bun-starter

minimal web development environment

A web page boils down to nothing more than a file stored inside a directory on a physical machine. A **browser** resolves the **domain name** to an **IP address** via `DNS` and connects on a **port** (`443` for `HTTPS`, `80` for `HTTP`).

A port is a numbered gateway - software binds to it on the inside to offer a service, while the outside world connects to it to access that service. **Think of a Minecraft server**: players connect to a world hosted on a machine, served by one computer or many working as a network.

Software on the website's host machine listens for requests (from an internet browser) and sends or 'serves' the requested files back - hence '**server**'. Not to be confused with the hardware hosting it, which is sometimes also referred to as a server or `VPS` (Virtual Private Server).

`HTTP` governs this exchange at **OSI** Layer 7 (`L7` Application). `HTTPS` layers `TLS` encryption beneath it at Layers 5-6 (`L5` Session/`L6` Presentation), which wraps the `HTTP` content and hands it down to `TCP` at Layer 4 (`L4` Transport).

To simplify, `HTTPS` is just `HTTP` (`L7`) over `TLS` (`L5/6`) over `TCP` (`L4`).

Request, serve, respond: these are the `primitives` of this domain - indivisible operations that everything else builds upon.

Structure

Template Files

```
flake.nix       environment definition
serve.ts        HTTP server
index.html      your markup
style.css       your styles
.envrc          direnv activation (commented)
.gitignore      ignored paths
README.md       this document
```

Generated and Committed

```
flake.lock      pinned Nix dependency versions
package.json    dependency declarations (when you add dependencies)
bun.lockb       pinned dependency versions (when you add dependencies)
```

Generated and Ignored

```
node_modules/   installed dependencies
.direnv/        direnv cache
server          compiled binary
```

Files

flake.nix declares what the environment provides (currently just Bun). Add databases, native libraries, or other tools here as needed.

flake.lock pins exact versions. Commit it for reproducibility, run `nix flake update` to get latest versions.

serve.ts:

```
// values the server needs
const port = 3000
const staticDir = "."

// how the server identifies what it sends
const contentTypes = {
  ".html": "text/html",
  ".css": "text/css",
  ...
}

// reads bytes from disc, returns response
function serveFile(path) { ... }

// returns 404 response
function notFound() { ... }

// prints request info to terminal
function log(request, status) { ... }

// brings everything to life
Bun.serve({ port, fetch })
```

index.html and **style.css** are placeholders. Replace them.

.envrc enables automatic shell activation via direnv. Uncomment and run `direnv allow` to use it.

Workflow

```
nix develop            # enter environment
bun --hot serve.ts     # run dev server at localhost:3000

                       # build production binary
bun build --compile serve.ts --outfile=server
```

The `-hot` flag reloads on file changes. The compiled binary is self-contained and runs without Bun installed.

Concepts

Request-Response

The browser sends an HTTP request (`GET /index.html`), the server reads the file and sends it back with a `Content-Type` header (`text/html`, `text/css`, `image/png`) so the browser knows how to handle it.

Static vs Dynamic

Static responses serve files from disc unchanged. Dynamic responses are computed - an API endpoint runs code and returns data that never existed as a file.

Adding a dynamic endpoint:

```
if (path === "/api/weather") {
    return Response.json({ temp: 18, conditions: "overcast" })
}
```

Proxy

Your binary listens on port `3000`. A proxy (nginx, caddy) sits in front, handles HTTPS on port `443`, and forwards requests to your binary. This separates network concerns from application logic.

Environment Isolation

`nix develop` creates an isolated shell with exactly what the flake specifies. Nothing is installed globally. Clone the repo anywhere, run `nix develop`, get an identical environment.

Lock Files

`flake.lock` records exactly which package versions were resolved. Commit it to freeze versions. Delete and regenerate to update.

Extension

API routes: add conditions in `serve.ts` matching paths to responses.

Dependencies: `bun add <package>`, then commit `package.json` and `bun.lockb`.

System tools: add to the packages list in `flake.nix`.

Database: add to flake for local dev, configure production separately via environment variables.

Containers: Nix can build OCI images containing just your compiled binary.