Bun Course

Flavio Copes

Updated August 7, 2026

Contents

Learn Bun by running TypeScript, managing packages, using runtime APIs, building an HTTP API with SQLite, testing it, and shipping it.

What you'll learn: Build, test, bundle, and ship a small notes API using Bun, Web Standard APIs, and SQLite.

Bun foundations

Install Bun, create a TypeScript project, run code, and use the development workflow.

Understand what Bun includes

Meet Bun as a JavaScript runtime, package manager, test runner, and bundler in one executable.

Bun is a toolkit for JavaScript and TypeScript. It gives you several tools through one `bun` command.

You get:

- a **runtime** for JavaScript and TypeScript
- a **package manager** for npm packages
- a **test runner**
- a **bundler**

The runtime fills the same role as Node.js. You give it a program, and Bun runs that program outside the browser.

Bun uses JavaScriptCore, the JavaScript engine used by Safari. Node.js uses V8 instead. You usually notice the difference through startup speed, compatibility, and the APIs each runtime provides.

Bun still uses the JavaScript ecosystem

Bun does not ask you to leave npm packages behind. It reads `package.json`, creates `node_modules`, and installs packages from the npm registry.

It also implements many Node.js APIs. This means a Node.js project can often run with Bun after a small change:

```
bun run start
```

But compatibility is not a promise that every project works unchanged. A package may depend on a Node.js API Bun does not fully support. Native add-ons and runtime-specific behavior also need careful testing.

My advice is simple: treat Bun as its own runtime. Reuse Node.js packages where they work, but test the application with Bun before shipping it.

What we will build

During this course, we'll build a small notes API. It will use TypeScript, `Bun.serve()`, and SQLite.

We'll also test it, bundle it, and compile it into an executable. This gives us a useful path through every important part of Bun.

Install and update Bun

Install the stable Bun release, verify the executable, and keep it updated without losing track of the version.

Let's install the stable version of Bun.

On macOS or Linux, run the official installer:

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

Open a new terminal after the installer finishes. Then verify the installation:

```
bun --version
```

You should see a version such as `1.3.11`.

You can also install Bun through npm:

```
npm install -g bun
```

On Windows, run the official PowerShell installer:

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

Bun supports Windows 10 version 1809 and newer.

If the command is missing

The installer normally adds Bun to your `PATH`. If the terminal says `bun: command not found`, check whether this directory exists:

```
ls ~/.bun/bin
```

For zsh, add Bun to `~/.zshrc`:

```
export BUN_INSTALL="$HOME/.bun"
export PATH="$BUN_INSTALL/bin:$PATH"
```

Restart the terminal and run `bun --version` again.

Update Bun

Bun can update its own executable:

```
bun upgrade
```

Use the stable release for real projects. Canary releases are useful for testing a fix, but they change on every commit and receive less testing.

Record the version used by your team and deployment environment. When a runtime bug appears, the exact version is one of the first facts you'll need.

Create and run a TypeScript project

Initialize a Bun project, inspect its files, and run TypeScript directly without a separate build step.

Let's create the project we'll use throughout the course.

Create a directory and initialize Bun inside it:

```
mkdir bun-notes
cd bun-notes
bun init --yes
```

`bun init` creates a small TypeScript project. The exact files may change as Bun evolves, but you should see `package.json`, `tsconfig.json`, and `index.ts`.

Replace the contents of `index.ts` with this:

```
const message: string = 'Hello from Bun'
```

```
console.log(message)
```

Run the file:

```
bun index.ts
```

Bun prints:

```
Hello from Bun
```

Bun transpiles TypeScript while it loads the file. We did not run `tsc` first, and we did not create a JavaScript copy.

Notice the word **transpiles**. Bun removes TypeScript syntax so the runtime can execute the program. This does not mean Bun performs a complete type check before every run.

