THE PRACTICAL GUIDE
FOR MODERN DEVELOPERS

MODERN JAVASCRIPT TOOLING IN 2026

NODE.JS, BUN, PNPM, VITE, TYPESCRIPT, BIOME, VITEST & BEYOND

Node.js


Bun


pnpm


Vite

TypeScript


Biome


Vitest


BUILD FASTER
Modern tools for high-performance development


CODE WITH CONFIDENCE
Type-safe, linted, and tested code by default


END-TO-END WORKFLOWS
From setup to deployment and beyond


PRODUCTION READY
Best practices for security, testing, and scaling

</>

PRACTICAL. COMPREHENSIVE. UP TO DATE.

EVERYTHING YOU NEED TO BUILD MODERN JAVASCRIPT APPS IN 2026

DANIEL HAYES

Modern JavaScript Tooling in 2026

Node.js, Bun, pnpm, Vite, TypeScript, Biome, Vitest & Beyond

Steve Publications

This book is available at
https://leanpub.com/modernjavascripttoolingin2026

This version was published on 2026-08-27

Contents

Node.js, Bun, pnpm, Vite, TypeScript, Biome, Vitest & Beyond

This book is a practical, technically rigorous guide to the modern JavaScript and TypeScript development ecosystem as it exists in 2026. It covers everything from package management and transpilation to bundling, testing, linting, deployment, and security. Each tool is explained not just in terms of how to use it but why it exists, what problems it solves, when to choose it over alternatives, and how it fits into an end-to-end production workflow. The content is organized as a progressive learning journey from fundamentals through professional-grade development practices, with complete code examples, configuration walkthroughs, comparison tables, and real-world project scenarios. Whether you are starting fresh or modernizing existing projects, this book gives you the knowledge to design, build, test, optimize, publish, and deploy production-quality JavaScript applications in 2026.

Introduction: The State of JavaScript Tooling in 2026

The JavaScript tooling ecosystem in 2026 looks nothing like it did five years ago. That is not an exaggeration. If you picked up a guide to modern JavaScript development from 2021, much of it would be outdated or misleading today. Webpack was the default bundler back then. Jest was unquestionably the test runner. ESLint plus Prettier was the only code quality stack worth considering. Node.js had no serious competition as a runtime. Those statements are all wrong now.

The ecosystem has not fragmented further, which might seem like the natural trajectory. Instead, it has consolidated and matured around a smaller set of tools that are faster, more interoperable, and designed to work together. Rust-based implementations dominate performance-critical layers. Native ES modules support is finally widespread enough to stop treating CommonJS as anything but legacy. TypeScript has moved from optional enhancement to the default language for serious JavaScript development. And the gap between what works locally on your machine and what works in CI or production has narrowed dramatically due to better caching, lock files, and containerization practices.

This book is written for developers who want to understand this ecosystem at a professional level. Not as a collection of isolated tool tutorials but as an integrated system where each piece has a purpose, tradeoffs, and relationships with everything else. You will learn how to choose between Node.js and Bun for your runtime, why pnpm might save you hours per week on large projects, when Biome can replace both ESLint and Prettier without sacrificing quality, how Vitest makes testing feel instantaneous instead of painful, and what it actually takes to publish a TypeScript library that works correctly for every consumer regardless of their own toolchain.

The structure follows a deliberate learning progression. We start with foundational concepts that explain why the ecosystem looks the way it does: module systems, runtime architectures, and the historical decisions that shaped today's tools. Then we move through each major layer of the development

stack in order: package management, transpilation, bundling, code quality, testing, debugging, distribution, frontend workflows, backend development, deployment, and security. Each chapter builds on previous ones so that by the time you reach later sections about monorepos or CI/CD pipelines, you understand not just the commands to run but why those commands exist and what problems they solve.

The code examples are complete and runnable. Every configuration file shown can be copied directly into a project and used as-is. Commands include their expected output so you know when things are working correctly. Where tools interact with each other, the examples show that integration explicitly rather than treating each tool in isolation. This is not theoretical documentation but practical guidance based on how real projects are structured and maintained in 2026.

A few conventions apply throughout the book. File paths are shown relative to project roots unless noted otherwise. Commands assume a standard Unix-like shell; Windows users should use PowerShell or Git Bash with equivalent syntax. All examples default to TypeScript where appropriate since it is now the recommended choice for production development, but JavaScript behavior and differences are called out explicitly whenever they matter. ESM versus CommonJS distinctions are treated seriously because getting them wrong causes real bugs that are difficult to debug. Development-time and production-time configurations are kept separate throughout because conflating them leads to security vulnerabilities and performance problems.

By the end of this book you will be able to look at any JavaScript or TypeScript project in 2026 and understand its toolchain at a glance: why it uses that particular package manager, how its build pipeline is structured, what testing strategy it employs, and whether its deployment configuration follows current best practices. More importantly, when starting a new project or evaluating tools for an existing one, you will have the knowledge to make informed decisions based on tradeoffs rather than trends.

The JavaScript ecosystem moves quickly. Tools that are cutting edge today may be obsolete tomorrow. The goal of this book is not to freeze a snapshot of what is popular at any given moment but to give you the underlying understanding that makes you resilient to change. When a new bundler appears or a linter adds features or a runtime gains compatibility, you will know how to evaluate it against your real requirements instead of following hype cycles. That is what professional-level tooling knowledge looks like, and that is what

this book delivers.

Chapter 1: Foundations (Modules, Runtimes, and the Modern Landscape)

Before touching any specific tool, you need to understand the foundational concepts that explain why JavaScript tooling exists in its current form. Every bundler, transpiler, package manager, and linter exists to solve problems created by earlier design decisions or environmental constraints. This chapter covers those foundations: module systems, how JavaScript actually runs, the historical forces that shaped the ecosystem, and the runtime choices available in 2026.

The Module Problem: From Globals to ESM

JavaScript was originally designed as a simple scripting language for web pages. It had no concept of modules. Every script loaded into a page executed in the same global scope, sharing variables freely. This worked fine when scripts were small and few in number. It stopped working when web applications grew beyond toy examples.

The first solution was convention-based: wrap code in immediately invoked function expressions (IIFEs) to create private scopes, expose only what you need on a global object like `MyApp`, and hope nobody else uses the same name. This is how jQuery, Backbone, and early React were distributed. It worked but required discipline that large teams rarely maintained consistently.

