Bun Handbook

Flavio Copes
flaviocopes.com

November 2026

Contents

What Bun is, and how it runs your code . . . 3
Installing Bun and setting up a project . . . 6
TypeScript, JSX, and modules . . . 9
Environment variables, watch mode, and hot reload . . . 11
Installing packages and managing `bun.lock` . . . 12
Security, lifecycle scripts, and workspaces . . . 14
Files, streams, and globs . . . 16
Shell scripts and child processes . . . 17
Hashing, passwords, and built-in utilities . . . 19
Built-in SQLite with `bun:sqlite` . . . 20
Unified SQL with `Bun.sql` . . . 24
HTTP servers with `Bun.serve()` . . . 25
WebSockets and real-time updates . . . 29
Testing with `bun:test` . . . 30
Bundling with `bun build` . . . 34
Standalone executables with `bun build --compile` . . . 36
Node.js compatibility and migrating a project . . . 38
When to choose Bun (and when to stay on Node.js) . . . 39

Preface

This is the Bun Handbook, written by Flavio Copes. You can find more programming tutorials and free handbooks at https://flaviocopes.com.

Bun is a JavaScript and TypeScript runtime, like Node.js and Deno. The same `bun` binary is also a package manager, a script runner, a test runner, a bundler, an HTTP server, and a compiler that turns TypeScript into a standalone executable.

What first made Bun interesting to me is how much setup it removes. In a typical JavaScript project you spend the first hour wiring together a runtime, a package manager, a TypeScript runner, a test library, a watcher, and a bundler. With Bun, all of those work out of the box from one binary.

In this book we'll learn Bun from scratch and build a real tool with it. We'll scan the 2,193 published Markdown posts of my blog, index them into SQLite with `bun:sqlite`, serve them through `Bun.serve()` with routes, HTML imports, and WebSockets, automate tasks with `Bun.$` and `Bun.spawn()`, test the whole app with `bun:test`, bundle it with `bun build`, and ship a standalone CLI binary with `bun build --compile`.

Every command, timing, and output in this book comes from a real run on my Mac, in November 2026, on these versions:

Software	Version
Bun	1.4.5 (`1.4.5+a7733435c`)
Node.js (for comparisons)	22.23.4
npm (for comparisons)	10.9.9
TypeScript (installed by `bun init`)	7.0.2

What Bun is, and how it runs your code

JavaScript started as a language for web browsers. A browser gives your code a JavaScript engine plus Web APIs for the DOM, `fetch()`, timers, and storage.

A server-side runtime does the same thing outside the browser. It pairs a JavaScript engine with operating system APIs for files, processes, and networking.

Node.js is the best-known runtime. It uses V8, the JavaScript engine built for Chrome.

Bun uses **JavaScriptCore**, the engine built for Safari and WebKit. Bun started in Zig, and the 1.4 series rewrote large parts of the runtime in Rust with a strong focus on startup speed, low memory overhead, and Node.js compatibility.

One binary instead of a stack of tools

Here is what ships inside the single `bun` executable, compared to the usual Node.js setup:

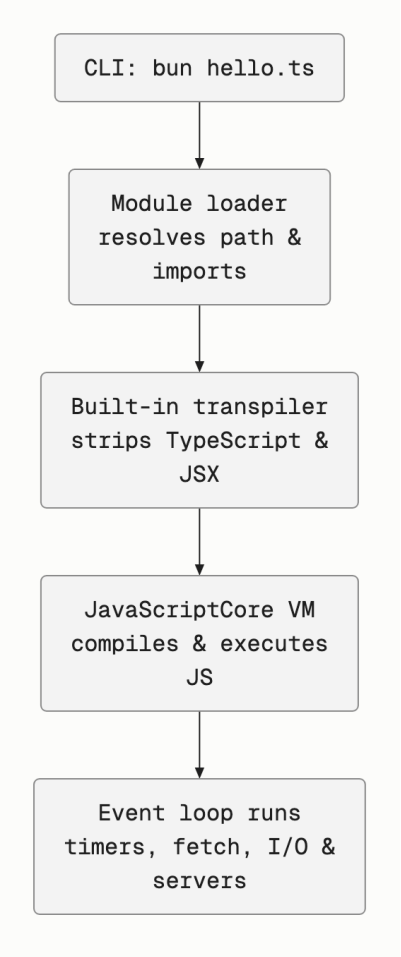
Job	With Bun	Usual Node.js stack
Run JS and TS	`bun file.ts`	`node` + `tsx` or `ts-node`
Type-check	`bun check`	`tsc --noEmit`
Install packages	`bun install`	`npm`, `pnpm`, or `yarn`
Run scripts	`bun run dev`	`npm run dev`
One-off CLIs	`bunx`	`npx`
Watch / reload	`bun --watch`, `bun --hot`	`node --watch` or `nodemon`
Environment files	automatic `.env` loading	`dotenv` or `node --env-file`
Tests & coverage	`bun test`	`node --test`, Vitest, or Jest
Bundler	`bun build`	esbuild, Vite, or Rollup
Single executable	`bun build --compile`	`node --build-sea`
SQLite	`bun:sqlite`	`node:sqlite` or `better-sqlite3`
HTTP & WebSockets	`Bun.serve()`	`node:http` + `ws` or Express

Node.js has been adding more built-in pieces over time (`node --watch`, `node:test`, `node:sqlite`, type stripping). Bun still goes further by designing every piece to share the same parser, resolver, and bundler.

What happens when Bun runs a TypeScript file

No JavaScript engine can execute type annotations like `name: string`. Those annotations have to disappear before JavaScriptCore sees the code.

When you run `bun hello.ts`, Bun does four things in one fast pipeline:



1. The CLI sees a file path instead of a built-in subcommand and starts the runtime.
2. The module loader resolves `hello.ts` and any `import` statements inside it.
3. Bun's native transpiler strips TypeScript syntax (and converts JSX if needed) in memory, without writing temporary `.js` files to disk.
4. JavaScriptCore parses and executes the resulting JavaScript, and the event loop keeps the process alive until all open handles finish.

Because the transpiler runs in native code inside the same process, TypeScript adds almost zero startup overhead.

Startup speed in practice

I timed two tiny files on my Mac, 20 runs each, taking the median:

```
// hello.js
console.log('Hello from Bun')
```

```
// hello.ts
function greet(name: string): string {
  return `Hello ${name}`
}
console.log(greet('Flavio'))
```

Here are the real results against Node.js 22.23.4:

Command	Median	Min	Max
`bun hello.js`	6.4 ms	5.8 ms	12.1 ms
`bun hello.ts`	6.5 ms	6.1 ms	6.9 ms
`node hello.js`	42.7 ms	41.4 ms	73.1 ms
`node hello.ts`	83.8 ms	73.2 ms	102.5 ms

6.4 ms for JavaScript and 6.5 ms for TypeScript. You feel that difference every time you run a CLI script, a git hook, or a test file.

Installing Bun and setting up a project

On macOS and Linux, the official install script puts `bun` in `~/.bun/bin`:

```
curl -fsSL https://bun.sh/install | bash
```

If you prefer Homebrew on macOS:

```
brew tap oven-sh/bun
brew install bun
```

On Windows, run this in PowerShell:

```
powershell -c "irm bun.sh/install.ps1 | iex"
```

You can also install Bun through npm if Node.js is already on the machine:

```
npm install -g bun
```

Check that it works:

```
bun --version
bun --revision
```

```
1.4.5
1.4.5+a7733435c
```

To update later, run `bun upgrade`. It downloads the newest release and replaces the binary in place.

Quick one-liners with `bun -e`

You don't need a file to test an idea. `bun -e` evaluates a snippet directly:

```
bun -e "console.log(2 + 2)"
```

```
4
```

You can use top-level `await` and TypeScript syntax inside `bun -e` too. I use it constantly for quick checks.

Creating a project with `bun init`

For a real project, create a folder and run `bun init`:

```
mkdir demo-p1
cd demo-p1
bun init -y
```

The `-y` flag accepts the defaults for a blank TypeScript project. Here is what Bun 1.4.5 created on my Mac:

```
 + .gitignore
 + CLAUDE.md
 + .cursor/rules/use-bun-instead-of-node-vite-npm-pnpm.mdc -> CLAUDE.md
 + index.ts
 + tsconfig.json (for bun check and editor autocomplete)
 + README.md

To get started, run:

    bun run index.ts

To type check, run:

    bun check

bun install v1.4.5 (a7733435c)
Resolving dependencies
Resolved, downloaded and extracted [61]
Saved lockfile

+ @types/bun@1.4.5
+ typescript@7.0.2

6 packages installed [2.45s]
```

Look at the generated `package.json`:

```
{
  "name": "demo-p1",
  "module": "index.ts",
  "type": "module",
  "private": true,
  "scripts": {
    "typecheck": "bun check"
  },
  "devDependencies": {
```

```
    "@types/bun": "latest"
  },
  "peerDependencies": {
    "typescript": "^7"
  }
}
```

And the generated `tsconfig.json` is already set up for Bun's bundler resolution, `strict: true`, `noUncheckedIndexedAccess: true`, and `noEmit: true`:

```
{
  "compilerOptions": {
    "lib": ["ESNext"],
    "target": "ESNext",
    "module": "Preserve",
    "moduleDetection": "force",
    "jsx": "react-jsx",
    "allowJs": true,
    "types": ["bun"],
    "moduleResolution": "bundler",
    "allowImportingTsExtensions": true,
    "verbatimModuleSyntax": true,
    "noEmit": true,
    "strict": true,
    "skipLibCheck": true,
    "noFallthroughCasesInSwitch": true,
    "noUncheckedIndexedAccess": true,
    "noImplicitOverride": true
  }
}
```

Run the entry point:

```
bun run index.ts
```

```
Hello via Bun!
```

Configuring Bun with `bunfig.toml`

When you want project-wide settings for the runtime, package manager, or test runner, put them in a `bunfig.toml` file at the root of the repository:

```
[install]
exact = true
frozenLockfile = false

[test]
coverage = true
timeout = 10000
```

You can also keep personal defaults in `~/.bunfig.toml`. Project settings override global ones, and command-line flags override both.