Keep your editor's TypeScript checks enabled. For a separate command-line type check, install TypeScript and run `tsc --noEmit`:

```
bun add --dev typescript
bunx tsc --noEmit
```

Running code and checking types are two different jobs. Keeping that distinction clear prevents a successful `bun index.ts` command from giving us false confidence about the types.

Use watch mode and project scripts

Restart a Bun program when files change and give common project commands stable names in package.json.

During development, we don't want to restart the program after every edit. Bun can watch the files loaded by our application.

Run `index.ts` in watch mode:

```
bun --watch index.ts
```

Change the message and save the file. Bun stops the old process and starts it again.

Notice where `--watch` appears. Bun flags belong immediately after `bun`:

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

If you put the flag at the end, Bun may pass it to the script instead.

Add project scripts

Scripts give the project a shared vocabulary. Open `package.json` and add these entries:

```
{
  "scripts": {
    "dev": "bun --watch index.ts",
    "start": "bun index.ts",
    "typecheck": "tsc --noEmit"
  }
}
```

Now start development with:

```
bun run dev
```

Run the ordinary command with:

```
bun run start
```

And check the TypeScript types with:

```
bun run typecheck
```

You can omit `run` for many scripts, but I prefer `bun run dev`. It makes the intention clear and avoids conflicts with Bun's built-in commands.

Run `bun run` without a script name to list the scripts available in the project.

Check your understanding: Bun foundations

Check Bun's toolchain, installation, TypeScript execution, type checking, watch mode, and project scripts.

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

Take this interactive quiz online

Packages and scripts

Manage dependencies, lock installs, run package binaries, and control install scripts.

Install and lock dependencies

Install a project's npm dependencies with Bun and use bun.lock for repeatable team and deployment installs.

Bun is also a package manager. It reads the same `package.json` file used by npm.

Install the dependencies declared by the project:

```
bun install
```

Bun creates two important things:

- `node_modules`, containing packages used by the project
- `bun.lock`, recording the exact dependency versions Bun resolved

Commit `bun.lock` to Git. Without the lockfile, the version ranges in `package.json` may resolve to newer packages on another machine.

Use the lockfile in automation

During ordinary development, `bun install` may update `bun.lock` when `package.json` changes.

In CI and deployments, use a frozen install:

```
bun install --frozen-lockfile
```

This command fails when `package.json` and `bun.lock` disagree. That failure is useful. It tells you someone changed the dependencies without recording a new resolution.

For a production-only install, skip development dependencies:

```
bun install --frozen-lockfile --production
```

Moving an existing project to Bun

When a project has no `bun.lock`, Bun can migrate `package-lock.json`, `yarn.lock`, or `pnpm-lock.yaml`. It preserves the old lockfile.

Do not delete the old file immediately. First install, run the tests, and verify the application with Bun. Then choose one package manager for the project and remove the extra lockfile in a deliberate change.

Two active lockfiles create ambiguity. A teammate should not have to guess which dependency graph is authoritative.

Add, remove, and update packages

Change project dependencies with Bun while keeping package.json and bun.lock synchronized.

Let's add Zod to validate the JSON sent to our notes API:

```
bun add zod
```

Bun installs the package, adds it to `dependencies`, and updates `bun.lock`.

Development tools belong in `devDependencies`. We already installed TypeScript this way:

```
bun add --dev typescript
```

You can install a specific version when the project requires it:

```
bun add zod@4.1.0
```

Remove a package

Use the package name to remove it:

```
bun remove zod
```

Bun removes the entry from `package.json`, updates the lockfile, and removes the installed package when nothing else needs it.

Add Zod again before continuing:

```
bun add zod
```

Update dependencies

Update packages within the version ranges declared in `package.json`:

```
bun update
```

Update one package with:

```
bun update zod
```

Use `bun update --latest` carefully. It can move packages beyond their current version ranges and introduce breaking changes.

My advice is to update a small group at a time. Read the release notes, run the tests, and commit the matching `package.json` and `bun.lock` changes together.