The next solution was CommonJS, created by Node.js in 2009. Every file became its own module with a private scope. You exported values using `module.exports` and imported them using `require()`. This was synchronous: when you called `require('./file')`, the system read that file from disk, executed it, and returned its exports before continuing. It worked perfectly for server-side code where files live on a local filesystem.

Browsers did not adopt CommonJS because it does not fit how browsers load resources. HTTP requests are asynchronous. You cannot synchronously read a file over the network without blocking the entire page. Instead, the

ECMAScript specification introduced ES modules (ESM) in 2015 as the official standard. ESM uses static `import` and `export` statements that can be analyzed before execution, enabling optimizations like tree shaking where unused code is eliminated from bundles.

For over a decade after ESM was standardized, JavaScript developers lived with two competing module systems side by side. Node.js continued defaulting to CommonJS for backward compatibility while slowly adding ESM support through file extensions (`.mjs`) and package-level configuration (`"type": "module"`). Browsers only understood ESM. Build tools had to translate between the two constantly, creating a layer of complexity that affected everything from how libraries were published to how errors appeared in stack traces.

By 2026 this situation has improved significantly but not disappeared entirely. Node.js fully supports both systems with reasonable interoperability. Modern bundlers handle ESM natively and convert CommonJS on demand. New projects overwhelmingly use ESM as their primary module system. But legacy code still exists, some popular packages ship only in CommonJS format, and understanding how these two systems interact remains essential knowledge for any JavaScript developer working with real-world projects.

CommonJS vs ES Modules: Mechanics, Interop, and When It Matters

The differences between CommonJS and ES modules are not just syntactic. They represent fundamentally different approaches to how modules load, export values, and interact with each other. Understanding these differences prevents subtle bugs that appear only in specific environments.

CommonJS uses dynamic exports through `module.exports`. You can assign anything to it at any time during module execution:

```
// math.js (CommonJS)
function add(a, b) {
  return a + b;
}

module.exports = { add };
// Or: module.exports.add = add;
// Or: module.exports = add; (single export)
```

Importing uses `require()`, which is synchronous and returns the exported value as an object:

```
// app.js (CommonJS)
const { add } = require('./math');
const result = add(2, 3); // 5
```

ES modules use static `import` and `export` statements that must appear at the top level of a file:

```
// math.js (ES module)
export function add(a, b) {
  return a + b;
}

export const VERSION = '1.0.0';
```

Importing uses named or default imports:

```
// app.js (ES module)
import { add, VERSION } from './math.js';
const result = add(2, 3); // 5
```

The critical mechanical differences are:

CommonJS exports are mutable value copies. When a module sets `module.exports`, other modules receive that object reference at the time of require. If the exporting module later mutates that object, changes are visible to importers. However, reassigning `module.exports` creates a new object that previous importers do not see.

ES modules use live bindings. When you import a named export, you get a live reference to that binding in the exporting module. If the exporting module later changes what that variable points to, all importers see the change immediately:

```
// config.js (ES module)
export let apiUrl = 'https://api.example.com';

// updater.js (ES module)
import { apiUrl } from './config.js';
console.log(apiUrl); // 'https://api.example.com'

// If config.js later does: apiUrl = 'https://new-api.example.com';
// Then updater.js reads the new value on next access
```

CommonJS resolution is path-based and synchronous. `require('./file')` reads from the filesystem relative to the current module. No file extension is required because Node.js tries `.js`, `.json`, and `.node` in order. This works because everything lives on a local disk.

ES modules require explicit file extensions for relative imports. `import './file'` without an extension throws an error. You must write `import './file.js'`. Absolute imports (like `import lodash from 'lodash'`) go through package resolution. This requirement exists because ESM was designed for browsers where files are loaded over HTTP and extensions matter for content negotiation.

How Node.js determines whether a file is CommonJS or ES module depends on three factors, checked in order: the `.mjs` extension always means ES module, the `.cjs` extension always means CommonJS, and for `.js` files the nearest `package.json` with `"type": "module"` makes them ES modules while `"type": "commonjs"` (or no type field) makes them CommonJS.

Interoperability between the two systems works but has limitations. When an ES module imports a CommonJS module, Node.js wraps the CommonJS exports as a default export and creates synthetic named exports based on properties of `module.exports`. This allows `import { something } from './commonjs-module'` to work in many cases but not all:

```
// commonjs-lib.js (CommonJS)
module.exports = { name: 'test', version: 1 };

// esm-consumer.js (ES module)
import lib from './commonjs-lib.js'; // Works: gets the whole exports object
import { name, version } from './commonjs-lib.js'; // Works: synthetic named
↪  exports
```

When CommonJS requires an ES module, Node.js 22 and later support this through `require(esm)` which returns a Promise. Earlier versions of Node.js

throw an error because synchronous require cannot load asynchronous ES modules. This was historically the biggest blocker for ESM adoption in Node.js projects, and its resolution in Node.js 22 removed that barrier:

```
// commonjs-consumer.js (CommonJS, Node.js 22+)
const esmModule = await import('./esm-module.js'); // Dynamic import always worked
const loaded = require('./esm-module.js'); // New in Node.js 22: synchronous for some cases
```

The practical guidance for 2026 is clear. New projects should use ES modules exclusively by setting `"type": "module"` in `package.json`. Use `.js` extensions for source files and include file extensions in all relative imports. When publishing packages, provide ESM-first exports through the `exports` field with CommonJS fallbacks only if you have concrete evidence that consumers need them. Mixing module systems within a single project creates unnecessary complexity and bugs that are difficult to trace.

How JavaScript Runs: V8, Sandboxing, and Runtime Architecture

JavaScript does not execute directly on hardware like C or Rust. It runs inside a runtime environment that provides the execution engine, standard library APIs, filesystem access, networking capabilities, and everything else needed for code to do useful work. Understanding how these runtimes are architected explains why certain tools behave the way they do and why performance characteristics differ between environments.

The dominant JavaScript engine is V8, developed by Google and used in both Chrome and Node.js. V8 converts JavaScript source code into machine code through a multi-stage compilation process. First, the Ignition interpreter parses and executes code immediately without optimization, providing fast startup times. As the engine identifies hot code paths that execute frequently, the TurboFan optimizing compiler kicks in to generate highly optimized machine code for those sections. This tiered compilation approach balances startup speed with long-term performance.

V8 manages memory through a generational garbage collector. Most objects die young, so V8 keeps recently allocated objects in a small, fast

"young generation" heap and only moves surviving objects to the larger "old generation" heap during promotion cycles. Full garbage collection of the old generation is expensive and triggers stop-the-world pauses where all JavaScript execution halts. Understanding this model matters for performance: creating many short-lived objects is cheap, but holding references to large objects that are no longer needed causes memory leaks that force expensive collection cycles.

