readme.md for @git.zone/tsdeno

A smart wrapper around `deno compile` that isolates `package.json` during compilation — preventing devDependencies from inflating your binary by hundreds of megabytes.

Issue Reporting and Security

For reporting bugs, issues, or security vulnerabilities, please visit community.foss.global/. This is the central community hub for all issue reporting. Developers who sign and comply with our contribution agreement and go through identification can also get a code.foss.global/ account to submit Pull Requests directly.

Install

```
# Global install (recommended for CLI usage)
npm install -g @git.zone/tsdeno

# Or as a project dev dependency
pnpm install --save-dev @git.zone/tsdeno
```

□□ The Problem

When you run `deno compile` in a project that has a `package.json`, Deno resolves **every** dependency listed — including `devDependencies`. It does **not** distinguish between `dependencies` and `devDependencies`. This means build tools like:

- □□ `rspack` (~110 MB of native binaries)
- □□ `rolldown` (~40 MB of native binaries)
- □□ `esbuild` (~11 MB)
- □□ `typescript` (~23 MB)

- □□ `tswatch`, `tsbundle`, and their entire transitive trees

...all get bundled into your compiled executable, even though they're **never imported at runtime**.

A real-world example: a Deno server binary went from **596 MB** to **1022 MB** — with 426 MB of pure dead weight from dev-only build tools.

The root cause: Deno reads `package.json` and resolves the full dependency graph into its global cache, then embeds everything when compiling. There is no `--omit=dev` flag, no config to skip devDependencies, and `--node-modules-dir=none` alone doesn't help if `package.json` is present.

□ The Solution

`tsdeno compile` wraps `deno compile` with a simple but effective strategy:

1. **Temporarily renames** `package.json` → `package.json.bak`
2. **Adds** `--node-modules-dir=none` automatically (uses Deno's global cache instead of local `node_modules`)
3. **Runs** `deno compile` with all your arguments passed through
4. **Restores** `package.json` — guaranteed, even if compilation fails (try/finally)

With `package.json` hidden, Deno only resolves dependencies declared in `deno.json` via `npm:` specifiers — which are your actual runtime dependencies.

Usage

CLI — Passthrough Mode

Drop-in replacement — just swap `deno compile` for `tsdeno compile`:

```
# Before (bloated binary):
deno compile --allow-all --no-check --output myapp mod.ts

# After (lean binary):
tsdeno compile --allow-all --no-check --output myapp mod.ts
```

All `deno compile` flags are passed through untouched. Cross-compilation works the same way:

```
tsdeno compile --allow-all --no-check \
  --output dist/myapp-linux-x64 \
  --target x86_64-unknown-linux-gnu \
  mod.ts

tsdeno compile --allow-all --no-check \
  --output dist/myapp-macos-arm64 \
  --target aarch64-apple-darwin \
  mod.ts
```

CLI — Config Mode (.smartconfig.json)

For projects with multiple compile targets, define them in `.smartconfig.json` instead of writing long CLI commands. Just run `tsdeno compile` with no arguments:

```
tsdeno compile
```

tsdeno reads compile targets from the `@git.zone/tsdeno` key in your `.smartconfig.json`:

```
{
  "@git.zone/tsdeno": {
    "compileTargets": [
      {
        "name": "myapp-linux-x64",
        "entryPoint": "mod.ts",
        "outDir": "./dist",
        "target": "x86_64-unknown-linux-gnu",
        "permissions": ["--allow-all"],
        "noCheck": true
      },
      {
        "name": "myapp-macos-arm64",
        "entryPoint": "mod.ts",
        "outDir": "./dist",
        "target": "aarch64-apple-darwin",
        "permissions": ["--allow-all"],
        "noCheck": true
      }
    ]
```

```
  }
}
```

Each compile target supports these fields:

Field	Type	Required	Description
`name`	`string`	✓	Output binary name (combined with `outDir` for path)
`entryPoint`	`string`	✓	Path to the entry TypeScript file
`outDir`	`string`	✓	Directory for the compiled output
`target`	`string`	✓	Deno compile target triple (e.g. `x86_64-unknown-linux-gnu`)
`permissions`	`string[]`	✓	Deno permission flags (e.g. `["--allow-all"]`)
`noCheck`	`boolean`	✓	Skip type checking (`--no-check`)

In config mode, `package.json` is hidden **once** for the entire batch — all targets compile in sequence with a single hide/restore cycle.

Programmatic API

You can also use `tsdeno` as a library in your build scripts:

```
import { TsDeno } from '@git.zone/tsdeno';

const tsDeno = new TsDeno(); // uses process.cwd()
// or: new TsDeno('/path/to/project')

// Passthrough mode — pass args directly
await tsDeno.compile([
  '--allow-all',
  '--no-check',
  '--output', 'dist/myapp',
  '--target', 'x86_64-unknown-linux-gnu',
  'mod.ts',
]);
```

```
// Config mode — reads compile targets from .smartconfig.json
await tsDeno.compileFromConfig();
```

The `TsDeno` class handles the full package.json isolation lifecycle automatically.

CI/CD Integration

Example Gitea/GitHub Actions workflow:

```
steps:
  - name: Set up Deno
    uses: denoland/setup-deno@v1
    with:
      deno-version: v2.x

  - name: Set up Node.js
    uses: actions/setup-node@v4
    with:
      node-version: '22'

  - name: Install tsdeno
    run: npm install -g @git.zone/tsdeno

  - name: Compile binary
    run: tsdeno compile --allow-all --no-check --output myapp mod.ts
```

□□ How It Works — Deep Dive

Why `package.json` Causes Bloat

Deno projects often have both `deno.json` (with `npm:` import specifiers for runtime deps) **and** a `package.json` (for npm publishing, scripts like `tswatch`/`tsbundle`, etc.). When `deno compile` runs:

1. Deno discovers `package.json` and resolves **all** listed packages (deps + devDeps)
2. These get cached in Deno's global npm cache
3. `deno compile` embeds everything it resolved — the full transitive closure
4. Your binary now contains build tools, linters, test frameworks, etc.

What tsdeno Does Differently

By hiding `package.json` during compilation, Deno falls back to resolving **only** the `npm:` specifiers in your `deno.json`. These are your intentionally declared runtime dependencies. Combined with `--node-modules-dir=none` (which prevents Deno from creating/reading a local `node_modules`), the result is a clean binary with only what you actually import.

Safety Guarantees

- **Atomic restore**: `package.json` is restored in a `finally` block — it will be put back even if `deno compile` crashes
- **No-op when absent**: If there's no `package.json`, tsdeno runs `deno compile` normally
- **Exit code passthrough**: If `deno compile` fails, tsdeno exits with the same code
- **Transparent**: All output from `deno compile` is streamed through to your terminal

License and Legal Information

Trademarks

Company Information

Task Venture Capital GmbH Registered at District Court Bremen HRB 35230 HB, Germany

For any legal inquiries or further information, please contact us via email at hello@task.vc.

By using this repository, you acknowledge that you have read this section, agree to comply with its terms, and understand that the licensing of the code does not imply endorsement by Task Venture Capital GmbH of any derivative works.

Revision #5
Created 2026-03-28 10:50:07 UTC by foss.global Team
Updated 2026-03-28 12:15:36 UTC by foss.global Team