TypeScript, JSX, and modules

Running TypeScript directly is great, as long as you know one rule: **Bun strips types, it doesn't type-check them when running a file.**

Let's prove it. Create `bad-type.ts`:

```
const age: number = 'twenty'
console.log('age is', age, typeof age)
```

Run it:

```
bun bad-type.ts
```

```
age is twenty string
```

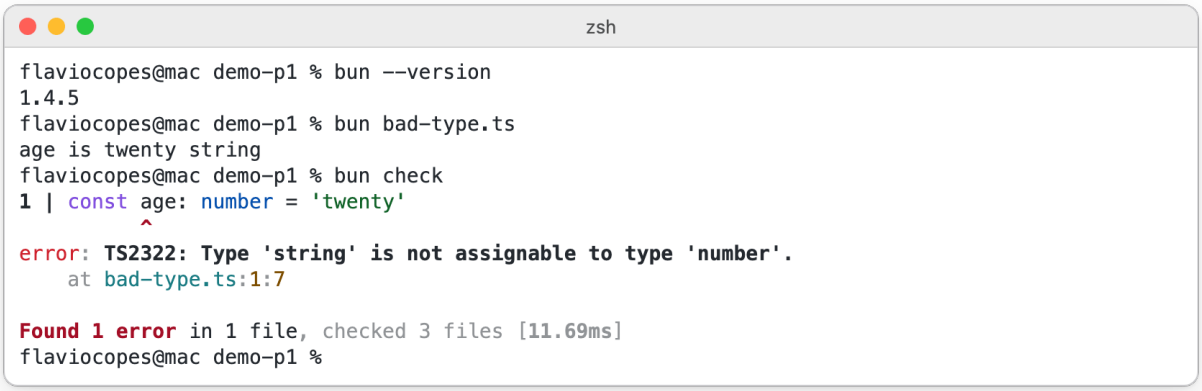
Bun removed `: number`, ran `const age = 'twenty'`, and printed `twenty string`. That's intentional, because skipping type checking at runtime is what keeps startup at 6 milliseconds.

To catch type errors, run `bun check` (new in Bun 1.4, backed by TypeScript 7):

```
bun check
```

```
bad-type.ts(1,7): error TS2322: Type 'string' is not assignable to type 'number'.
Found 1 error in 1 file, checked 3 files [11.25ms]
```

11 milliseconds to type-check the project. You can also run `bunx tsc --noEmit` if you prefer `tsc` directly. Either way, wire `"typecheck": "bun check"` into your CI and run it before shipping.



ES modules and CommonJS

Modern JavaScript uses ES modules (`import` and `export`):

```
import { join } from 'node:path'

console.log(join('src', 'posts', 'bun.md'))
```

Older Node.js packages use CommonJS (`require()` and `module.exports`):

```
const { join } = require('node:path')

console.log(join('src', 'posts', 'bun.md'))
```

Bun supports both, and you can import CommonJS packages from ES modules without any extra config. My advice is to stick to ES modules with `"type": "module"` in `package.json` for all new code.

One handy Bun detail: `import.meta.main` is `true` only when the current file is the entry point you ran, and `false` when another file imports it. We'll use that pattern later so our HTTP server and database indexer can both run from the CLI and be imported by tests:

```
if (import.meta.main) {
  // only runs when executed directly with `bun server.ts`
}
```

Importing JSON, TOML, and text files

Bun's loader understands more than JavaScript and TypeScript. You can import `.json`, `.toml`, and `.txt` files directly, with no parser package needed.

Create `settings.toml`:

```
[site]
name = "flaviocopes.com"
port = 3000
features = ["sqlite", "serve", "test"]
```

Import both `package.json` and `settings.toml`:

```
import pkg from './package.json'
import settings from './settings.toml'

console.log('pkg:', pkg.name)
console.log('site:', settings.site.name, 'port:', settings.site.port)
console.log('features:', settings.site.features.join(', '))
```

```
pkg: demo-p1
site: flaviocopes.com port: 3000
features: sqlite, serve, test
```

When you bundle the project later with `bun build`, Bun inlines those imported files into the bundle automatically.

Environment variables, watch mode, and hot reload

Every project needs configuration that changes between your laptop and production, and a fast way to reload while you edit code. Bun builds both into the runtime.

Automatic `.env` loading

Bun reads `.env` files on startup without needing the `dotenv` package.

Suppose you have `.env`:

```
APP_NAME=Blog Index
PORT=3000
DEBUG=false
```

And `.env.local`:

```
PORT=4000
DEBUG=true
```

Read them through `Bun.env` (or `process.env`, which points to the same object):

```
bun -e "console.log({ app: Bun.env.APP_NAME, port: Bun.env.PORT, debug: Bun.env.DEBUG })"
```

```
{
  app: "Blog Index",
  port: "4000",
  debug: "true",
}
```

Real environment variables from your shell still win over every file:

```
PORT=5000 bun -e "console.log('override:', Bun.env.PORT)"
```

```
override: 5000
```

The precedence order, from lowest to highest:

1. `.env`
2. `.env.development` (or `.env.production` / `.env.test` depending on `NODE_ENV`)
3. `.env.local` (skipped when `NODE_ENV=test` so local overrides don't break tests)
4. Shell environment variables

If you want to load a specific file instead, pass `--env-file=.env.staging`. And never commit `.env.local` or any file with real secrets to Git.

`--watch` vs `--hot`

Bun has two reload modes for development:

Mode	Command	What happens on save	Good for
Watch	`bun --watch server.ts`	Restarts the whole process	Most apps, clean state every time
Hot	`bun --hot server.ts`	Reloads changed modules in place	Servers where you want to keep connections or in-memory state

Watch mode is easier to reason about because every save gives you a fresh process with no leftover state. Hot reload keeps `globalThis` and an active `Bun.serve()` socket alive, swapping out only the updated modules.

Pay attention to where the flag goes when running a `package.json` script:

```
bun --watch run dev
```

Flags for Bun itself go right after `bun`. If you put `--watch` at the end (`bun run dev --watch`), Bun passes `--watch` as an argument to the script instead.

Installing packages and managing `bun.lock`

Bun's package manager is compatible with the npm registry, `package.json`, and `node_modules`. The commands are short and familiar:

```
bun install             # install everything in package.json
bun add hono            # add a dependency
bun add -d typescript   # add a devDependency
bun remove hono         # remove a package
bun update              # update within semver ranges
bun outdated            # list newer versions available
```

Real install speed: Bun vs npm

I created two identical folders with four packages (`hono`, `zod`, `yaml`, `picocolors`) and timed `bun install` against `npm install --no-audit --no-fund`:

```
bun install v1.4.5 (a7733435c)
Resolving dependencies
Resolved, downloaded and extracted [12]
Saved lockfile

+ hono@4.13.23
+ picocolors@1.1.1
+ yaml@2.9.1
+ zod@3.25.76 (v4.6.7 available)

4 packages installed [324.00ms]
```

Command	Wall time	Notes
`bun install` (first run)	0.32 s	Downloads and links from global cache
`bun ci` (cached, no changes)	0.003 s (3 ms)	Verifies lockfile and `node_modules`
`npm install --no-audit --no-fund`	0.78 s	Same 4 packages

On larger projects with hundreds of dependencies, the gap widens to seconds vs tens of seconds, because Bun keeps one global content-addressed cache and links files with copy-on-write clones on macOS (`clonefile`) and hardlinks on Linux.

The text lockfile: `bun.lock`

Older versions of Bun wrote a binary `bun.lockb` file that was impossible to review in a Git diff. Modern Bun writes `bun.lock`, a plain JSONC text file:

```
{
  "lockfileVersion": 2,
  "configVersion": 1,
  "workspaces": {
    "": {
      "name": "bench",
      "dependencies": {
        "hono": "^4.7.0",
        "picocolors": "^1.1.1",
        "yaml": "^2.7.0",
        "zod": "^3.24.0",
      },
    },
  },
  "packages": {
    "hono": ["hono@4.13.23", "", {}, "sha512-..."],
    "picocolors": ["picocolors@1.1.1", "", {}, "sha512-..."],
    "yaml": ["yaml@2.9.1", "", { "bin": { "yaml": "bin.mjs" } }, "sha512-..."],
    "zod": ["zod@3.25.76", "", {}, "sha512-..."],
  }
}
```

Commit `bun.lock` to Git. In CI, always run:

```
bun ci
```

`bun ci` is shorthand for `bun install --frozen-lockfile`. If `package.json` and `bun.lock` don't match, it fails instead of silently updating the lockfile.