Node.js builds on V8 by adding non-JavaScript components that provide system-level capabilities. libuv handles asynchronous I/O operations like file reading and network requests through an event loop and thread pool. OpenSSL provides cryptographic functions. Zlib handles compression. These are all written in C or C++ and exposed to JavaScript through bindings. The result is a runtime where JavaScript code can perform blocking operations asynchronously without freezing the event loop.

The Node.js event loop is central to how everything works. It continuously processes pending callbacks from I/O operations, timers, and microtasks (Promises). JavaScript code runs single-threaded on one CPU core for the main thread. When you make an HTTP request or read a file, the operation is delegated to libuv which uses system threads or kernel-level async I/O, then schedules a callback when complete. This allows Node.js to handle thousands of concurrent connections without creating a thread per connection, but it also means any synchronous blocking operation freezes all other work until it completes.

Bun takes a different architectural approach. Instead of V8, it uses JavaScriptCore, the engine that powers Safari. JavaScriptCore is smaller and faster at startup than V8 but historically had less aggressive optimization for long-running server workloads. Bun compensates by implementing many Node.js APIs directly in Zig rather than through C++ bindings, reducing overhead. It also includes a built-in bundler, test runner, and package manager as first-class components rather than separate tools that communicate over process boundaries. This integration allows operations like running a TypeScript file to happen without intermediate build steps: Bun transpiles on the fly using its own compiler while executing.

Deno also uses V8 but with a fundamentally different security model. Every operation that touches the outside world requires explicit permission flags. Reading a file needs `--allow-read`. Making network requests needs `--allow-net`. Accessing environment variables needs `--allow-env`. This prevents

accidental or malicious code from performing unintended side effects, which matters for scripts run on untrusted code or in shared environments. Deno 2.x added full Node.js compatibility including npm package support, making migration from Node.js practical without rewriting existing code.

The choice of runtime affects everything downstream: what APIs are available, how fast code starts and runs, how well it handles concurrent operations, and what deployment options exist. In 2026 the landscape has stabilized enough that you can make informed choices based on actual requirements rather than speculation about future compatibility or ecosystem support.

The Node.js Legacy: Why It Dominated and What It Left Behind

Node.js launched in 2009 with a simple premise: run JavaScript on the server using the same language developers already used for browser scripting. This was not novel conceptually (other server-side JavaScript implementations existed before it) but Node.js succeeded because it arrived at the right time with the right design choices.

The combination of V8's performance, libuv's async I/O model, and a package ecosystem centered around npm created positive feedback loops that accelerated adoption. Developers familiar with JavaScript could write backend code without learning a new language. The event-driven non-blocking architecture handled concurrent requests efficiently on modest hardware. npm became the largest software registry in the world, making it trivial to find and reuse existing solutions for almost any problem.

By 2015 Node.js was the default runtime for server-side JavaScript development. Microservices architectures emerged as a popular pattern because Node.js made it easy to deploy many small services rather than one large monolith. Build tools like Webpack and Gulp were themselves Node.js applications, creating an ecosystem where tooling and application development shared the same platform.

This dominance came with costs that became apparent over time. The event loop model that enabled high concurrency also meant that CPU-intensive operations blocked everything else unless moved to worker threads or separate processes. The CommonJS module system worked fine for local file systems but complicated browser compatibility and prevented tree shaking

optimizations. npm's early package resolution algorithm created deeply nested dependency trees with duplicate packages consuming disk space. Security was an afterthought: millions of packages on the registry with no verification of publisher identity, no malware scanning, and trivial supply chain attack vectors through typosquatting or compromised maintainer accounts.

Node.js addressed many of these issues gradually but slowly due to its commitment to backward compatibility. ES modules support arrived years after browser implementation because breaking existing CommonJS code was unacceptable. Performance improvements required careful benchmarking against existing workloads. Security features like provenance attestations and trusted publishing were added only after high-profile supply chain attacks demonstrated their necessity.

The legacy Node.js left behind is both an asset and a constraint. The ecosystem of packages, patterns, and developer knowledge built around it represents enormous collective investment that cannot be recreated overnight. Every alternative runtime must provide sufficient compatibility with existing code to make migration worthwhile. At the same time, Node.js's size and complexity make radical changes difficult. New runtimes can start fresh with modern defaults because they do not carry the same backward compatibility burden.

Modern Alternatives: Bun, Deno, and the New Competition

By 2026 three runtimes dominate serious JavaScript development: Node.js, Bun, and Deno. Each has distinct strengths, weaknesses, and ideal use cases. Understanding these differences allows you to choose based on actual project requirements rather than brand recognition or inertia.

Node.js remains the safe default for production workloads that require maximum compatibility and stability. As of 2026, Node.js 24 (codenamed Krypton) is the Active LTS release with support until April 2028. Node.js 22 (Jod) is in Maintenance LTS until April 2027 and remains widely deployed. The ecosystem around Node.js is unmatched: every major framework, library, and tool supports it first or exclusively. Production operational knowledge (debugging memory leaks, profiling performance, configuring load balancers, setting up monitoring) is well documented and broadly available across engineering

teams. If your priority is minimizing risk and maximizing the pool of developers who can work on your codebase, Node.js is still the correct choice.

Bun reached production stability with version 1.0 in September 2023 and continued maturing through 2025 and 2026 with versions reaching 1.4+ by mid-2026. It positions itself as an all-in-one toolkit: runtime, package manager, bundler, and test runner in a single binary. The performance advantages are real and measurable. Bun installs dependencies 5 to 30 times faster than npm depending on project size. Its JavaScriptCore-based startup is significantly faster than Node.js for many workloads. Built-in TypeScript support means you can run `.ts` files directly without separate compilation steps.

Bun's Node.js compatibility reached approximately 98 percent by early 2026, which covers the vast majority of packages and patterns used in typical applications. The remaining gaps tend to appear with packages that rely on internal Node.js APIs or unusual edge cases in event loop behavior. For greenfield projects using mainstream libraries and frameworks, Bun works without modification in most cases. Existing large projects should test their specific dependency set before migrating because the compatibility gap, while small, can break things in production.

The pragmatic recommendation for many teams is to use Bun selectively: `bun install` for fast dependency installation, `bun run` for executing scripts during development, and Node.js for production deployment until compatibility confidence grows further. This approach captures Bun's developer experience benefits without risking production stability on a newer runtime.