Run package binaries with bunx

Use bunx to run package command-line tools without turning every temporary tool into a global installation.

Some npm packages provide command-line tools. `bunx` finds and runs those tools.

For example, run Prettier without installing it globally:

```
bunx prettier --check .
```

`bunx` first looks for the executable in the project's installed packages. If it is missing, Bun downloads the package into a shared cache and runs it.

This is similar to `npx`.

Prefer local versions for project tools

A formatter used by a team should have a version recorded by the project:

```
bun add --dev prettier
```

Add a script to `package.json`:

```
{
  "scripts": {
    "format": "prettier --write .",
    "format:check": "prettier --check ."
  }
}
```

Now everyone runs the same installed version:

```
bun run format:check
```

Use `bunx` directly for temporary commands, project generators, or tools you are evaluating. Use a local development dependency for commands that belong to the project's normal workflow.

Choose the runtime deliberately

Many package binaries start with a Node.js shebang. `bunx` respects it and uses Node.js.

You can force Bun with `--bun`:

```
bunx --bun vite --version
```

Do this only when you intend to test the tool under Bun. A command working through ordinary `bunx` does not prove the package itself ran on the Bun runtime.

Trust dependency scripts deliberately

Understand Bun's lifecycle-script policy and approve only the dependency install scripts a project really needs.

An npm package can define lifecycle scripts such as `postinstall`. These scripts run commands on your machine during installation.

That power is useful. Native packages may need to download or build a platform-specific binary. It is also a security boundary.

Bun does not run arbitrary dependency lifecycle scripts by default. If a package needs one, Bun reports that the script was blocked.

List packages with blocked scripts:

```
bun pm untrusted
```

Before trusting one, inspect why the package needs the script. Check its package metadata, source, and documentation.

Then trust the specific package:

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

Bun runs its blocked scripts and records the package in `trustedDependencies` inside `package.json`.

You can also trust a package while adding it:

```
bun add --trust package-name
```

Keep the approval narrow

Avoid this command in a project you have not audited:

```
bun pm trust --all
```

It approves every currently blocked dependency script. That removes the useful review boundary Bun created.

Commit changes to `trustedDependencies`. The approval is part of the project's dependency policy, not a private setting on your laptop.

If a package works without its blocked script, leave it blocked. If it fails, investigate the exact requirement before granting more access.

Check your understanding: packages and scripts

Check dependency installation, bun.lock, package changes, bunx, and Bun's trusted-dependency boundary.

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

Take this interactive quiz online

Runtime APIs

Read configuration, work with files, run child processes, and judge Node.js compatibility.

Read environment variables and arguments

Read configuration from environment files and accept command-line arguments without hard-coding changing values.

Applications need values that change between machines. Ports, database paths, and API tokens do not belong in source code.

Bun automatically reads `.env` files. Create `.env` in the project root:

```
PORT=3000
DATABASE_PATH=notes.sqlite
```

Read those values through `Bun.env`:

```
const port = Number(Bun.env.PORT ?? 3000)
const databasePath = Bun.env.DATABASE_PATH ?? 'notes.sqlite'

console.log({ port, databasePath })
```

`Bun.env` contains strings or `undefined`. Convert numbers and validate required values before using them.

Do not commit secrets in `.env`. Add the file to `.gitignore`, and provide an `.env.example` containing safe placeholder names instead.

Read command-line arguments

Command-line arguments are useful for small tools and maintenance tasks.

Create `greet.ts`:

```
const name = Bun.argv[2] ?? 'friend'

console.log(`Hello ${name}`)
```

Pass the name after the file:

```
bun greet.ts Flavio
```

The output is:

```
Hello Flavio
```

The first two entries in `Bun.argv` identify the Bun executable and the script. Your first argument starts at index `2`.

Environment variables configure how an application runs. Arguments describe what one execution should do. Keeping those roles separate makes commands easier to understand.

Read and write files

Use Bun.file and Bun.write to load and save text or JSON with small Web API-compatible primitives.