Inspecting dependencies with `bun pm`

When you want to know what's installed or why a package ended up in `node_modules`, use `bun pm`:

```
bun pm ls
```

```
bench node_modules (4 installed)
+-- hono@4.13.23
+-- picocolors@1.1.1
+-- yaml@2.9.1
+-- zod@3.25.76
```

```
bun pm why yaml
```

```
yaml@2.9.1
  +- bench (requires ^2.7.0)
```

`bun pm ls --all` shows transitive dependencies too, and `bun pm cache rm` clears the global package cache if you ever need a fresh download.

Security, lifecycle scripts, and workspaces

Supply-chain attacks on npm often work through `postinstall` scripts: you run `npm install`, and a compromised package executes arbitrary code on your machine immediately.

Lifecycle scripts are blocked by default

Bun refuses to run `preinstall`, `install`, and `postinstall` scripts from third-party dependencies unless you explicitly allow them.

If a package you installed needs its install script (for example a native addon or a binary downloader), check which packages were blocked:

```
bun pm untrusted
```

Read what the script does first. If you trust the package, allow it:

```
bun pm trust package-name
```

That adds the package name to `trustedDependencies` in `package.json`, so your team and CI get the same explicit allowlist.

You can also add `minimumReleaseAge` in `bunfig.toml` (in seconds) so Bun refuses to install newly published versions that are younger than a few days, giving the ecosystem time to catch compromised releases:

```
[install]
minimumReleaseAge = 259200 # 3 days
```

Running scripts and `bunx`

`bun run <script>` runs entries from `package.json`:

```
{
  "scripts": {
    "dev": "bun --watch server.ts",
    "test": "bun test",
    "typecheck": "bun check"
  }
}
```

Always keep the `run` keyword when invoking a script whose name collides with a Bun built-in. `bun run test` runs the `"test"` script from `package.json`, while `bun test` launches Bun's test runner directly.

For one-off CLI tools you don't want to install globally, use `bunx` (Bun's equivalent of `npx`):

```
bunx cowsay "Hello from Bun"
```

Workspaces and parallel scripts

When a repository holds more than one package (say a shared library and a CLI), declare `"workspaces"` in the root `package.json`:

```
{
  "name": "blog-monorepo",
  "private": true,
  "workspaces": ["packages/*"]
}
```

In `packages/shared/package.json`:

```
{
  "name": "@blog/shared",
  "version": "1.0.0",
  "exports": "./index.ts",
  "scripts": {
    "check": "echo 'shared ok'"
  }
}
```

In `packages/cli/package.json`, reference the local package with `"workspace:*"`:

```
{
  "name": "@blog/cli",
  "version": "1.0.0",
  "dependencies": {
    "@blog/shared": "workspace:*"
  },
```

```
  "scripts": {
    "check": "echo 'cli ok'"
  }
}
```

Because Bun executes TypeScript directly, `@blog/shared` can export `./index.ts` with no build step between packages:

```
// packages/cli/index.ts
import { readingTime } from '@blog/shared'
console.log('1800 words ->', readingTime(1800), 'min read')
```

```
bun install
bun packages/cli/index.ts
```

```
1800 words -> 9 min read
```

And you can run a script across every workspace at once with `--parallel` and `--filter`:

```
bun run --parallel --filter '*' check
```

```
@blog/shared:check | shared ok
@blog/cli:check    | cli ok
@blog/shared:check | Done in 1ms
@blog/cli:check    | Done in 2ms
```

Files, streams, and globs

Bun implements Web APIs (`fetch`, `Request`, `Response`, `URL`, `FormData`, `Blob`, `ReadableStream`) and adds fast native helpers for local files.

`Bun.file()` and `Bun.write()`

`Bun.file(path)` returns a lazy `BunFile` reference. Creating it doesn't open or read the file yet. Bun waits until you call `.text()`, `.json()`, `.bytes()`, `.arrayBuffer()`, or `.stream()`.

```
const pkg = await Bun.file('package.json').json()
console.log(pkg.name)

await Bun.write('message.txt', 'Hello from Bun\n')
```

`Bun.write()` accepts strings, `Uint8Array`, `ArrayBuffer`, `Response`, and another `BunFile`. Passing a `BunFile` copies it using the fastest OS syscall available:

```
await Bun.write('message-copy.txt', Bun.file('message.txt'))
```

You can also check `.exists()` and `.size` on a `BunFile`, or stream a large file straight into an HTTP `Response` without loading it into memory:

```
return new Response(Bun.file('./archive.zip'))
```

For directory operations like `mkdir`, `readdir`, or `rm`, reach for `node:fs/promises`. Bun implements `node:fs` natively, so mixing `Bun.file()` with `node:fs/promises` is completely normal.

Scanning files with `Bun.Glob`

To find files matching a pattern without installing `fast-glob` or `glob`, use `Bun.Glob`. Here I scan all Markdown files in my blog's `src/posts` directory:

```
import { Glob } from 'bun'
import path from 'node:path'

const postsDir = process.argv[2]
const glob = new Glob('*.md')
const t0 = performance.now()
let total = 0
let totalBytes = 0

for await (const name of glob.scan(postsDir)) {
  const file = Bun.file(path.join(postsDir, name))
  total += 1
  totalBytes += file.size
}

const ms = (performance.now() - t0).toFixed(1)
console.log(`Scanned ${total} files (${(totalBytes / 1024 / 1024).toFixed(2)} MB) in ${ms} ms`)
```

```
Scanned 2243 files (17.29 MB) in 7.8 ms
```

2,243 files in 7.8 milliseconds. `glob.scanSync()` is also available when you want a synchronous loop in a short script.

Shell scripts and child processes

Scripts often need to call external tools like `git`, `ffmpeg`, or `sqlite3`. Bun gives you two ways to do that: the `$` shell template tag and `Bun.spawn()`.

The built-in shell: `Bun.$`

`$` runs shell commands inside TypeScript using a tagged template literal, and it works across macOS, Linux, and Windows (Bun implements common builtins like `ls`, `rm`, `cp`, `echo`, and pipes itself):

```
import { $ } from 'bun'

const branch = (await $`git rev-parse --abbrev-ref HEAD`.text()).trim()
const count = (await $`ls src/posts | wc -l`.text()).trim()
console.log({ branch, markdownFiles: Number(count) })
```

```
{
  branch: "master",
  markdownFiles: 2243,
}
```

The best part of `$` is **automatic escaping**. Any value interpolated with `${...}` is escaped as a single literal argument, never parsed as shell syntax:

```
const unsafe = 'hello; echo INJECTED'
const out = (await $`printf '%s\n' ${unsafe}`.text()).trim()
console.log('escaped:', out)
```

```
escaped: hello; echo INJECTED
```

`echo INJECTED` never ran. That eliminates the classic shell-injection bug where a filename or user string containing `;`, `|`, or `$(...)` executes extra commands.

By default, `$` throws a `ShellError` if the command exits with a non-zero status. Call `.nothrow()` when you want to inspect `exitCode` yourself, or `.quiet()` when you don't want output echoed to the terminal.

`Bun.spawn()` and `Bun.spawnSync()`

When you want direct control over processes, streaming pipes, or signals without a shell layer, use `Bun.spawn()` with an array of arguments:

```
const proc = Bun.spawn(['git', 'log', '-n', '1', '--oneline'], {
  stdout: 'pipe',
  stderr: 'pipe',
})

const [exitCode, stdout, stderr] = await Promise.all([
  proc.exited,
  new Response(proc.stdout).text(),
  new Response(proc.stderr).text(),
])

console.log({ exitCode, latestCommit: stdout.trim(), stderr: stderr.trim() })
```

```
{
  exitCode: 0,
  latestCommit: "dff4e6c75 plans: Bun handbook outline [skip ci]",
  stderr: "",
}
```

Two details to remember:

- Passing an array skips the shell completely, so filenames with spaces stay intact and metacharacters can't be injected.

- Pipe `stderr: 'pipe'` explicitly if you want to capture errors, since Bun inherits `stderr` by default.

	`Bun.$`	`Bun.spawn()`
Best for	Build scripts, pipes, quick automation	Servers, streaming I/O, fine control
Syntax	Tagged template `$`cmd ${arg}``	Argument array `['cmd', arg]`
Shell injection safe	Yes (auto-escapes `${...}`)	Yes (no shell invoked)
Pipes & redirects	Built-in (‘	, >, &&‘)
Non-zero exit	Throws by default (use `.nothrow()`)	Resolves with `proc.exited` code

Hashing, passwords, and built-in utilities

A lot of small utility packages you'd normally install from npm are already in the `Bun` global.