Deno took a different path by prioritizing security and modern web standards over backward compatibility with the existing Node.js ecosystem. Deno 1.x launched in 2020 with native TypeScript support, a permission-based security model, and standard library modules distributed as URLs rather than through npm. These features were compelling but adoption was slow because migrating from Node.js meant rewriting imports, replacing familiar APIs with Deno-specific alternatives, and dealing with an ecosystem that had fewer packages available.

Deno 2.0, released in October 2024, changed the calculus by adding full Node.js compatibility and npm package support. You can now import npm packages directly using the `npm:` prefix or install them through a Deno-managed lock file. The permission model remains opt-in rather than forced, allowing gradual adoption of stricter security policies. Deno Deploy provides

an edge runtime platform for deploying JavaScript code globally without managing infrastructure.

Deno's ideal use cases are projects where security matters: scripts that process untrusted input, internal tools run by non-technical team members, and applications deployed to edge environments where the permission model prevents accidental data leaks. For general-purpose server-side development, Deno competes directly with Node.js on feature parity while offering TypeScript support without separate compilation steps.

The three-way comparison in 2026 looks like this:

Criterion	**Node.js**	**Bun**	**Deno 2.x**
Ecosystem compatibility	100 percent (it is the standard)	~98 percent	~95 percent with npm support
Startup time	Slowest	Fast	Moderate
TypeScript support	Requires separate compilation or ts-node	Built-in, runs .ts files directly	Built-in, runs .ts files directly
Security model	Traditional, trust-based	Traditional, trust-based	Permission-based, security-first
Production track record	15+ years	Growing (since 2023)	Mature for specific workloads
Package manager	npm bundled separately	Built-in bun install	Built-in deno add
Best for	Maximum compatibility and stability	Speed and developer experience	Security and edge deployments

No single runtime is universally superior. The right choice depends on your project's constraints: how much existing code you need to run, how important startup time is, whether security guarantees matter, and what deployment targets you support.

Choosing Your Runtime: Decision Framework for 2026

Selecting a runtime is one of the first architectural decisions you make when starting a new project or evaluating a migration. The decision should be based on concrete factors rather than trends or preferences. Here is a practical decision framework.

Start with Node.js if:

- You are building a production application where stability and broad team familiarity are priorities
- Your project depends on packages that use internal Node.js APIs or have known incompatibilities with alternative runtimes
- You need to deploy to environments where only Node.js is officially supported by your infrastructure provider
- Your team has limited experience with JavaScript runtime internals and you want the most documented, well-supported option

Consider Bun if:

- Developer experience and iteration speed are critical (fast installs, instant TypeScript execution, integrated tooling)
- You are starting a new project with mainstream dependencies that have been tested on Bun
- Startup time matters for your workload (serverless functions, CLI tools, development servers)
- You want to consolidate multiple tools into one binary without sacrificing functionality

Evaluate Deno if:

- Security is a primary concern and you want permission-based access controls as a default
- You are deploying to edge environments through Deno Deploy or similar platforms
- Native TypeScript support without build steps is a requirement
- Your project benefits from web-standard-first APIs rather than Node.js-specific conventions

For many teams the pragmatic answer is hybrid: use Bun for development tooling where its speed advantages matter most, and deploy on Node.js for production until your compatibility testing gives you confidence to switch. This approach has become common enough that most major frameworks and tools support this pattern explicitly.

The runtime decision also affects downstream choices. If you choose Node.js, you have the widest selection of frameworks, databases drivers, and deployment options. Bun's ecosystem is growing rapidly but still narrower. Deno's ecosystem is focused around its own patterns and platforms. These constraints compound over time as your project grows and dependencies accumulate.

Ultimately the runtime is an infrastructure choice that should be revisited periodically as the landscape evolves. Migration between runtimes is rarely zero-cost but has become significantly easier in 2026 than it was five years ago due to improved compatibility across all major options. The most important principle is choosing based on your actual requirements today while keeping migration paths open for tomorrow.

Chapter 2: Package Management (npm, pnpm, Corepack, and the Dependency Model)

Package management is the foundation of every JavaScript project. Every tool you use, every framework you build with, every utility function you import comes from packages distributed through a registry and installed by a package manager. How dependencies are resolved, stored, versioned, and updated affects everything from disk usage and install times to security vulnerabilities and reproducibility across environments. This chapter covers the complete dependency management landscape in 2026: what `package.json` actually means, how npm and pnpm differ architecturally, why Corepack exists, how monorepos organize shared code, and how to keep your dependency tree secure and up to date.

The package.json Contract: Fields, Scripts, and Modern Conventions

Every JavaScript project begins with a `package.json` file. This single JSON document defines your project's identity, its dependencies, the module system it uses, the scripts you can run, and how other packages should consume your code when published. Understanding every field in this file prevents subtle bugs that appear only in specific environments or consumer configurations.

The essential fields are:

- `name`: A unique identifier for your package on the registry. Must be lowercase, URL-safe, and cannot start with a dot or underscore. Scoped packages use the format `@scope/name` where scope is typically your organization name.
- `version`: Follows semantic versioning (semver) in the format `MAJOR.MINOR.PATCH`. Major versions indicate breaking changes, minor

versions add backward-compatible features, patch versions fix bugs without changing behavior.

- `description`: A one-line summary that appears in registry listings and search results.
- `main`: The entry point for CommonJS consumers using `require()`. Points to a file path relative to `package.json`, typically `./dist/index.js` or `./src/index.js`. This field is legacy but still required if you support CommonJS.
- `module`: A non-standard field originally created by Rollup that points to an ES module entry point for bundlers. Many tools check this before falling back to `main`. It is being superseded by the more flexible `exports` field.
- `type`: Determines how `.js` files in your package are interpreted. `"module"` means ES modules, `"commonjs"` (or omitting the field entirely) means CommonJS. This single field is one of the most common sources of module system bugs when set incorrectly.
- `exports`: The modern replacement for `main`, `module`, and similar fields. Defines conditional exports that specify different entry points based on how your package is consumed:

```
{
  "name": "@myorg/my-lib",
  "version": "1.0.0",
  "type": "module",
  "exports": {
    ".": {
      "types": "./dist/index.d.ts",
      "import": "./dist/index.js",
      "require": "./dist/index.cjs"
    },
    "./utils": {
      "types": "./dist/utils.d.ts",
      "import": "./dist/utils.js"
    }
  }
}
```

The `exports` field maps import specifiers to file paths with conditions. When a consumer imports from your package, Node.js or their bundler checks the conditions in order: `types` for TypeScript declarations, `import` for ES module consumers, `require` for CommonJS consumers. This allows a single package to serve different formats to different environments correctly.

