consumer failure, and add a release check that

catches it. Hand the result to someone else and see whether the behavior is clear without an oral explanation.

Check your understanding: local package development

Check scripts, lifecycle behavior, local consumers, workspaces, build outputs, and type declarations.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Publish safely

Protect registry identity, run a release gate, publish, verify, and recover.

Prepare the registry identity

Confirm package scope, access, ownership, account protection, and registry target before the first release.

Before choosing a tool or writing more code, make the requirement concrete. The scoped utility has explicit public access, multiple maintainers where appropriate, and a protected npm account.

Publishing changes an immutable public namespace, so package and maintainer identity deserve the same care as code. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

A common failure is to discover the active registry and account only after running the publish command. The happy path may still work, which makes the mistake easy to miss. verify registry, login identity, package name, access level, owners, two-factor or trusted publishing, and recovery details. Keep the first version small enough that every important input and output remains visible.

This supports the module goal: Publish an intentional immutable artifact with strong account controls and a release record you can verify. Capture the request, command, trace, screenshot, test result, or other evidence that proves the result.

Run the read-only identity and package checks from a clean shell and save a release checklist without tokens. Change one condition on purpose, predict the result, and compare the prediction with what actually happened.

Run the release gate

Make tests, build, consumer checks, metadata, tarball inspection, and version state block publishing.

Start from the behavior you can observe. The gate refuses dirty files, stale build output, failing tests, an already-used version, or unexpected tarball contents.

A release should be the output of repeatable checks, not a manual command run from an uncertain working tree. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

It is tempting to publish first and plan to repair the package if consumers report a problem. That trades a clear decision for an assumption you will eventually have to debug. The stronger move

is to run one release command that performs clean install, tests, build, pack inspection, consumer verification, and metadata checks. Make the boundary explicit in code and in the project notes.

Tie the decision back to the larger job: Publish an intentional immutable artifact with strong account controls and a release record you can verify. Record enough evidence that another developer can repeat the result without relying on your memory.

Start from a dirty tree and a stale dist file, then prove the release gate catches both before any network write. Repeat it once with an invalid or hostile condition and write down the boundary the failure revealed.

Publish and verify

Publish the exact inspected artifact, then install it from the registry and verify metadata and provenance.

The release records the tarball integrity, tag, version, commit, checks, and a post-publish smoke test. That contrast gives us something useful to test instead of a rule to memorize.

A successful publish response is not the final proof; the registry artifact and fresh consumer install are. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

The happy path can hide a bad design: you can rebuild between inspection and publishing or verify only from the local workspace and still get one successful demo. Push past the demo. publish the already-checked state, inspect registry metadata, and install the exact version in a clean consumer, then test the condition most likely to prove the choice wrong.

The module is moving toward one outcome: Publish an intentional immutable artifact with strong account controls and a release record you can verify. Save the before-and-after evidence now; it will make the final project review much more honest.

Practice with a private test scope or dry run, then write the commands and evidence required for the real release. Save the evidence, then explain which requirement would force you to choose a different design.

Recover from a bad release

Use deprecation, a fixed follow-up version, dist-tags, and clear communication instead of rewriting history.

A faulty release is deprecated with a specific message, the latest tag moves only after a fixed version passes the gate, and affected users get upgrade guidance. This is a small part of a small `slugify-title` package that is tested locally, consumed from another project, and published safely, but the boundary it creates affects everything that follows.

Published versions are normally immutable; recovery means steering consumers toward a known-good version. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

Watch for one shortcut in particular: trying to reuse the same version number or silently move tags without explaining the impact. It makes later failures harder to locate. Instead, assess exposure, stop further promotion, deprecate precisely, publish a tested fix, and document the affected range. Prefer an implementation you can explain from the outside before optimizing it.

Keep the wider goal in view: Publish an intentional immutable artifact with strong account controls and a release record you can verify. A short observation with concrete evidence is more useful than a confident sentence with no reproduction path.

Simulate a broken latest release and write the exact recovery timeline, commands, consumer message, and verification steps. Hand the result to someone else and see whether the behavior is clear without an oral explanation.