Password hashing with `Bun.password`

Never store plain-text passwords, and don't use fast hashes like SHA-256 for passwords. `Bun.password` implements `argon2id` (the default) and `bcrypt` in native code:

```
const t0 = performance.now()
const hash = await Bun.password.hash('correct-horse-battery-staple', {
  algorithm: 'argon2id',
})
const ms = (performance.now() - t0).toFixed(1)

const ok = await Bun.password.verify('correct-horse-battery-staple', hash)
const bad = await Bun.password.verify('wrong-password', hash)
console.log('argon2id:', hash.slice(0, 48) + '...', `(${ms} ms)`)
console.log({ ok, bad })
```

```
argon2id: $argon2id$v=19$m=65536,t=2,p=1$tQrhlKtH7sNldixe6... (59.4 ms)
{
  ok: true,
  bad: false,
}
```

59 milliseconds per hash: fast enough for a login request, slow enough to punish brute-force attacks. `verify()` reads the algorithm, salt, and cost parameters straight from the stored hash string.

Fast hashing, crypto, and compression

- `Bun.hash(input)` computes a fast non-cryptographic 64-bit hash (wyhash), useful for cache keys and deduplication.
- `new Bun.CryptoHasher('sha256')` computes cryptographic digests (`sha256`, `sha512`, `blake2b256`).
- `Bun.gzipSync()` and `Bun.gunzipSync()` (plus `deflateSync` and `zstdCompressSync`) compress buffers without importing `node:zlib`.

```
console.log('Bun.hash (wyhash):', Bun.hash('flaviocopes.com'))
const sha = new Bun.CryptoHasher('sha256').update('flaviocopes.com').digest('hex')
console.log('sha256:', sha)

const sample = await Bun.file('src/posts/bun.md').text()
const gz = Bun.gzipSync(sample)
console.log(`gzip: ${sample.length} bytes -> ${gz.byteLength} bytes (${((1 - gz.byteLength / sample.
    ↪ length) * 100).toFixed(1)}% smaller)`)
```

```
Bun.hash (wyhash): 6226282639336797062n
sha256: 7a814e81bcec4afc76f703db88359a8b57e6ed61fd90babb1eae78e65a8446f7
gzip: 22644 bytes -> 8820 bytes (61.0% smaller)
```

Markdown, semver, and tables

Bun 1.4 added `Bun.markdown` for fast native Markdown rendering, alongside `Bun.semver` for version checks and `Bun.inspect.table()` for clean CLI output:

```
console.log('semver satisfies:', Bun.semver.satisfies('1.4.5', '^1.4.0'), Bun.semver.order('1.3.11',
    ↪ '1.4.5'))

const html = Bun.markdown.html('# Hello **Bun** 1.4\n\n- Fast\n- Built-in')
console.log('markdown html:', JSON.stringify(html))

console.log(Bun.inspect.table([
  { tool: 'Runtime', bun: 'bun run', node: 'node + tsx' },
  { tool: 'Packages', bun: 'bun install', node: 'npm / pnpm' },
  { tool: 'Tests', bun: 'bun test', node: 'node --test / vitest' },
  { tool: 'Bundler', bun: 'bun build', node: 'esbuild / vite' },
]))
```

```
semver satisfies: true -1
markdown html: "<h1>Hello <strong>Bun</strong> 1.4</h1>\n<ul>\n<li>Fast</li>\n<li>Built-in</li>\n</ul>\
    ↪ n"
+---+----------+-------------+----------------------+
|   | tool     | bun         | node                 |
+---+----------+-------------+----------------------+
| 0 | Runtime  | bun run     | node + tsx           |
| 1 | Packages | bun install | npm / pnpm           |
| 2 | Tests    | bun test    | node --test / vitest |
| 3 | Bundler  | bun build   | esbuild / vite       |
+---+----------+-------------+----------------------+
```

Built-in SQLite with `bun:sqlite`

Bun ships with a native SQLite driver in `bun:sqlite`. You don't need an npm package or a C++ build step, and queries run synchronously in native code with

very low overhead.

Let's use it to index all 2,193 published posts of `flaviocopes.com` into a local `blog.sqlite` file. Create `index-posts.ts`:

```
import { Database } from 'bun:sqlite'
import { Glob } from 'bun'
import path from 'node:path'
import { parse } from 'yaml'

export interface Post {
  slug: string
  title: string
  published_on: string
  words: number
  tags: string[]
  body: string
}

export async function readPosts(dir: string): Promise<Post[]> {
  const glob = new Glob('*.md')
  const today = new Date()
  const posts: Post[] = []

  for await (const file of glob.scan(dir)) {
    const source = await Bun.file(path.join(dir, file)).text()
    const match = source.match(/^---\r?\n([\s\S]*?)\r?\n---\r?\n([\s\S]*)$/)
    if (!match) continue
    const data = parse(match[1]) ?? {}
    const date = new Date(data.date)
    if (data.draft || Number.isNaN(date.getTime()) || date > today) continue
    const body = match[2]
    const tags = [data.tag ?? []].flat().map((t: unknown) => String(t).toLowerCase())
    const words = body.replace(/```[\s\S]*?```/g, '').split(/\s+/).filter(Boolean).length
    posts.push({
      slug: file.replace(/\.md$/, ''),
      title: String(data.title),
      published_on: date.toISOString().slice(0, 10),
      words,
      tags: [...new Set(tags)],
      body,
    })
  }

  posts.sort((a, b) => a.published_on.localeCompare(b.published_on) || a.slug.localeCompare(b.slug))
  return posts
}

export function buildDatabase(dbPath: string, posts: Post[]) {
  const db = new Database(dbPath, { create: true })
  db.run('PRAGMA journal_mode = WAL')
  db.run(`
    CREATE TABLE IF NOT EXISTS posts (
      id INTEGER PRIMARY KEY AUTOINCREMENT,
      slug TEXT NOT NULL UNIQUE,
      title TEXT NOT NULL,
      published_on TEXT NOT NULL,
      words INTEGER NOT NULL,
      tags TEXT NOT NULL,
      body TEXT NOT NULL
    )
  `)
  db.run('DELETE FROM posts')

  const insert = db.query(
    'INSERT INTO posts (slug, title, published_on, words, tags, body) VALUES (?, ?, ?, ?, ?, ?)',
  )
  const insertAll = db.transaction((items: Post[]) => {
    for (const p of items) {
```

```
      insert.run(p.slug, p.title, p.published_on, p.words, p.tags.join(','), p.body)
    }
  })

  const t0 = performance.now()
  insertAll(posts)
  const ms = performance.now() - t0
  return { db, ms }
}
```

Prepared statements and transactions

Three things in `buildDatabase()` make a big difference in real SQLite apps:

1. **`PRAGMA journal_mode = WAL`** turns on write-ahead logging so readers don't block writers and writes are much faster.
2. **`db.query(...)`** prepares the statement once and caches it on the database instance. Call `.all(...)` for every matching row, `.get(...)` for the first row (or `null`), and `.run(...)` for `INSERT`/`UPDATE`/`DELETE`. Always pass values as `?` or `$name` parameters, never string interpolation.
3. **`db.transaction(fn)`** wraps a batch of writes in a single `BEGIN` / `COMMIT` (rolling back automatically if `fn` throws).

I timed inserting 200 rows *without* a transaction against inserting all 2,193 full Markdown posts *inside* one transaction on my Mac:

```
Read 2193 posts in 250.5 ms
200 inserts without transaction: 43.0 ms
2193 inserts in one transaction: 54.6 ms
Stats: {
  posts: 2193,
  words: 2088531,
}
Longest posts: [
  {
    slug: "grok-bot",
    words: 19669,
  }, {
    slug: "claude-code-guide",
    words: 18848,
  }, {
    slug: "livekit",
    words: 16045,
  }
]
```

Without a transaction, every single `INSERT` commits and syncs to disk on its own, so 200 tiny rows took 43 ms. Inside one transaction, all 2,193 posts (over 2 million words of Markdown) went in in 54.6 ms.

For tests or temporary data, pass `':memory:'` instead of a file path: `new Database(':memory:')`.

Full-text search with FTS5 in `bun:sqlite`

SQLite's FTS5 extension is compiled into `bun:sqlite`, so you can build fast full-text search with ranking and highlighted snippets right inside the same file.

Create `fts-demo.ts` to index our 2,193 posts with the `porter` stemmer and query them using named parameters (`$q`, `$limit`):

```
import { Database } from 'bun:sqlite'

const db = new Database('/tmp/bunb/blog-app/blog.sqlite')
db.run('DROP TABLE IF EXISTS posts_fts')
db.run(`
  CREATE VIRTUAL TABLE posts_fts USING fts5(
    slug UNINDEXED,
    title,
    body,
    tokenize = 'porter unicode61'
  )
`)

const t0 = performance.now()
db.run('INSERT INTO posts_fts (slug, title, body) SELECT slug, title, body FROM posts')
const buildMs = (performance.now() - t0).toFixed(1)

const t1 = performance.now()
const hits = db
  .query(
    `SELECT slug, title,
            round(bm25(posts_fts, 0.0, 5.0, 1.0), 2) AS score,
            snippet(posts_fts, 2, '[', ']', '...', 12) AS snippet
     FROM posts_fts
     WHERE posts_fts MATCH $q
     ORDER BY score
     LIMIT $limit`,
  )
  .all({ $q: 'sqlite AND transaction', $limit: 3 })
const searchMs = (performance.now() - t1).toFixed(2)

console.log(`Built FTS5 index in ${buildMs} ms, searched in ${searchMs} ms`)
console.log(hits)
db.close()
```

```
Built FTS5 index in 210.6 ms, searched in 0.86 ms
[
  {
    slug: "node-sqlite",
    title: "node:sqlite: SQLite built into Node.js",
    score: -12.79,
    snippet: "...The [transaction] also makes large batches much faster because [SQLite] doesn't...",
  },
  {
    slug: "durable-agent-approval",
    title: "Build an agent approval flow that survives a restart",
    score: -11.18,
    snippet: "...The run command also claims execution inside a [SQLite] [transaction], which stops
    ↪ ...",
  },
  {
    slug: "do-you-need-a-database",
    title: "Do you always need a database for your app?",
    score: -10.34,
    snippet: "...[SQLite] gives you real SQL, indexes, constraints, and [transactions], but the whole
    ↪ ...",
  }
]
```

210.6 milliseconds to build the index over 2 million words, and 0.86 milliseconds per ranked search. Notice how `bm25()` returns negative numbers (more negative means a stronger match), so `ORDER BY score` puts the best result first, and weighting the title column `5.0` vs body `1.0` boosts posts that mention the search term in their title.

