JAVASCRIPT (SERVER) RUNTIMES

HI 👋

Sebastian Springer
Fullstack JavaScript Developer
Munich, Germany

AGENDA

- Node
- Deno
- Bun
- **Who's the best?**

COMPARISON

Criteria	Node.js	Deno	Bun
Release	2009	2020	2022
Engine	V8	V8	JavaScriptCore
Language (impl.)	C, C++, JS	Rust, TS, JS	Zig, C++
Package Manager	npm	none (URL)	bun (npm-kompat.)
TS Support	via Transpiler	native	native
Security	Permission System	Permission System	No Isolation
Ecosystem	Very big	Medium	Small, growing
Philosophy	Modular, stable	Secure, modern	fast, all-in-1

node

JS®

ABOUT NODE.JS

https://nodejs.org

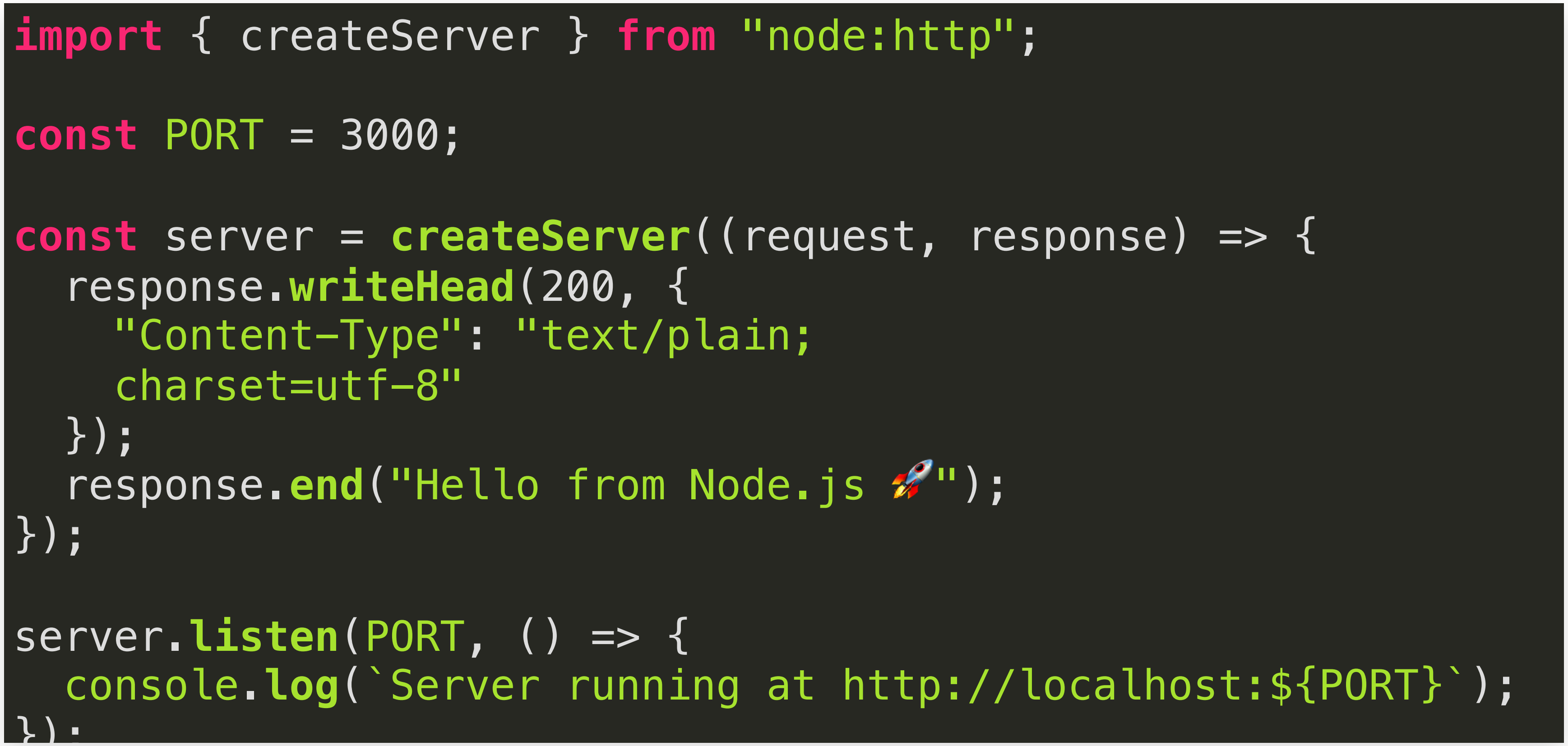
Node.js is the veteran runtime that brought JavaScript from the browser to the server, carrying a few scars from its early design battles but still running half the internet.