Check your understanding: publish safely

Check registry identity, authentication, dry runs, provenance, dist-tags, deprecation, and release recovery.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Maintain the package

Protect compatibility, communicate changes, update dependencies, and retire responsibly.

Protect the public contract

Test documented behavior and types so refactors cannot quietly break consumers.

Before choosing a tool or writing more code, make the requirement concrete. Contract tests run against the packed artifact and preserve the documented handling of Unicode and duplicate separators.

The public API includes runtime behavior, errors, types, import paths, environment support, and sometimes performance characteristics. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

A common failure is to treat a passing internal unit test as proof that every consumer contract survived. The happy path may still work, which makes the mistake easy to miss. turn documented examples and previous bug fixes into black-box tests of the public package. Keep the first version small enough that every important input and output remains visible.

This supports the module goal: Maintain the package as a promise to consumers, with explicit support, measured changes, and a retirement path. Capture the request, command, trace, screenshot, test result, or other evidence that proves the result.

Refactor the implementation aggressively while keeping consumer tests unchanged; investigate every failure as a contract question. Change one condition on purpose, predict the result, and compare the prediction with what actually happened.

Write useful release notes

Explain what changed, who is affected, how to upgrade, and whether action is required.

Start from the behavior you can observe. The minor release notes show the new option, default behavior, compatibility, and a copy-paste upgrade example.

Consumers need consequences and migration steps more than a list of commit messages. The useful

target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

It is tempting to generate a changelog that names files but never explains user-visible behavior. That trades a clear decision for an assumption you will eventually have to debug. The stronger move is to organize notes around added, changed, fixed, deprecated, and security impact with concrete migration guidance. Make the boundary explicit in code and in the project notes.

Tie the decision back to the larger job: Maintain the package as a promise to consumers, with explicit support, measured changes, and a retirement path. Record enough evidence that another developer can repeat the result without relying on your memory.

Write release notes for one patch, minor, and major change to the utility and ask a consumer to predict the work required. Repeat it once with an invalid or hostile condition and write down the boundary the failure revealed.

Update dependencies with evidence

Review ownership, release notes, graph changes, install scripts, tests, and packed output before accepting updates.

A small update changes one lockfile region, passes the consumer fixture, and does not add an unexpected install script. That contrast gives us something useful to test instead of a rule to memorize.

Dependency updates are code changes from another trust boundary and deserve proportionate review. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

The happy path can hide a bad design: you can merge a large automated batch because every version number is newer and still get one successful demo. Push past the demo. update in reviewable groups, inspect the graph and package metadata, run the full gate, and record why the change is safe, then test the condition most likely to prove the choice wrong.

The module is moving toward one outcome: Maintain the package as a promise to consumers, with explicit support, measured changes, and a retirement path. Save the before-and-after evidence now; it will make the final project review much more honest.

Compare the dependency tree before and after one update and explain every new package and lifecycle script. Save the evidence, then explain which requirement would force you to choose a different design.

Deprecate and retire

Give consumers a clear successor, migration path, support date, and archived security expectations.

The final release removes secrets and automation, points to a maintained alternative, and states whether security reports are still handled. This is a small part of a small `slugify-title` package that is tested locally, consumed from another project, and published safely, but the boundary it creates affects everything that follows.

Abandoned packages create lasting risk because installations continue after maintainers stop watching them. The useful target is not encyclopedic coverage. Make one deliberate choice, observe its consequences, and know which requirement would make you revise it.

Watch for one shortcut in particular: trying to delete the repository or unpublish without warning and leave consumers guessing. It makes later failures harder to locate. Instead, announce a timeline, deprecate versions with a useful message, document migration, and preserve the namespace safely. Prefer an implementation you can explain from the outside before optimizing it.

Keep the wider goal in view: Maintain the package as a promise to consumers, with explicit support, measured changes, and a retirement path. A short observation with concrete evidence is more useful than a confident sentence with no reproduction path.

Write a retirement notice and test that a new install clearly directs the user to the replacement. Hand the result to someone else and see whether the behavior is clear without an oral explanation.

Check your understanding: maintain the package

Check compatibility, changelogs, support ranges, dependency risk, automation, and package retirement.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

The interactive version of this course is available at https://flaviocopes.com/courses/npm-package-publishing/.