Unified SQL with `Bun.sql`

Alongside `bun:sqlite`, Bun includes `SQL` (and the default `sql` helper) in the `bun` module: one tagged-template SQL client that talks to PostgreSQL, MySQL, and SQLite.

Here is a query against that same `blog.sqlite` file using `SQL`:

```
import { SQL } from 'bun'

const db = new SQL('sqlite:///tmp/bunb/blog-app/blog.sqlite')
const year = '2026'

const rows = await db`
  SELECT slug, title, words
  FROM posts
  WHERE published_on >= ${year + '-01-01'}
  ORDER BY words DESC
  LIMIT 3
`

console.log(rows)
await db.close()
```

```
[
  {
    slug: "grok-bot",
    title: "A deep dive into Grok Bot",
    words: 19669,
  }, {
    slug: "claude-code-guide",
    title: "The complete guide to Claude Code",
    words: 18848,
  }, {
    slug: "livekit",
    title: "LiveKit tutorial: how to build a voice AI agent",
    words: 16045,
  }, count: 3, command: "SELECT", lastInsertRowid: null, affectedRows: 0
]
```

Values inside `${...}` are bound as parameters automatically, not concatenated into SQL, so it's safe against SQL injection.

When should you pick `bun:sqlite` vs `Bun.sql`?

	`bun:sqlite`	`Bun.sql`
Databases	SQLite only	PostgreSQL, MySQL, SQLite
Execution model	Synchronous	Async (`await`)
Connection pooling	Single local file	Built-in pool for Postgres/MySQL
Best for	Local tools, CLIs, single-node SQLite apps	Apps talking to Postgres/MySQL or sharing queries across engines

For a local SQLite database I prefer `bun:sqlite` because the synchronous API is simple and direct. For PostgreSQL or MySQL, `Bun.sql` lets you skip external driver setup altogether.

HTTP servers with `Bun.serve()`

Bun's built-in HTTP server uses standard Web `Request` and `Response` objects. You don't need Express just to serve JSON routes, HTML pages, and static assets.

Declarative routes, HTML imports, and fallbacks

Let's wrap our `blog.sqlite` database in an HTTP server (`server.ts`) that serves an HTML page at `/`, JSON endpoints at `/api/stats`, `/api/posts/:slug`, and `/api/search`, plus a WebSocket endpoint at `/ws`:

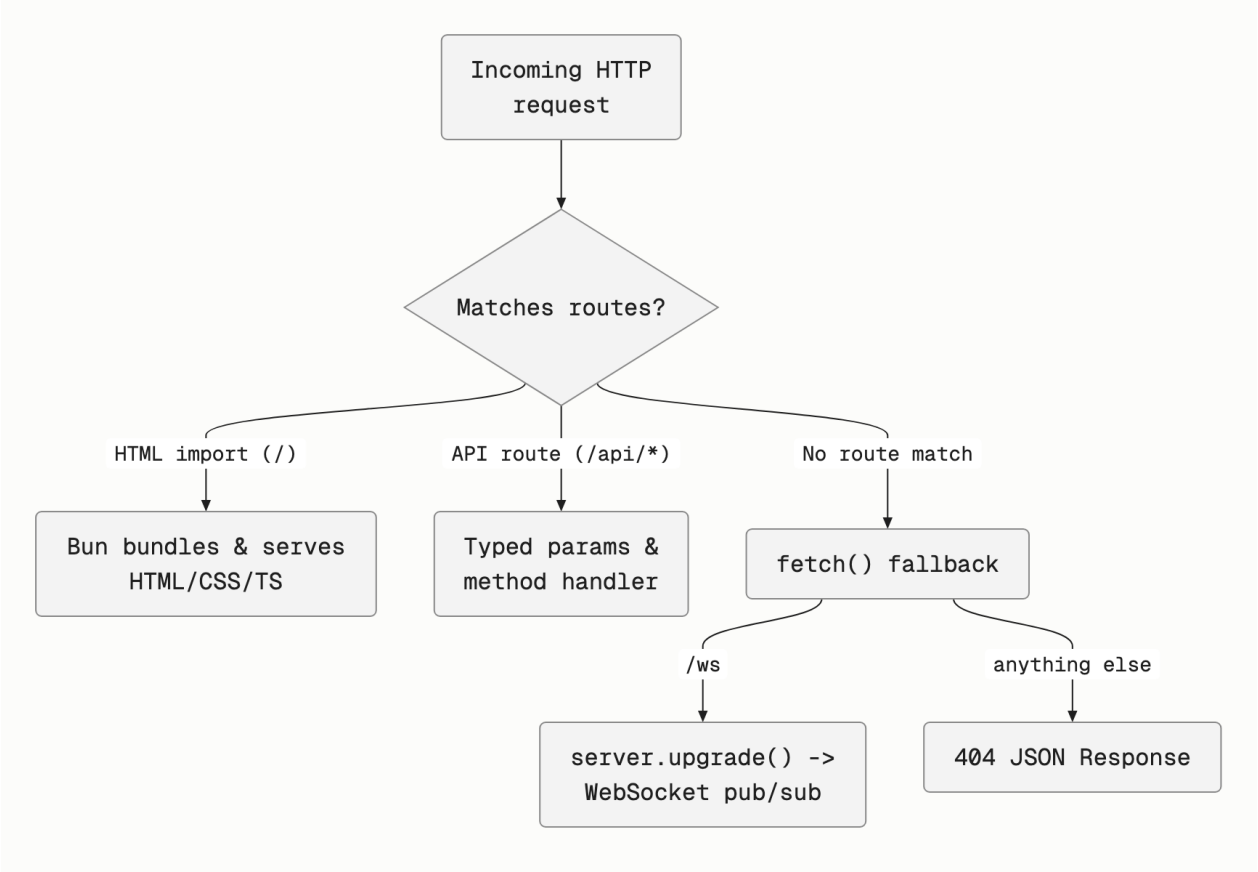
```
<!-- page.html -->
<!doctype html>
<html>
  <head><meta charset="utf-8"><title>Blog Search</title></head>
  <body>
    <h1>Blog Search</h1>
    <p>Powered by Bun.serve()</p>
  </body>
</html>
```

```
// server.ts
import { Database } from 'bun:sqlite'
import page from './page.html'

export function createApp(dbPath: string, port = 0) {
  const db = new Database(dbPath, { readonly: true })
  const searchQuery = db.query(
    `SELECT slug, title, published_on, words
     FROM posts
     WHERE title LIKE ? OR body LIKE ?
     ORDER BY published_on DESC
     LIMIT ?`,
  )
  const bySlug = db.query(
```

```
    'SELECT slug, title, published_on, words, tags FROM posts WHERE slug = ?',
  )
  const statsQuery = db.query(
    'SELECT count(*) AS posts, sum(words) AS words FROM posts',
  )

  const server = Bun.serve({
    port,
    routes: {
      '/': page,
      '/api/stats': () => Response.json(statsQuery.get()),
      '/api/posts/:slug': (req) => {
        const post = bySlug.get(req.params.slug)
        if (!post) return Response.json({ error: 'not found' }, { status: 404 })
        return Response.json(post)
      },
      '/api/search': {
        GET: (req) => {
          const url = new URL(req.url)
          const q = (url.searchParams.get('q') ?? '').trim()
          const limit = Math.min(Number(url.searchParams.get('limit')) || 5, 50)
          if (!q) return Response.json({ error: 'missing q' }, { status: 400 })
          const pattern = `%${q}%`
          const rows = searchQuery.all(pattern, pattern, limit)
          server.publish('searches', JSON.stringify({ q, count: rows.length }))
          return Response.json({ q, count: rows.length, rows })
        },
      },
    },
    fetch(req, server) {
      const url = new URL(req.url)
      if (url.pathname === '/ws') {
        if (server.upgrade(req)) return
        return new Response('WebSocket upgrade failed', { status: 400 })
      }
      return Response.json({ error: 'not found' }, { status: 404 })
    },
    websocket: {
      open(ws) {
        ws.subscribe('searches')
        ws.send(JSON.stringify({ type: 'connected' }))
      },
      message(ws, message) {
        ws.send(JSON.stringify({ type: 'echo', message: String(message) }))
      },
    },
  })

  return { server, db }
}

if (import.meta.main) {
  const { server, db } = createApp('/tmp/bunb/blog-app/blog.sqlite', 3010)
  console.log(`Listening on ${server.url}`)
  process.on('SIGTERM', () => {
```

```
    server.stop(true)
    db.close()
    process.exit(0)
  })
}
```

Look at how clean the request pipeline is:



Notice two nice details:

- **HTML imports (`import page from './page.html'`)**: if `page.html` links `<script src="./app.tsx">` or `<link rel="stylesheet" href="./style.css">`, Bun automatically transpiles and bundles them in development and production, with no Vite config required.
- **`port = 0`**: when tests pass `0`, the operating system picks a free ephemeral port and `server.url` tells you what it chose, so tests never collide with a running dev server.