FEATURES

- Lots of low level built-in modules
- Multiple package managers (npm, yarn, pnpm)
- Huge ecosystem (npm)
- Supports CommonJS and ES Modules
- Starts to think about TypeScript

INSTALLATION

- Installer package for multiple Platforms
- Multiple package managers supported (Homebrew, Chocolatey, etc.)
- Version managers (nvm, asdf, fnm, etc.)
- Container (e.g. Docker)



```javascript
import { createServer } from "node:http";

const PORT = 3000;

const server = createServer((request, response) => {
  response.writeHead(200, {
    "Content-Type": "text/plain;
    charset=utf-8"
  });
  response.end("Hello from Node.js 🚀");
});

server.listen(PORT, () => {
  console.log(`Server running at http://localhost:${PORT}`);
});
```

RUN

- Save code to `server.js`
- Run with `node server.js`

IMPLEMENTATION

- ES Modules - autodetection
- Node.js core modules
- Single threaded, single process by default
- Executed via node on the command line

PACKAGE MANAGEMENT

- npm is the default package manager
- Initialization: `npm init -y`
- Installation: `npm install <package>`
- Package content is stored in local `node_modules` folder

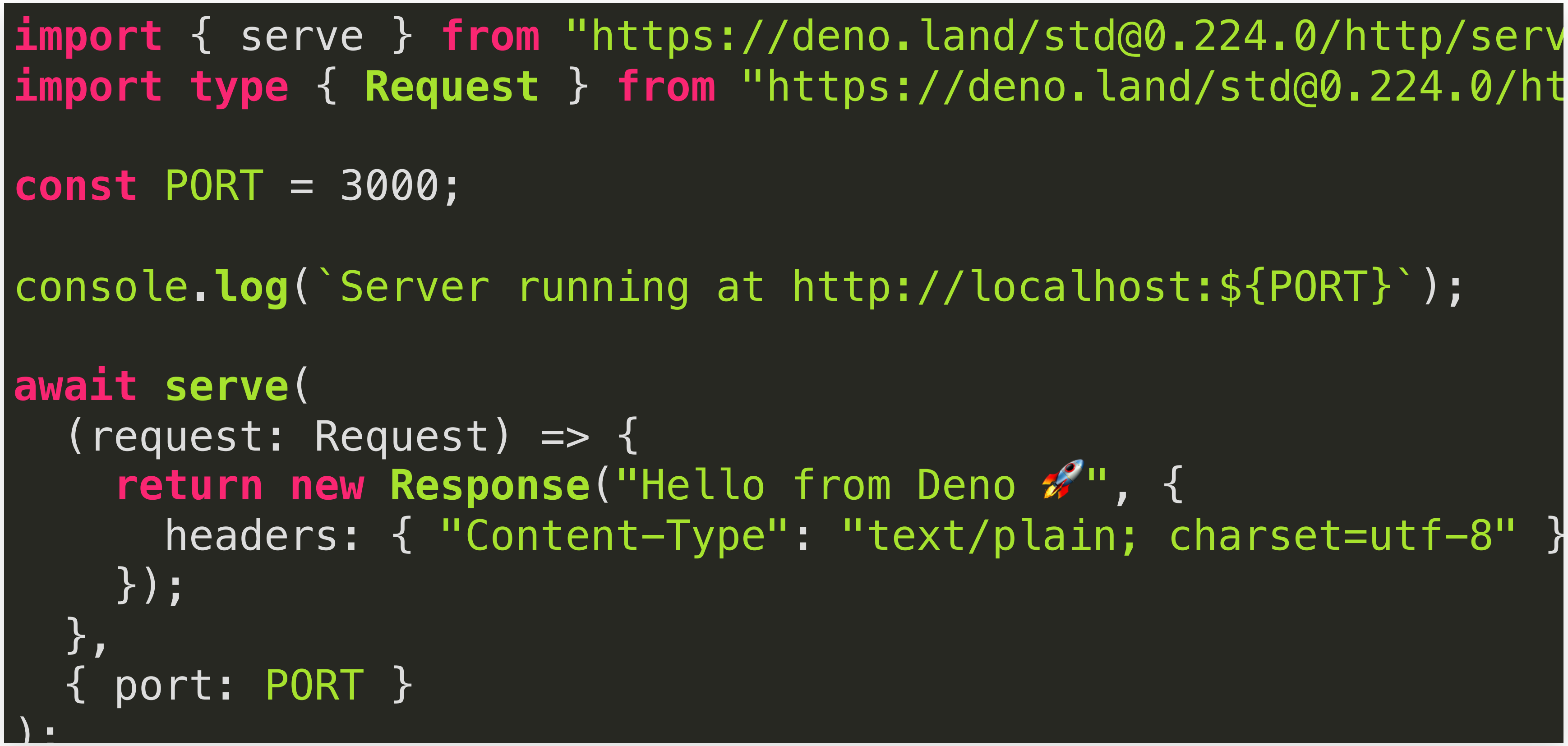
ABOUT DENO

https://deno.com/

Deno is the clean-slate reimagining of a JavaScript runtime, promising modern standards, built-in security, and no mysterious node_modules lurking in the shadows.

INSTALLATION

- Install Script
- npm
- Multiple package managers supported (Homebrew, Chocolatey, etc.)
- Version managers (asdf, etc.)
- Container (e.g. Docker)



```
import { serve } from "https://deno.land/std@0.224.0/http/serv
import type { Request } from "https://deno.land/std@0.224.0/ht

const PORT = 3000;

console.log(`Server running at http://localhost:${PORT}`);

await serve(
  (request: Request) => {
    return new Response("Hello from Deno 🚀", {
      headers: { "Content-Type": "text/plain; charset=utf-8" }
    });
  },
  { port: PORT }
);
```

RUN

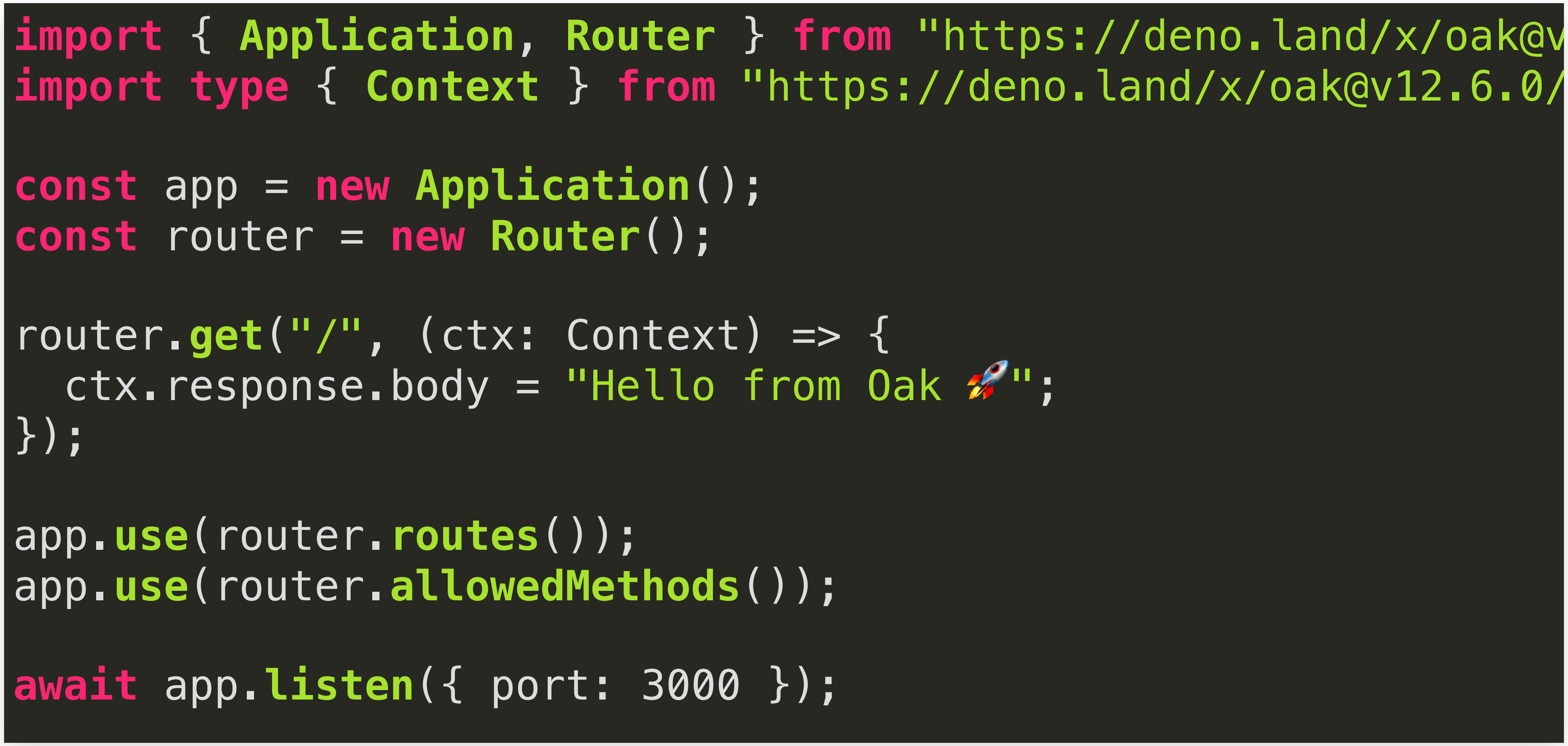
- Save code to `server.ts`
- Run with `deno run --allow-net server.ts`