The `exports` field also enables subpath exports so that consumers can import from specific parts of your package without accessing arbitrary internal files. In the example above, `import { helper } from '@myorg/my-lib/utils'` resolves to `./dist/utils.js`, but `import something from '@myorg/my-lib/internal/private'` fails because no such export is defined. This prevents consumers from depending on implementation details that may change between versions.

Dependency fields separate packages by purpose:

- `dependencies`: Packages required at runtime. These are installed in both development and production environments.
- `devDependencies`: Packages needed only during development: test runners, linters, build tools, TypeScript itself. These are not installed when someone installs your published package.
- `peerDependencies`: Packages that your code depends on but expects the consumer to provide. Common for plugins and framework integrations where you want to avoid bundling a duplicate copy of React or Vue with every plugin.
- `optionalDependencies`: Packages that improve functionality if available but are not required. Installation failures for optional dependencies do not fail the overall install.

Version specifications use semver ranges:

- `"lodash": "4.17.21"` : exact version only
- `"lodash": "^4.17.21"` : compatible with 4.17.21, allows updates within the same major version (up to but not including 5.0.0)
- `"lodash": "~4.17.21"` : approximately equivalent, allows only patch-level updates (4.17.x)
- `"lodash": ">=4.17.0 <5.0.0"` : explicit range
- `"*"` : any version (strongly discouraged in production)

The caret (^) is the most common specifier because it balances getting bug fixes and minor improvements with avoiding breaking changes automatically. The tilde (~) is more conservative, useful for packages where even minor versions have historically introduced issues.

Scripts are defined as a `scripts` object mapping command names to shell commands:

```
{
  "scripts": {
    "dev": "vite",
    "build": "tsc && vite build",
    "test": "vitest run",
    "lint": "biome check .",
    "typecheck": "tsc --noEmit"
  }
}
```

Scripts are run with `npm run <name>` or the equivalent for other package managers. Special scripts like `start`, `test`, `preinstall`, and `postinstall` can be invoked directly without `run`. Lifecycle scripts prefixed with `pre` or `post` run before or after their base script: `prebuild` runs before `build`, `postbuild` runs after.

Modern conventions for 2026 include setting `"type": "module"` by default for new projects, using the `exports` field instead of separate `main` and `module` fields, specifying exact versions in lock files rather than loose ranges in `package.json`, and including a `packageManager` field for Corepack integration (covered later in this chapter).

How npm Works: Registry, Resolution, Lock Files, and Installation

npm is the default package manager that ships with Node.js. When you run `node --version`, npm is already installed on your system. This bundling ensures universal availability: every developer has npm regardless of their operating system or how they installed Node.js. In 2026 npm version 11+ is current, with significant performance and reliability improvements over earlier versions that earned a reputation for being slow and unreliable.

The npm registry at `registry.npmjs.org` hosts millions of packages. When you run `npm install lodash`, npm queries the registry for the latest version matching your specified range, downloads the package tarball, extracts it into `node_modules/lodash`, and records the exact version in `package-lock.json`. This lock file is critical: it ensures that every installation of your project uses exactly the same dependency versions regardless of when or where the install happens.

npm's resolution algorithm determines which version of each package to install when multiple dependencies request different versions of the same package. Starting with npm v3, the algorithm flattens the dependency tree whenever possible: if both `package-a` and `package-b` depend on `lodash@^4.17.0`, npm installs a single copy of lodash at the highest matching version rather than duplicating it in nested directories. When versions conflict irreconcilably (one dependency requires lodash 3.x while another requires 4.x), npm creates nested copies to satisfy both requirements.

The lock file format evolved significantly between npm v1 (`npm-shrinkwrap.json`) and modern versions (`package-lock.json`). The current format records the complete dependency tree with exact resolved versions, integrity hashes for each downloaded tarball, and resolution metadata that explains why each version was chosen. This makes installs fully deterministic: given the same `package.json` and `package-lock.json`, npm always produces an identical `node_modules` directory.

The command `npm install` reads `package.json`, checks or creates `package-lock.json`, downloads packages, and installs them into `node_modules`. The command `npm ci` (clean install) is designed for CI environments: it deletes `node_modules`, verifies that `package-lock.json` matches `package.json` exactly, and installs from the lock file without modifying it. This ensures builds are reproducible and fails fast if someone changed dependencies without updating the lock file. Always use `npm ci` in CI pipelines rather than `npm install`.

npm caches downloaded packages in a global cache directory (`~/.npm` on Unix-like systems). Subsequent installs of the same package version read from this cache rather than downloading again, which speeds up repeated installations across projects. The cache can be cleared with `npm cache clean --force` if corruption is suspected, though this is rarely necessary.

One persistent issue with npm's approach is disk space usage. Because npm creates a physical copy of every package in every project's `node_modules`, large projects with many dependencies consume gigabytes of storage. Monorepos with multiple packages each declaring overlapping dependencies compound this problem: the same lodash tarball might be extracted into dozens of nested directories across a single repository. pnpm was created specifically to solve this problem through a fundamentally different storage model.

pnpm: Content Addressable Storage, Symlinks, and Disk Efficiency

pnpm (performant npm) uses a content-addressable store instead of duplicating packages in each project. When you install a package with pnpm, the tarball is extracted once into a global store directory keyed by its cryptographic hash. Your project's `node_modules` directory contains symlinks pointing into this store rather than physical copies of files. If ten projects all depend on lodash@4.17.21, only one copy of lodash exists on disk while each project has its own symlinked reference.

The performance implications are substantial. According to pnpm's own benchmarks updated through August 2026, clean installs (no cache, no lock file) for a large project with many dependencies take approximately 8 seconds with pnpm versus 58 seconds with npm on comparable hardware. Cached installs (lock file present, packages already in store) drop to under half a second with pnpm compared to 1.4 seconds with npm. For teams running installs multiple times daily across many projects, this translates to hours of saved time per week.

pnpm's strict dependency resolution prevents "phantom dependencies" : packages that your code imports but did not explicitly declare in `package.json`. Because pnpm uses symlinks into a flat store rather than nesting packages, importing from `node_modules/some-package` only works if `some-package` is listed as a direct dependency of your project. This catches real bugs where code accidentally depends on a transitive dependency that could be removed by an upstream package at any time. npm and Yarn's flattened trees allow these imports to work until they do not, creating fragile dependencies that break unpredictably.