Let's hit the running server with `curl`:

```
curl -s http://localhost:3010/api/stats
```

```
{"posts":2193,"words":2088531}
```

```
curl -s http://localhost:3010/api/posts/bun
```

```
{"slug":"bun","title":"Bun: a faster JavaScript runtime","published_on":"2026-08-07","words":2693,"tags
    ↪ ":"js"}
```

```
curl -s 'http://localhost:3010/api/search?q=sqlite&limit=2'
```

```
{"q":"sqlite","count":2,"rows":[{"slug":"ai-product-engineering","title":"AI product engineering:
    ↪ building reliable AI features after the demo","published_on":"2026-11-28","words":6584},{"slug
    ↪ ":"structured-outputs-few-shot","title":"Structured outputs and few-shot examples: getting
    ↪ reliable data from LLMs","published_on":"2026-11-28","words":3116}]}
```

Streaming responses and built-in cookies

`Bun.serve()` handles streaming responses with standard `ReadableStream` objects, and every route request includes a `req.cookies` map for reading and setting HTTP cookies without installing a cookie parser:

```
const server = Bun.serve({
  port: 0,
  routes: {
    '/stream': () => {
      const stream = new ReadableStream({
        start(controller) {
          controller.enqueue('chunk 1\n')
          controller.enqueue('chunk 2\n')
          controller.close()
        },
      })
      return new Response(stream, {
        headers: { 'content-type': 'text/plain; charset=utf-8' },
      })
    },
    '/visit': (req) => {
      const count = Number(req.cookies.get('visits') ?? '0') + 1
      req.cookies.set('visits', String(count), { httpOnly: true, path: '/' })
      return Response.json({ visits: count })
    },
  },
})

const streamed = await (await fetch(`${server.url}stream`)).text()
const visitRes = await fetch(`${server.url}visit`)
console.log('streamed:', JSON.stringify(streamed))
console.log('visit:', await visitRes.json(), 'set-cookie:', visitRes.headers.get('set-cookie'))
server.stop(true)
```

```
streamed: "chunk 1\nchunk 2\n"
visit: {
  visits: 1,
} set-cookie: visits=1; Path=/; HttpOnly; SameSite=Lax
```

Notice that `req.cookies.set()` adds `SameSite=Lax` automatically, and Bun attaches the `Set-Cookie` header to the `Response` you return.

Rewriting HTML on the fly with `HTMLRewriter`

Bun includes `HTMLRewriter` (the same streaming HTML parser used in Cloudflare Workers), which lets you extract elements or transform HTML without loading a

heavy DOM library like jsdom or Cheerio.

Here I render my Bun blog post with `Bun.markdown.html()`, extract every `<h2>` heading, and add `rel="noopener noreferrer"` to external links:

```
const md = await Bun.file('src/posts/bun.md').text()
const html = Bun.markdown.html(md.replace(/^---[\s\S]*?---\n/, ''))

const headings: string[] = []
let current = ''
const rewriter = new HTMLRewriter()
  .on('h2', {
    element() {
      current = ''
    },
    text(chunk) {
      current += chunk.text
      if (chunk.lastInTextNode && current.trim()) {
        headings.push(current.trim())
      }
    },
  })
  .on('a[href^="http"]', {
    element(el) {
      el.setAttribute('rel', 'noopener noreferrer')
    },
  })

const transformed = rewriter.transform(html)
console.log(`Extracted ${headings.length} h2 headings, first 5:`, headings.slice(0, 5))
console.log('Added rel attribute:', transformed.includes('rel="noopener noreferrer"'))
```

```
Extracted 26 h2 headings, first 5: [ "What is a JavaScript runtime?", "What comes with Bun?",
  "Installing Bun", "Running your first JavaScript program", "Creating a Bun project"
]
Added rel attribute: true
```

Graceful shutdown and `error()`

In production, handle `SIGTERM` and call `server.stop()` so in-flight requests finish before the process exits, then close your database. You can also pass an `error(err)` callback to `Bun.serve()` to catch unhandled exceptions and return a clean 500 response instead of leaking stack traces.

For larger APIs where you want middleware, OpenAPI generation, or structured validation, frameworks like Hono run natively on `Bun.serve()` because they use the same `Request`/`Response` standard.

WebSockets and real-time updates

In Node.js, WebSockets require an extra package like `ws` or Socket.IO. In Bun, WebSockets and topic-based pub/sub are built straight into `Bun.serve()`.

In `server.ts` above:

1. When a client connects to `/ws`, `server.upgrade(req)` upgrades the HTTP connection.

2. In `websocket.open(ws)`, we call `ws.subscribe('searches')` to join the `searches` topic and send a welcome message.
3. Whenever someone hits `/api/search`, the HTTP handler calls `server.publish('searches', ...)`, which broadcasts to every subscribed WebSocket client in native code.

Let's verify it with Bun's built-in WebSocket client:

```
const ws = new WebSocket('ws://localhost:3010/ws')
const msgs: string[] = []

await new Promise<void>((resolve) => {
  ws.onmessage = (e) => {
    msgs.push(String(e.data))
    if (msgs.length === 1) fetch('http://localhost:3010/api/search?q=bun&limit=1')
    if (msgs.length === 2) {
      ws.close()
      resolve()
    }
  }
})

console.log('ws received:', msgs)
```

```
ws received: [ "{\"type\":\"connected\"}", "{\"q\":\"bun\",\"count\":1}" ]
```

The first message is the welcome frame from `open()`, and the second is the live broadcast triggered by the HTTP search request.

Testing with `bun:test`

Bun includes a fast Jest-compatible test runner in `bun:test`. It discovers `*.test.ts`, `*.spec.ts`, and their `.js`/`.tsx` equivalents automatically.

Let's write a real integration test suite (`blog.test.ts`) for our SQLite indexer and HTTP server, plus a mock check:

```
import { afterAll, beforeAll, describe, expect, mock, test } from 'bun:test'
import { buildDatabase, type Post } from './index-posts'
import { createApp } from './server'

const samplePosts: Post[] = [
  {
    slug: 'bun',
    title: 'The Bun guide',
    published_on: '2026-11-28',
    words: 3500,
    tags: ['js'],
    body: 'Bun is fast and includes sqlite.',
  },
  {
    slug: 'sqlite',
```

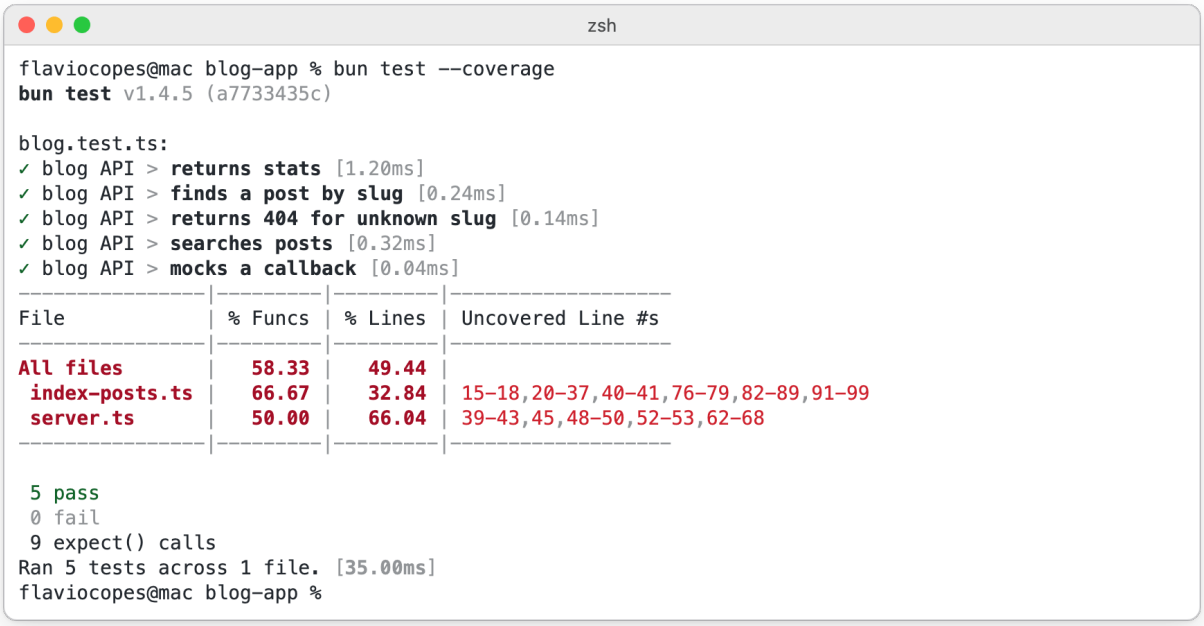
```
    title: 'SQLite basics',
    published_on: '2026-11-20',
    words: 2200,
    tags: ['database'],
    body: 'SQLite is one file.',
  },
]

let server: ReturnType<typeof Bun.serve>
let db: ReturnType<typeof createApp>['db']
const testDbPath = '/tmp/bunb/blog-app/test.sqlite'

beforeAll(() => {
  const built = buildDatabase(testDbPath, samplePosts)
  built.db.close()
  const app = createApp(testDbPath, 0)
  server = app.server
  db = app.db
})

afterAll(() => {
  server.stop(true)
  db.close()
})

describe('blog API', () => {
  test('returns stats', async () => {
    const res = await fetch(`${server.url}api/stats`)
    expect(res.status).toBe(200)
    expect(await res.json()).toEqual({ posts: 2, words: 5700 })
  })

  test('finds a post by slug', async () => {
    const res = await fetch(`${server.url}api/posts/bun`)
    expect(res.status).toBe(200)
    const data = await res.json()
    expect(data.title).toBe('The Bun guide')
  })

  test('returns 404 for unknown slug', async () => {
    const res = await fetch(`${server.url}api/posts/does-not-exist`)
    expect(res.status).toBe(404)
  })

  test('searches posts', async () => {
    const res = await fetch(`${server.url}api/search?q=sqlite`)
    const data = await res.json()
    expect(data.count).toBe(2)
  })

  test('mocks a callback', () => {
    const notify = mock((slug: string) => `indexed:${slug}`)
    expect(notify('bun')).toBe('indexed:bun')
    expect(notify).toHaveBeenCalledTimes(1)
    expect(notify).toHaveBeenCalledWith('bun')
```

```
  })
})
```