`Bun.file()` creates a reference to a file. It does not read the contents immediately.

Create `settings.json`:

```
{
  "siteName": "Bun Notes",
  "itemsPerPage": 20
}
```

Read it with:

```
type Settings = {
  siteName: string
  itemsPerPage: number
```

```
}

const file = Bun.file('settings.json')
const settings = await file.json() as Settings

console.log(settings.siteName)
```

A `BunFile` follows the Web `Blob` interface. You can read it as text, JSON, bytes, an array buffer, or a stream.

Check whether a file exists before reading optional data:

```
const file = Bun.file('settings.json')

if (await file.exists()) {
  console.log(await file.text())
}
```

Write a file

`Bun.write()` accepts a destination and some data:

```
const settings = {
  siteName: 'Bun Notes',
  itemsPerPage: 30,
}

await Bun.write(
  'settings.json',
  JSON.stringify(settings, null, 2),
)
```

The returned promise resolves to the number of bytes written.

You can also copy one file into another:

```
await Bun.write('settings.backup.json', Bun.file('settings.json'))
```

Use `node:fs` for directory operations such as `mkdir()` and `readdir()`. Bun implements those Node.js APIs, while `Bun.file()` and `Bun.write()` cover the common file-content path.

Run child processes

Start another program with Bun.spawn, read its output, and handle its exit code without building a shell command string.

Use `Bun.spawn()` when your program needs to run another executable.

For example, ask Git for its version:

```
const process = Bun.spawn(['git', '--version'])
const output = await process.stdout.text()
const exitCode = await process.exited

console.log(output.trim())
```

```
console.log(`Exit code: ${exitCode}`)
```

`Bun.spawn()` starts the process without blocking the JavaScript event loop. `process.exited` resolves when the child finishes.

Check the exit code before trusting the output:

```
const process = Bun.spawn(['git', 'status', '--short'], {
  stderr: 'pipe',
})

const [exitCode, output, error] = await Promise.all([
  process.exited,
  process.stdout.text(),
  process.stderr.text(),
])

if (exitCode !== 0) {
  throw new Error(error.trim())
}

console.log(output)
```

Pass the command as an array. Each argument stays separate, so spaces inside one value do not become new shell syntax.

Avoid joining user input into a command string. A shell interprets characters such as `;`, `|`, and `$`, which can turn untrusted text into a second command.

For long-running servers and applications, prefer asynchronous `Bun.spawn()`. `Bun.spawnSync()` blocks the current process and fits short command-line tasks where that behavior is intentional.

Use Node.js packages with clear boundaries

Run Node.js APIs and npm packages in Bun while identifying the compatibility assumptions your application depends on.

Bun implements many Node.js built-in modules. Use the explicit `node:` prefix when importing one:

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

const file = join('data', 'notes.json')
console.log(file)
```

This code runs in Bun and Node.js. The prefix also tells the reader that `path` comes from the runtime, not `node_modules`.

Most npm packages install normally:

```
bun add zod
```

Then import them with standard ES module syntax:

```
import { z } from 'zod'

const Note = z.object({
```

```
  title: z.string().min(1),
})

console.log(Note.parse({ title: 'Learn Bun' }))
```

Compatibility needs evidence

A successful installation only proves the package was downloaded. Run the paths your application uses.

Pay extra attention when a dependency uses:

- a native Node.js add-on
- a recently added Node.js API
- Node.js internals rather than public APIs
- process, stream, or module behavior at the edges

Check Bun's Node.js compatibility documentation when something fails. Then create a small reproduction before changing the application.

You can detect Bun at runtime with:

```
if (process.versions.bun) {
  console.log(`Running Bun ${process.versions.bun}`)
}
```

Use runtime checks only when behavior must differ. Shared Web APIs such as `Request`, `Response`, `fetch`, and `URL` usually create a cleaner boundary than many Bun-versus-Node branches.

Check your understanding: runtime APIs

Check environment values, arguments, file APIs, child processes, npm packages, and Node.js compatibility.

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

Take this interactive quiz online

HTTP and SQLite

Build a notes API with Bun.serve, Web Standard requests and responses, and bun:sqlite.

Start an HTTP server

Create a small HTTP server with Bun.serve and return a Web Standard Response from its fetch handler.

`Bun.serve()` starts an HTTP server. Its `fetch` function receives a Web Standard `Request` and returns a `Response`.

Replace `index.ts` with:

```
const server = Bun.serve({
  port: Number(Bun.env.PORT ?? 3000),
  fetch() {
```

```
    return new Response('Bun Notes is running')
  },
})

console.log(`Listening on ${server.url}`)
```

Start the server:

```
bun --watch index.ts
```

Open `http://localhost:3000` in your browser. You can also make the request with curl:

```
curl http://localhost:3000
```

The response is:

```
Bun Notes is running
```

Read the request

The request contains the method, URL, headers, and body sent by the client.

Return a small JSON description of each request:

```
const server = Bun.serve({
  fetch(request) {
    const url = new URL(request.url)

    return Response.json({
      method: request.method,
      path: url.pathname,
    })
  },
})

console.log(`Listening on ${server.url}`)
```

`Request`, `Response`, and `URL` are the same core APIs used by browsers and other modern runtimes. This makes the HTTP boundary familiar and easier to test.

Keep the port configurable. Deployment platforms normally provide it through a `PORT` environment variable.

Route requests and return JSON

Use Bun.serve routes to match HTTP methods and paths, read path parameters, and return useful JSON responses.

Bun can match routes before calling the fallback `fetch` handler.

Let's create two read-only routes:

```
type Note = {
  id: number
  title: string
}
```

```
const notes: Note[] = [
  { id: 1, title: 'Learn Bun' },
  { id: 2, title: 'Build the notes API' },
]

const server = Bun.serve({
  routes: {
    '/api/notes': {
      GET: () => Response.json(notes),
    },
    '/api/notes/:id': request => {
      const id = Number(request.params.id)
      const note = notes.find(note => note.id === id)

      if (!note) {
        return Response.json(
          { error: 'Note not found' },
          { status: 404 },
        )
      }

      return Response.json(note)
    },
  },
  fetch() {
    return Response.json(
      { error: 'Route not found' },
      { status: 404 },
    )
  },
})

console.log(`Listening on ${server.url}`)
```

The `:id` segment creates `request.params.id`. It is a string, so we convert it before comparing it with numeric IDs.

Try the collection route:

```
curl http://localhost:3000/api/notes
```

Then request one note:

```
curl http://localhost:3000/api/notes/1
```

An unknown ID returns JSON with status `404`. An unmatched path reaches `fetch()` and returns a different `404` response.

HTTP status codes are part of the API contract. Do not return `200` for every outcome and ask clients to inspect an error string.

Accept and validate JSON

Read a JSON request body, validate unknown input with Zod, and return a clear creation or client-error response.

TypeScript types do not validate data sent over HTTP. A client can send any JSON value.

Define the input with Zod:

```
import { z } from 'zod'

const NoteInput = z.object({
  title: z.string().trim().min(1).max(120),
})
```

Add a `POST` handler beside the existing `GET` handler:

```
let nextId = 3

const notesRoute = {
  GET: () => Response.json(notes),
  POST: async (request: Request) => {
    const json = await request.json().catch(() => null)
    const result = NoteInput.safeParse(json)

    if (!result.success) {
      return Response.json(
        { error: 'Send a title between 1 and 120 characters' },
        { status: 400 },
      )
    }

    const note = {
      id: nextId++,
      title: result.data.title,
    }

    notes.push(note)

    return Response.json(note, { status: 201 })
  },
}
```

Use it in the server:

```
Bun.serve({
  routes: {
    '/api/notes': notesRoute,
  },
})
```

Create a note with curl:

```
curl -X POST http://localhost:3000/api/notes \
```

```
  -H 'Content-Type: application/json' \
  -d '{"title":"Test the API"}'
```

A valid request returns the new note with status `201 Created`. Invalid JSON and invalid titles return `400 Bad Request`.

Validate at the boundary. After `safeParse()` succeeds, the rest of the handler can use a known input shape.

Store data with SQLite

Persist notes with Bun's built-in SQLite driver and reuse prepared statements with bound values.

Our notes currently disappear when the process restarts. Bun includes a SQLite driver, so we can persist them without installing another package.

Create `database.ts`:

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

export type Note = {
  id: number
  title: string
}

const databasePath = Bun.env.DATABASE_PATH ?? 'notes.sqlite'
const db = new Database(databasePath, { create: true })

db.run(`
  CREATE TABLE IF NOT EXISTS notes (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    title TEXT NOT NULL
  )
`)

const listQuery = db.query<Note, []>(`
  SELECT id, title
  FROM notes
  ORDER BY id DESC
`)

const createQuery = db.query<Note, [string]>(`
  INSERT INTO notes (title)
  VALUES (?)
  RETURNING id, title
`)

export function listNotes() {
  return listQuery.all()
}

export function createNote(title: string) {
```

```
  return createQuery.get(title)!
}
```

`db.query()` prepares and caches each SQL statement. The `?` placeholder keeps the title separate from the SQL syntax.

Now import the functions in `index.ts`:

```
import { createNote, listNotes } from './database'
```

Use them in the notes route:

```
const notesRoute = {
  GET: () => Response.json(listNotes()),
  POST: async (request: Request) => {
    const json = await request.json().catch(() => null)
    const result = NoteInput.safeParse(json)

    if (!result.success) {
      return Response.json(
        { error: 'Send a title between 1 and 120 characters' },
        { status: 400 },
      )
    }

    return Response.json(
      createNote(result.data.title),
      { status: 201 },
    )
  },
}
```

Restart the server, create a note, and restart again. The note remains in `notes.sqlite`.

Add `notes.sqlite` to `.gitignore`. Runtime data does not belong in the source repository.

Check your understanding: HTTP and SQLite

Check Bun.serve, Web Standard HTTP objects, routing, validation, status codes, and SQLite prepared statements.

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

Take this interactive quiz online

Test, build, and ship

Test the application, create production output, compile an executable, and review the release.

Write tests with bun:test

Create fast TypeScript tests with Bun's built-in test runner and use focused assertions to describe behavior.

Bun includes a test runner. You do not need to install Jest or another test package to start.

Create `title.ts`:

```
export function normalizeTitle(title: string) {
  return title.trim().replaceAll(/\s+/g, ' ')
}
```

Create `title.test.ts` beside it:

```
import { expect, test } from 'bun:test'
import { normalizeTitle } from './title'

test('normalizes whitespace in a note title', () => {
  expect(normalizeTitle('  Learn   Bun  ')).toBe('Learn Bun')
})
```

Run every discovered test:

```
bun test
```

Bun finds files with names such as `.test.ts`, `_test.ts`, `.spec.ts`, and `_spec.ts`.

The test has three small parts:

1. provide an input
2. call the function
3. compare the result with the expected value

Add an edge case:

```
test('keeps a title that is already clean', () => {
  expect(normalizeTitle('Build the API')).toBe('Build the API')
})
```

Run tests whenever files change:

```
bun test --watch
```

You can also generate a coverage report:

```
bun test --coverage
```

Coverage tells you which code executed. It does not tell you whether the assertions describe the right behavior. Prefer a few meaningful cases over many tests that only repeat the implementation.

Test an HTTP server

Start a Bun server on a temporary port, make a real HTTP request, and stop it reliably after the test.

An HTTP test should check the response a client receives. Bun can start a server on an available port by using port `0`.