IMPLEMENTATION

- JS and TS autodetection (based on file extension)
- deno object and the standard library
- Single threaded, single process by default
- Executed via `deno run` on the command line

PACKAGE MANAGEMENT

- Deno has its own package management system
- There is no `package.json` and no explicit installation step
- Modules are imported via URLs; Deno fetches them on demand
- Fetched modules are stored in a global cache
- You can create a lock file (`--lock`) and use a resolver file to pin versions



```
import { Application, Router } from "https://deno.land/x/oak@v
import type { Context } from "https://deno.land/x/oak@v12.6.0/

const app = new Application();
const router = new Router();

router.get("/", (ctx: Context) => {
  ctx.response.body = "Hello from Oak 🚀";
});

app.use(router.routes());
app.use(router.allowedMethods());

await app.listen({ port: 3000 });
```

DENO CLI COMMANDS

- `deno run [file]` - Run a script (JS/TS) with optional permissions (`--allow-net`, `--allow-read`)
- `deno cache [file]` - Pre-cache remote dependencies
- `deno fmt [file]` - Format code automatically
- `deno lint [file]` - Lint code and detect issues
- `deno test` - Run tests defined with `Deno.test()`
- `deno bundle [file] [out.js]` - Bundle code into a single JavaScript file
- `deno info [file]` - Show dependency graph, cache info, and file metadata
- `deno upgrade` - Upgrade Deno to the latest version
- `deno eval "console.log('Hello')"` - Quick one-liner execution
- `deno doc [file]` - Generate documentation for modules

DENO PERMISSION SYSTEM

- **Secure by default:** Deno scripts run in a sandbox with no access to the file system, network, environment, or subprocesses unless explicitly allowed
- `--allow-net` - Allow network access
- `--allow-read` - Allow reading files from disk
- `--allow-write` - Allow writing files to disk
- `--allow-env` - Allow access to environment variables
- `--allow-run` - Allow running subprocesses
- `--allow-hrtime` - Allow high-resolution timers
- Permissions can be combined or restricted to specific hosts/paths:

```
deno run --allow-net=example.com --allow-read=./data server.ts
```

- Runtime checks ensure that scripts can't do more than what's explicitly allowed

DENO STANDARD LIBRARY (STD)

- Official collection of **stable, audited modules** maintained by the Deno team
- No installation required — import directly via URL:

```
import { serve } from "https://deno.land/std@0.201.0/http/server.ts";
```

- Provides utilities for:
  - HTTP servers (`http`)
  - File system operations (`fs`)
  - Path manipulation (`path`)
  - Dates and times (`datetime`)
  - Streams, buffers, and more
- Versioned per module — you can pin a specific version for stability
- Well-tested, fully TypeScript-ready, and integrates seamlessly with Deno CLI

NODE COMPATIBILITY

- You can use Node.js core modules directly with the `node:` prefix.
- Deno provides a compatibility layer for Node.js APIs: https://docs.deno.com/api/node/
- This allows you to run many Node.js modules and code directly in Deno.
- NPM packages can be used with the `npm:` prefix.