The tradeoff is compatibility. Some packages assume they can access sibling packages in `node_modules` without declaring them as dependencies. These packages fail under pnpm's strict model unless you configure hoisting rules or add the missing dependency explicitly. Most well-maintained packages work correctly out of the box, but legacy packages or those with unconventional structures may require adjustment.

pnpm supports workspaces (monorepos) natively through a `pnpm-workspace.yaml` file that lists package directories:

```
# pnpm-workspace.yaml
packages:
  - 'apps/*'
  - 'packages/*'
```

Workspaces allow multiple packages in the same repository to share a single dependency store and reference each other using the `workspace:*` protocol instead of downloading from the registry. Changes to a local package are immediately visible to dependent packages without republishing. This is essential for monorepo workflows where shared libraries evolve alongside consuming applications.

pnpm also includes built-in Node.js version management through `pnpm env use`. It downloads and switches between Node.js versions faster than nvm or fnm because it uses the same content-addressable model: each Node.js version is stored once globally and symlinked into projects as needed. Switching versions takes milliseconds instead of seconds.

The recommendation for 2026 is clear: use pnpm for any project where install time, disk space, or dependency strictness matters. This includes monorepos, large teams with many parallel projects, CI environments where build times are billed per minute, and development machines with limited storage. The compatibility gaps with npm are small enough that migration is usually straightforward: run `pnpm import` to convert an existing `package-lock.json` or `yarn.lock` into a `pnpm-lock.yaml`, test your project, and switch.

Corepack: Version Pinning Without nvm or fnm

Corepack solves a different problem than package managers solve: ensuring that every developer and CI environment uses the same version of the package manager itself. Before Corepack existed, teams faced situations where one developer had npm 8 installed globally, another had npm 10, and CI used whatever version shipped with their Node.js image. Different package manager versions could resolve dependencies differently, create incompatible lock files, or run scripts with subtle behavioral differences.

Corepack is bundled with Node.js starting from version 16.9 and enabled by default in Node.js 20+. It acts as a proxy between your terminal commands and the actual package manager binary. When you type `pnpm install`, Corepack

intercepts the call, checks your project's `package.json` for a `packageManager` field, downloads that specific version of pnpm if not already cached, and runs the command with it.

The `packageManager` field uses a simple format:

```
{
  "name": "my-project",
  "version": "1.0.0",
  "packageManager": "pnpm@9.15.0"
}
```

With this field present and Corepack enabled, running `pnpm install` anywhere in the project automatically uses pnpm 9.15.0 regardless of what version is installed globally on the system. New team members cloning the repository get the correct package manager version without manual setup steps. CI pipelines inherit the same behavior.

Corepack supports npm, pnpm, and Yarn out of the box. It can also be configured to manage other tools through the `corepack enable` command followed by registering specific versions with `corepack prepare`. The tool is still marked as experimental in Node.js documentation but has been stable enough for production use that major projects like Vite, Next.js, and Angular rely on it.

Enabling Corepack requires one command: `corepack enable`. After running this globally on your development machine, all projects with a `packageManager` field automatically use their specified tool version. Projects without the field fall back to whichever package manager is installed globally. This opt-in behavior means existing projects are not affected unless you explicitly add the field.

The practical recommendation: add the `packageManager` field to every project's `package.json` and ensure Corepack is enabled on all development machines and CI runners. This eliminates an entire category of environment-related bugs with zero ongoing maintenance cost.

Monorepos and Workspaces: npm, pnpm, and Turborepo Integration

A monorepo stores multiple related packages in a single repository rather than separating them into independent repositories. The JavaScript ecosystem adopted monorepos heavily because they enable shared code across projects without versioning friction, coordinated releases when multiple packages change together, and atomic commits that update consumers and providers simultaneously.

All three major package managers support workspaces (their term for monorepo sub-packages). pnpm's implementation is the most feature-complete and performant, which is why pnpm has become the default choice for JavaScript monorepos in 2026. npm workspaces are functional but lack some advanced features like strict phantom dependency detection. Yarn workspaces with Plug'n'Play offer unique capabilities around zero-install workflows but have narrower ecosystem compatibility.

A typical monorepo structure separates applications from libraries:

```
my-monorepo/
├── apps/
│   ├── web/            # Frontend application
│   └── api/            # Backend service
├── packages/
│   ├── ui/             # Shared UI components
│   ├── utils/          # Utility functions
│   └── config/         # Shared build configs
├── pnpm-workspace.yaml
├── package.json        # Root workspace config
├── turbo.json          # Turborepo task configuration
└── pnpm-lock.yaml
```

The root `package.json` declares the workspace and shared dev dependencies:

```
{
  "name": "my-monorepo",
  "private": true,
  "scripts": {
    "build": "turbo run build",
    "dev": "turbo run dev",
    "test": "turbo run test",
    "lint": "turbo run lint"
  },
  "devDependencies": {
    "turbo": "^2.3.0",
    "typescript": "^5.9.0"
  }
}
```

The `private: true` field prevents accidental publishing of the root workspace as a package. Each sub-package has its own `package.json` with dependencies and scripts appropriate to its role. Internal packages reference each other using the `workspace:*` protocol:

```
{
  "name": "@myorg/web",
  "version": "1.0.0",
  "dependencies": {
    "@myorg/ui": "workspace:*",
    "@myorg/utils": "workspace:*",
    "react": "^19.0.0"
  }
}
```

The `workspace:*` protocol tells pnpm to link directly to the local package source rather than downloading from npm. Changes to `@myorg/ui` are immediately visible in `@myorg/web` without rebuilding or republishing. This enables fast iteration on shared code while maintaining clear dependency boundaries.

Turborepo layers task orchestration and caching on top of the workspace setup. It reads your existing `package.json` scripts and a single `turbo.json` configuration file to determine task dependencies, execution order, and cache keys:

```
{
  "$schema": "https://turbo.build/schema.json",
  "tasks": {
    "build": {
      "dependsOn": ["^build"],
      "inputs": ["src/**/*.ts", "src/**/*.tsx"],
      "outputs": ["dist/**"]
    },
    "test": {
      "dependsOn": ["build"],
      "inputs": ["src/**/*.ts", "tests/**/*.ts"]
    },
    "lint": {
      "inputs": ["src/**/*.ts", ".biome.json"]
    },
    "dev": {
      "cache": false,
      "persistent": true
    }
  }
}
```

The `dependsOn` field declares task dependencies. `"^build"` means this package's build depends on all upstream packages' builds completing first. The `inputs` field lists files that invalidate the cache when changed. The `outputs` field specifies which directories contain build artifacts to cache. Turborepo uses content hashing of inputs to determine whether a task needs re-execution or can use cached output from a previous run.