Run it with built-in coverage reporting:

```
bun test --coverage


bun test v1.4.5 (a7733435c)

blog.test.ts:
(pass) blog API > returns stats [1.50ms]
(pass) blog API > finds a post by slug [0.20ms]
(pass) blog API > returns 404 for unknown slug [0.12ms]
(pass) blog API > searches posts [0.30ms]
(pass) blog API > mocks a callback [0.08ms]
----------------|---------|---------|-------------------
File            | % Funcs | % Lines | Uncovered Line #s
----------------|---------|---------|-------------------
All files       |   58.33 |   49.44 |
 index-posts.ts |   66.67 |   32.84 | 15-18,20-37,40-41,76-79,82-89,91-99
 server.ts      |   50.00 |   66.04 | 39-43,45,48-50,52-53,62-68
----------------|---------|---------|-------------------

 5 pass
 0 fail
 9 expect() calls
Ran 5 tests across 1 file. [30.00ms]
```

30 milliseconds for a real SQLite build, a real HTTP server, four real HTTP requests, a mock assertion, and a line-by-line coverage table.



Useful flags you'll reach for:

- `bun test blog` filters to test files matching `blog`
- `bun test -t "404"` runs only tests whose name matches `"404"`
- `bun --watch test` re-runs tests whenever a source or test file changes
- `bun test --bail` stops on the first failure
- `bun test --parallel` runs test files concurrently

Keep one caveat in mind: `bun:test` aims for Jest compatibility, not 100% Jest parity. Before migrating a huge legacy Jest suite, run it locally first to see if any custom Jest plugins or jsdom quirks need adjustment.

Table-driven tests, spies, and freezing time

Three features I use in almost every test suite are built right into `bun:test`: `test.each()` for running the same assertion over a table of inputs, `spyOn()` for observing existing methods, and `setSystemTime()` for deterministic date tests:

```
import { describe, expect, setSystemTime, spyOn, test } from 'bun:test'

function readingMinutes(words: number, wpm = 200) {
  return Math.max(1, Math.ceil(words / wpm))
}

describe('readingMinutes', () => {
  test.each([
    [50, 1],
    [200, 1],
    [201, 2],
    [1800, 9],
  ])('%i words -> %i min', (words, expected) => {
    expect(readingMinutes(words)).toBe(expected)
  })
})

test('spies on console.log', () => {
  const spy = spyOn(console, 'log').mockImplementation(() => {})
  console.log('hello')
  expect(spy).toHaveBeenCalledWith('hello')
  spy.mockRestore()
})

test('freezes the clock with setSystemTime', () => {
  setSystemTime(new Date('2026-11-28T12:00:00Z'))
  expect(new Date().toISOString()).toBe('2026-11-28T12:00:00.000Z')
  setSystemTime()
})


extra.test.ts:
(pass) readingMinutes > 50 words -> 1 min [0.07ms]
(pass) readingMinutes > 200 words -> 1 min
(pass) readingMinutes > 201 words -> 2 min
(pass) readingMinutes > 1800 words -> 9 min
(pass) spies on console.log [0.04ms]
(pass) freezes the clock with setSystemTime [0.02ms]
```

```
 6 pass
 0 fail
 6 expect() calls
Ran 6 tests across 1 file. [2.00ms]
```

Calling `setSystemTime()` with no arguments restores the real system clock so later tests aren't stuck in the past.

Bundling with `bun build`

You don't need to bundle server code just to run it with Bun. Bundling becomes useful when you ship JavaScript to browsers, want to combine many modules into one file, or need output targeting Node.js.

Let's write a small CLI (`cli.ts`) that queries our `blog.sqlite` database:

```
import { Database } from 'bun:sqlite'
import { parseArgs } from 'node:util'

const { values, positionals } = parseArgs({
  allowPositionals: true,
  options: {
    db: { type: 'string', default: '/tmp/bunb/blog-app/blog.sqlite' },
    limit: { type: 'string', default: '5' },
  },
})

const query = positionals.join(' ').trim()
if (!query) {
  console.error('Usage: blog-cli <query> [--limit 5] [--db path]')
  process.exit(1)
}

const db = new Database(values.db, { readonly: true })
const pattern = `%${query}%`
const rows = db
  .query(
    'SELECT published_on, slug, words FROM posts WHERE title LIKE ? OR body LIKE ? ORDER BY
    ↪ published_on DESC LIMIT ?',
  )
  .all(pattern, pattern, Number(values.limit)) as {
    published_on: string
    slug: string
    words: number
  }[]

for (const row of rows) {
  console.log(`${row.published_on}  ${row.slug} (${row.words} words)`)
}
db.close()
```

Bundle it for Bun with minification and a source map:

```
bun build ./cli.ts --outdir ./dist --target bun --minify --sourcemap


Bundled 1 module in 3ms

  cli.js      0.70 KB  (entry point)
```

```
  cli.js.map  1.54 KB  (source map)
```

Three targets are built in:

- `--target browser` (the default): bundles for web browsers, resolves browser fields in `package.json`, and supports code splitting (`--splitting`)
- `--target bun`: leaves `bun:*` imports external so the bundle runs on Bun
- `--target node`: leaves `node:*` imports external and emits Node-compatible output

You can also call `Bun.build()` from a TypeScript script when you want custom plugins or build-time constants (`define`).

Writing a `Bun.build()` plugin

Bun's plugin API is modeled after esbuild's. Here is a small plugin that lets any file `import note from './note.md'` and converts the Markdown to HTML at bundle time using `Bun.markdown.html()`:

```
const mdPlugin: import('bun').BunPlugin = {
  name: 'markdown-loader',
  setup(build) {
    build.onLoad({ filter: /\.md$/ }, async (args) => {
      const raw = await Bun.file(args.path).text()
      const html = Bun.markdown.html(raw)
      const words = raw.split(/\s+/).filter(Boolean).length
      return {
        exports: { html, words },
        loader: 'object',
      }
    })
  },
}

const result = await Bun.build({
  entrypoints: ['./use-note.ts'],
  outdir: './dist-plugin',
  target: 'bun',
  plugins: [mdPlugin],
  define: {
    __APP_VERSION__: JSON.stringify('1.0.0'),
  },
})

console.log('success:', result.success, 'outputs:', result.outputs.map((o) => o.path.split('/').pop()))
```

When `use-note.ts` imports `./note.md` and runs the built bundle:

```
success: true outputs: [ "use-note.js" ]
words: 9 html: <h1>Build plugins</h1>
<p>Bun plugins run at <strong>bundle time</strong>.</p>
```

The Markdown parsing happened once during `Bun.build()`, and the output bundle contains only the pre-rendered HTML and word count.