Move server creation into `server.ts`:

```
export function createServer(port = 0) {
  return Bun.serve({
    port,
```

```
    routes: {
      '/health': Response.json({ ok: true }),
    },
    fetch() {
      return new Response('Not found', { status: 404 })
    },
  })
}
```

Use it from `index.ts`:

```
import { createServer } from './server'

const port = Number(Bun.env.PORT ?? 3000)
const server = createServer(port)

console.log(`Listening on ${server.url}`)
```

Now create `server.test.ts`:

```
import { expect, test } from 'bun:test'
import { createServer } from './server'

test('reports that the server is healthy', async () => {
  const server = createServer()

  try {
    const url = new URL('/health', server.url)
    const response = await fetch(url)

    expect(response.status).toBe(200)
    expect(await response.json()).toEqual({ ok: true })
  } finally {
    server.stop(true)
  }
})
```

The test uses the real HTTP stack. It does not assume a fixed port, so it can run beside another development server.

The `finally` block matters. Bun stops the server even when an assertion fails. Without cleanup, the test process can keep listening or affect the next test.

Use this pattern for a few important request paths. Keep validation and database logic testable as smaller functions too, so every behavior does not require a network request.

Bundle server code

Create a production Bun bundle, choose the Bun target, keep source maps, and understand what bundling does not verify.

Bun can run the TypeScript source directly. Bundling is optional for a Bun server, but it can reduce the number of files you deploy.

Create one Bun-targeted bundle:

```
bun build \
  --target=bun \
  --outdir=dist \
  --sourcemap=external \
  index.ts
```

Run the result with:

```
bun dist/index.js
```

`--target=bun` preserves Bun runtime APIs such as `Bun.serve()` and `bun:sqlite`. A browser target would be wrong for server code.

The source map helps error traces point back to the original TypeScript. Keep it with the deployment when your logging system can use it.

Know what the bundle contains

The bundler follows imports from `index.ts` and combines application code and package code. Runtime files remain separate unless you explicitly embed them.

Our `notes.sqlite` database should stay outside the bundle. It is changing application data, not source code.

The bundler transpiles TypeScript, but it does not replace a complete type check. Run the full verification sequence:

```
bun run typecheck
bun test
bun build --target=bun --outdir=dist index.ts
```

Then start the bundle and request `/health`. A successful build only proves Bun produced output. A startup check proves the output can run in the expected environment.

Compile and ship the application

Compile the Bun application into one executable and review configuration, data, platform, and recovery before release.

Bun can combine your application and the Bun runtime into one executable.

Compile the notes API:

```
bun build \
  --compile \
  --outfile=notes-api \
  index.ts
```

Run it without the `bun` command:

```
./notes-api
```

The executable includes the runtime and bundled source. The target machine does not need a separate Bun installation.

Build for the target platform

An executable is platform-specific. A macOS ARM64 binary does not run as a Linux x64 binary.

The safest path is to build in the same operating system and architecture used for deployment. Bun also supports explicit cross-compilation targets when you need them.

Do not embed production secrets into the executable. Supply configuration through protected environment variables at runtime.

Keep `notes.sqlite` on durable storage. Replacing the executable during a deployment must not replace the database.

Review the release

Before shipping, run:

```
bun run typecheck
bun test
bun build --compile --outfile=notes-api index.ts
```

Then verify:

- the executable starts with production configuration
- `/health` returns `200`
- the notes database is writable and backed up
- logs do not contain secrets or complete request bodies
- the previous executable is available for rollback

My advice is to keep the first deployment boring. Use one known Bun version, one tested artifact, explicit configuration, and a recovery path.

You now have the complete Bun workflow: run TypeScript, manage packages, use Bun APIs, build an HTTP service, persist data, test it, and ship it.

Check your understanding: test, build, and ship

Check bun:test, HTTP cleanup, Bun-targeted bundles, compiled executables, configuration, data, and release verification.

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

Take this interactive quiz online

The interactive version of this course is available at https://flaviocopes.com/courses/bun/.