Running `pnpm build` at the monorepo root triggers `turbo run build`, which executes builds in dependency order, parallelizes independent packages across CPU cores, and skips packages whose inputs have not changed since the last successful build. In CI with remote caching enabled (free on Vercel's infrastructure), Turborepo can restore cached outputs from previous pipeline runs, reducing build times by 70 percent or more for unchanged packages.

The combination of pnpm workspaces and Turborepo has become the standard monorepo stack in 2026. It handles dependency management, task orchestration, caching, and parallelization without requiring custom tooling or framework lock-in. Migration from separate repositories to a monorepo is a significant organizational change that should be justified by actual pain points: version coordination overhead, duplicated code across repos, inability to make atomic changes across packages, or CI costs from building each repo independently.

Dependency Hygiene: Auditing, Updating, and Security Scanning

Dependencies are both an asset and a liability. They give you access to millions of hours of collective engineering work but also introduce vulnerabilities, breakage when upstream packages change behavior, and supply chain attack vectors. Managing this tradeoff requires systematic practices for auditing what you depend on, keeping versions current, and responding to security advisories promptly.

npm includes built-in vulnerability scanning through `npm audit`. This command checks your dependency tree against a database of known vulnerabilities and reports affected packages with severity ratings and suggested fixes:

```
$ npm audit
found 3 vulnerabilities (1 moderate, 2 high) in 1247 scanned packages
  run `npm audit fix` to fix 2 of them.
  run `npm audit fix --force` to fix all issues (may introduce breaking
  ↪  changes).
```

The `npm audit fix` command attempts to update vulnerable packages within the version ranges specified in your `package.json`. The `--force` flag allows updates that exceed those ranges, which may introduce breaking changes and should be tested carefully before merging.

pnpm provides equivalent functionality through `pnpm audit`, which uses the same vulnerability database but reports results in pnpm's format. Both tools integrate with CI pipelines so that builds fail when new vulnerabilities are introduced by dependency updates.

Automated dependency update tools reduce the manual effort of keeping packages current. Dependabot is GitHub's built-in tool that opens pull requests when newer versions of your dependencies become available or when security advisories affect packages you use. It requires zero configuration beyond enabling it in repository settings and works within GitHub's ecosystem natively.

Renovate is a more powerful alternative that supports multiple platforms (GitHub, GitLab, Bitbucket), offers extensive configuration options for grouping updates, defining schedules, and customizing versioning policies, and provides a dependency dashboard for visibility across repositories. The tradeoff is complexity: Renovate's configuration file can become large and requires

maintenance as your project evolves. Many teams use both: Dependabot for security alerts integrated into GitHub's interface, and Renovate for actual update pull requests with custom grouping rules.

A practical update strategy for 2026 balances freshness with stability. Group minor and patch updates together in weekly or biweekly batches rather than creating individual pull requests for every package change. Test grouped updates thoroughly before merging. Reserve major version upgrades for dedicated migration efforts with explicit testing and rollback plans. Pin critical dependencies to exact versions when you have verified they work correctly and want to avoid unexpected breakage from upstream changes.

Dependency auditing should be part of your regular development workflow, not an occasional exercise. Run `npm audit` or `pnpm audit` locally before committing dependency changes. Integrate it into CI so that every pull request checks for newly introduced vulnerabilities. Subscribe to security advisories for packages you depend on heavily. Review the dependency tree periodically with `npm ls` or `pnpm why <package>` to understand why each package is included and whether it can be removed or replaced with a more actively maintained alternative.

The supply chain attacks that targeted npm in 2024 and 2025 demonstrated that vulnerability scanning alone is insufficient. Malicious packages can pass automated checks while containing hidden payloads that activate only under specific conditions. Defense requires layered controls: prefer packages published through trusted publishing pipelines with provenance attestations, limit the permissions of publish tokens used in CI, scan package contents for suspicious patterns beyond known CVE databases, and maintain rollback procedures that allow rapid removal of compromised dependencies when incidents occur. These practices are covered in depth in Chapter 13 on security.

Chapter 3: TypeScript (The Type System That Changed Everything)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernjavascripttoolingin2026.

Why TypeScript Won: Adoption, Tooling, and Developer Experience

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernjavascripttoolingin2026.

tsconfig.json Mastery: Strict Mode, Paths, and Project References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernjavascripttoolingin2026.

Type System Fundamentals: Inference, Generics, and Utility Types

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernjavascripttoolingin2026.

TypeScript with Node.js: Runtime vs Type-Only Concerns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernjavascripttoolingin2026.

TypeScript in the Browser: Bundling, Transpilation, and Distribution

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernjavascripttoolingin2026.

Migration Strategies: Adding Types to Legacy JavaScript Codebases

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernjavascripttoolingin2026.

Chapter 4: Transpilation and Compilation (tsc, SWC, esbuild, Babel)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernjavascripttoolingin2026.

What Transpilation Solves: Targeting Environments That Do Not Exist Yet

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernjavascripttoolingin2026.

TypeScript Compiler (tsc): Capabilities, Limitations, When to Use It

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernjavascripttoolingin2026.

esbuild: Go-Based Speed and the Bundler/Transpiler Blur

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernjavascripttoolingin2026.

SWC: Rust-Powered Compilation for Maximum Performance

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernjavascripttoolingin2026.

Babel's Evolving Role: Plugins, Presets, and Legacy Support

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernjavascripttoolingin2026.

Choosing Your Transpiler: A Decision Matrix

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernjavascripttoolingin2026.

Chapter 5: Bundling (Vite, Rollup, esbuild, and the Modern Build Pipeline)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernjavascripttoolingin2026.

Why We Bundle: Modules, HTTP, and Browser Constraints

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernjavascripttoolingin2026.

Vite Deep Dive: Dev Server, Build Mode, Plugins, and Configuration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernjavascripttoolingin2026.

Rollup: Library Bundling, Tree Shaking, and the ESM Standard

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernjavascripttoolingin2026.

esbuild as a Bundler: Speed Versus Flexibility

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernjavascripttoolingin2026.