Running heavy work off the main thread with Workers

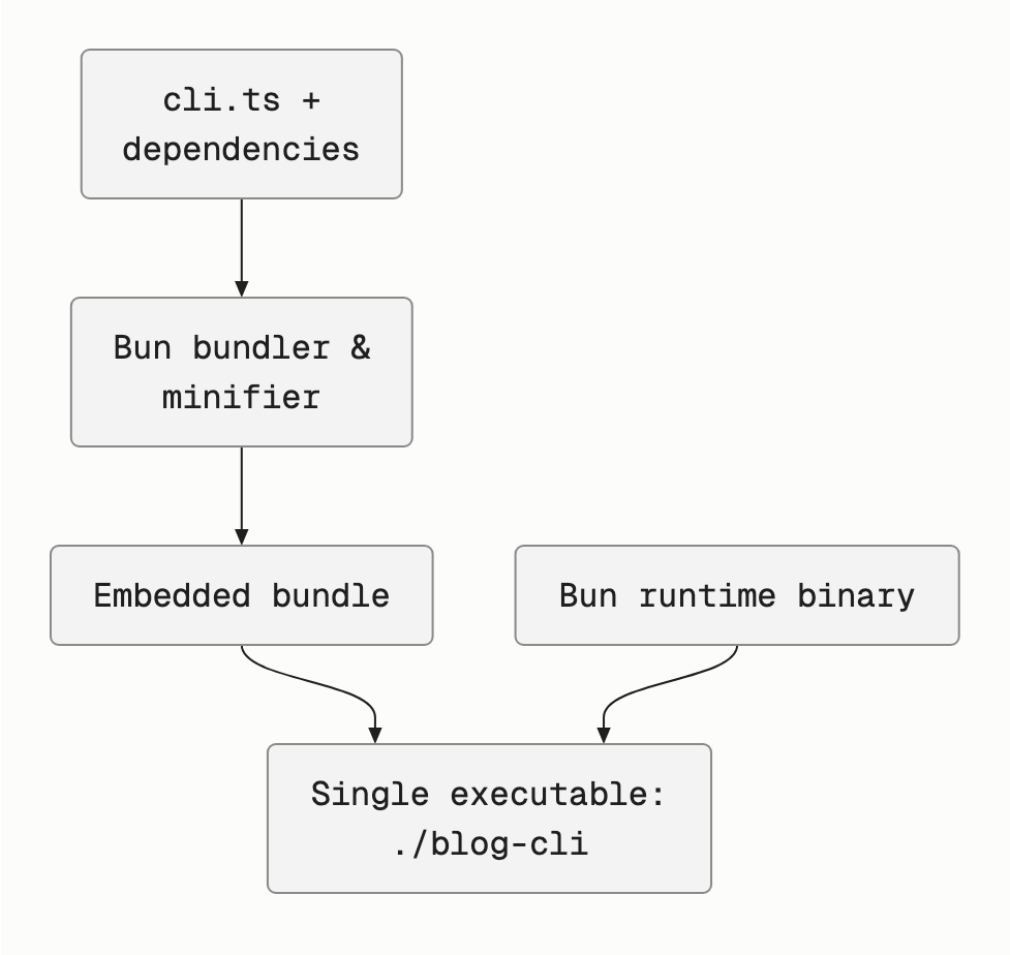
If your server or CLI needs to run CPU-heavy work without blocking incoming HTTP requests, Bun supports standard Web Workers that execute TypeScript files directly:

```
// hash-worker.ts
declare var self: Worker

self.onmessage = (event: MessageEvent<string>) => {
  const digest = new Bun.CryptoHasher('sha256').update(event.data).digest('hex')
  self.postMessage(digest)
}


// main-worker.ts
const worker = new Worker(new URL('./hash-worker.ts', import.meta.url).href)

const digest = await new Promise<string>((resolve) => {
  worker.onmessage = (event) => resolve(event.data)
  worker.postMessage('flaviocopes.com')
})

console.log('worker sha256:', digest)
worker.terminate()


worker sha256: 7a814e81bcec4afc76f703db88359a8b57e6ed61fd90babb1eae78e65a8446f7
```

Always call `worker.terminate()` when a one-off worker finishes, or the open worker thread will keep your script from exiting.

Standalone executables with `bun build --compile`

With `--compile`, Bun bundles your TypeScript and embeds it inside a copy of the Bun runtime, producing a single executable you can copy to another machine that doesn't have Bun or Node.js installed.



Let's compile `cli.ts`:

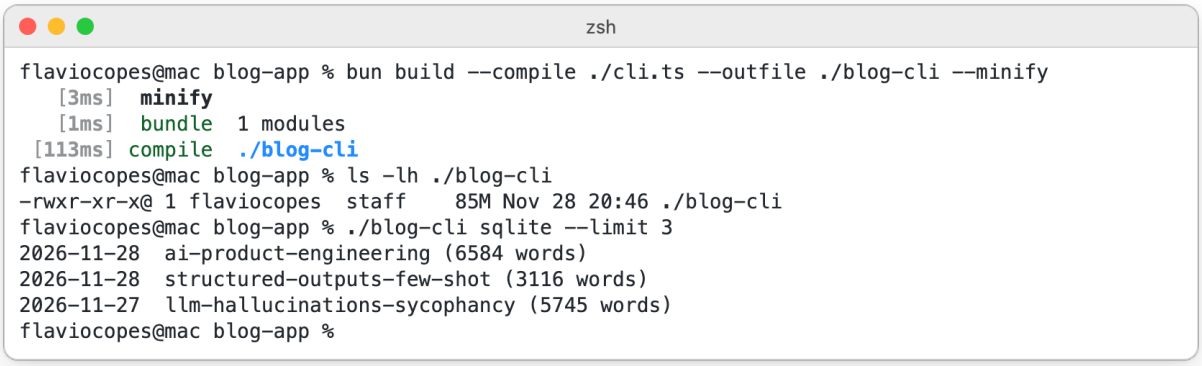
```
bun build --compile ./cli.ts --outfile ./blog-cli --minify
```

```
   [2ms]  minify
   [0ms]  bundle  1 modules
 [127ms] compile  ./blog-cli
```

Check the size and run it:

```
ls -lh ./blog-cli
./blog-cli sqlite --limit 3
```

```
-rwxr-xr-x@ 1 flaviocopes  staff    85M Nov 28 20:41 ./blog-cli
2026-11-28  ai-product-engineering (6584 words)
2026-11-28  structured-outputs-few-shot (3116 words)
2026-11-27  llm-hallucinations-sycophancy (5745 words)
```



Two real measurements worth knowing:

- **Size:** 85 MB, because the entire JavaScriptCore engine and Bun runtime travel inside the file.
- **Speed:** on macOS, the very first launch of a freshly built unsigned binary took 0.91 s while the OS verified it. Every run after that took **0.02 s (20 ms)**, including opening SQLite and searching 2,193 posts.

You can also cross-compile for another OS or architecture from your Mac without needing Docker:

```
bun build --compile --target=bun-linux-x64 ./cli.ts --outfile ./blog-cli-linux
bun build --compile --target=bun-windows-x64 ./cli.ts --outfile ./blog-cli.exe
```

I compared `bun build --compile`, `deno compile`, and `node --build-sea` in How to compile JavaScript into a single executable.

Node.js compatibility and migrating a project

Bun implements most of the Node.js standard library (`node:fs`, `node:path`, `node:http`, `node:crypto`, `node:util`, `node:stream`, `Buffer`, `process`) plus Node-API (N-API) for native addons.

Still, Bun isn't Node.js. They run different JavaScript engines, and edge cases around native addons, V8-specific debugging APIs (`v8`, `inspector`), or subtle stream behaviors can show up in older packages. Check Bun's Node.js compatibility page when a dependency relies on low-level Node internals.

A boring, low-risk migration path

You don't have to switch an existing Node.js repository to Bun in one giant pull request. Move one layer at a time:

1. **Package manager first.** Run `bun install`. Bun reads `package.json`, migrates your existing lockfile into `bun.lock`, and builds a normal `node_modules` folder. Keep running the app with `node` while you verify installs and `bun pm untrusted`.

2. **Scripts next.** Switch `npm run dev` and `npm run build` to `bun run dev` and `bun run build`.
3. **Tests.** Try `bun test` on unit tests that don't rely on Jest-only plugins.
4. **Runtime last.** Run your server or CLI with `bun run src/index.ts`, paying close attention to file I/O, HTTP streaming, child processes, database drivers, and native addons.
5. **CI and production.** Switch CI to `bun ci` and test the production container (`oven/bun:1.4`) in the same Linux environment you deploy to.

If any layer hits a snag, stop there. Using `bun install` and `bun run` on a Node.js production app is already a solid win.

When to choose Bun (and when to stay on Node.js)

Both Bun and Node.js are great tools, and they overlap more every year. Here is how I decide between them today:

Question	Pick Bun	Pick Node.js
New TypeScript API, CLI, or internal tool	Great fit: zero build setup, 6 ms startup	Good too, needs more tooling config
Built-in SQLite, shell scripts, bundling, single executable	All built into one binary	Needs extra packages or experimental flags
Fast CI installs and test runs	`bun ci` + `bun test` are noticeably faster	`pnpm` + Vitest is fast, more moving parts
Heavy legacy Jest or native C++ addon suite	Test carefully first	Safest default
Hosting platform only supports Node.js	Use Bun locally, deploy on Node	Match production runtime everywhere

Deploying Bun with Docker

When deploying a Bun server to a VPS or container platform, use the official `oven/bun` image:

```
FROM oven/bun:1.4-slim
WORKDIR /app
COPY package.json bun.lock ./
RUN bun ci --production
COPY . .
EXPOSE 3000
CMD ["bun", "run", "server.ts"]
```

You now have the whole picture: running TypeScript, managing dependencies and workspaces, working with files, shell scripts, SQLite, HTTP servers, WebSockets, tests, bundles, and standalone binaries, all from one tool.