Webpack's Remaining Role: When It Still Makes Sense in 2026

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernjavascripttoolingin2026.

Bundle Analysis: Measuring Size, Dependencies, and Performance Impact

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernjavascripttoolingin2026.

Chapter 6: Linting, Formatting, and Code Quality (Biome, ESLint, Prettier)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernjavascripttoolingin2026.

The Code Quality Stack: Why Formatting and Linting Are Separate Concerns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernjavascripttoolingin2026.

Biome: The Rising All-in-One Alternative to ESLint + Prettier

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernjavascripttoolingin2026.

ESLint in 2026: Flat Config, Plugins, and Continued Dominance

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernjavascripttoolingin2026.

Prettier: Opinionated Formatting and the End of Style Debates

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernjavascripttoolingin2026.

Type Checking as Linting: Integrating TypeScript into Your CI

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernjavascripttoolingin2026.

Designing a Code Quality Pipeline: Fast Local Feedback, Strict CI Gates

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernjavascripttoolingin2026.

Chapter 7: Testing (Vitest, Jest, Playwright, and the Testing Pyramid)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernjavascripttoolingin2026.

The Testing Pyramid for JavaScript Applications

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernjavascripttoolingin2026.

Vitest: Vite-Native Unit Testing with esbuild Speed

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernjavascripttoolingin2026.

Jest in 2026: Legacy Strengths and When It Still Belongs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernjavascripttoolingin2026.

Playwright: Cross-Browser E2E Testing and Component Testing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernjavascripttoolingin2026.

Mocking Strategies: Modules, APIs, and External Dependencies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernjavascripttoolingin2026.

Test Configuration for Production Workflows: Coverage, CI, and Flaky Tests

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernjavascripttoolingin2026.

Chapter 8: Debugging, Profiling, and Developer Experience

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernjavascripttoolingin2026.

Source Maps: How They Work and Why They Are Essential

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernjavascripttoolingin2026.

Debugger Tools: Chrome DevTools, VS Code, and Node.js Debugging

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernjavascripttoolingin2026.

Profiling Performance: CPU, Memory, and Startup Time Analysis

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernjavascripttoolingin2026.

Error Tracking in Production: Sentry, LogRocket, and Observability

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernjavascripttoolingin2026.

Logging Strategies: From console.log to Structured JSON Logs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernjavascripttoolingin2026.

DX Optimization: Fast Feedback Loops and Hot Module Replacement

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernjavascripttoolingin2026.

Chapter 9: Building for Distribution (Libraries, CLIs, and npm Publishing)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernjavascripttoolingin2026.

Package Design: API Surface, SemVer, and Breaking Changes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernjavascripttoolingin2026.

Dual ESM/CommonJS Exports: The export Field Map Strategy

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernjavascripttoolingin2026.

TypeScript Declaration Files: Publishing Types with Your Library

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernjavascripttoolingin2026.

CLI Applications: Shebangs, Bin Fields, and Cross-Platform Execution

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernjavascripttoolingin2026.

Publishing to npm: Tokens, Access Control, and Release Automation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernjavascripttoolingin2026.

Versioning Strategies: Changesets, Conventional Commits, and CalVer

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernjavascripttoolingin2026.

Chapter 10: Frontend Development Workflows (Frameworks, SSR, and the Vite Ecosystem)

Vite as the Universal Frontend Toolchain

React in 2026: Server Components, Streaming, and Build Integration

Next.js App Router: SSR, ISR, Edge Functions, and Bundling

SvelteKit and Astro: Alternative Architectures

CSS Tooling: Tailwind, PostCSS, and Modern Styling Approaches

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernjavascripttoolingin2026.

Asset Handling: Images, Fonts, and Content Delivery

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernjavascripttoolingin2026.

Chapter 11: Backend Development (Node.js, Bun, and Server-Side JavaScript)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernjavascripttoolingin2026.

Express in 2026: Still Relevant or Time to Move On?

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernjavascripttoolingin2026.

Fastify, Hono, and Elysia: High-Performance Alternatives

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernjavascripttoolingin2026.

Database Integration: Prisma, Drizzle, and Type-Safe Queries

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernjavascripttoolingin2026.

API Design: REST, GraphQL, tRPC, and OpenAPI

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernjavascripttoolingin2026.

Authentication and Authorization in Modern JavaScript Apps

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernjavascripttoolingin2026.

WebSockets, Server-Sent Events, and Real-Time Communication

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernjavascripttoolingin2026.

Chapter 12: CI/CD, Containers, and Deployment

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernjavascripttoolingin2026.

Continuous Integration for JavaScript: GitHub Actions, GitLab CI, and Build Caching

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernjavascripttoolingin2026.

Dependency Caching Strategies Across CI Providers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernjavascripttoolingin2026.

Containerizing Node.js Applications: Multi-Stage Builds and Minimal Images

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernjavascripttoolingin2026.

Serverless Deployment: Vercel, Netlify, Cloudflare Workers, AWS Lambda

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernjavascripttoolingin2026.

Deploying to Containers: Docker Compose, Kubernetes, and Orchestration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernjavascripttoolingin2026.

Monitoring and Alerting: Health Checks, Metrics, and Uptime

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernjavascripttoolingin2026.

Chapter 13: Security (Dependencies, Runtime, and Supply Chain)

The Dependency Vulnerability Problem: npm Ecosystem Risks

Scanning Tools: npm audit, Snyk, Dependabot, and Renovate

Supply Chain Attacks: Typosquatting, Compromised Packages, and Mitigation

Hardening Node.js Applications in Production

Secrets Management: Environment Variables, Vault Integration, and Runtime Safety

Content Security Policy and Browser-Side Security

Chapter 14: End-to-End Projects (Complete Workflows for Real-World Development)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernjavascripttoolingin2026.

Project 1: A Production TypeScript Library with CI, Testing, and Automated Publishing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernjavascripttoolingin2026.

Project 2: Full-Stack Vite + Node.js Application with Database Integration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernjavascripttoolingin2026.

Project 3: A Monorepo with Turborepo, pnpm Workspaces, Changesets, and Shared Configuration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernjavascripttoolingin2026.

Migration Case Study: Moving from Create React App to Vite with Modern Tooling

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernjavascripttoolingin2026.

Conclusion (Building With Confidence in a Fast-Moving Ecosystem)

References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/modernjavascripttoolingin2026.