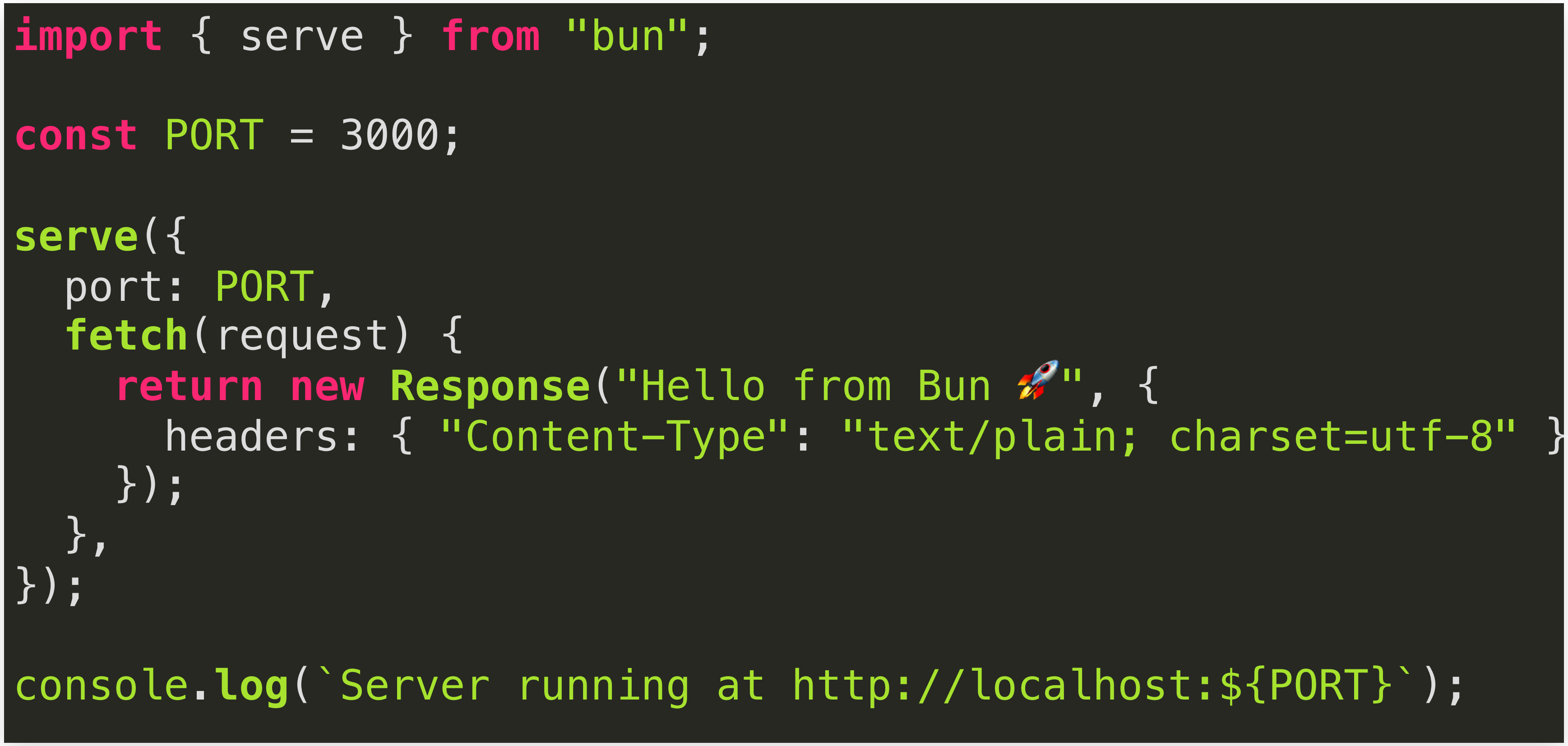
ABOUT BUN

https://bun.com/

Bun is the speed-obsessed new kid on the block, baking JavaScript, TypeScript, and even bundling into one ultra-fast runtime that's still figuring out how to grow up.

INSTALLATION

- Install Script
- Container (e.g. Docker)



```
import { serve } from "bun";

const PORT = 3000;

serve({
  port: PORT,
  fetch(request) {
    return new Response("Hello from Bun 🚀", {
      headers: { "Content-Type": "text/plain; charset=utf-8" }
    });
  },
});

console.log(`Server running at http://localhost:${PORT}`);
```

RUN

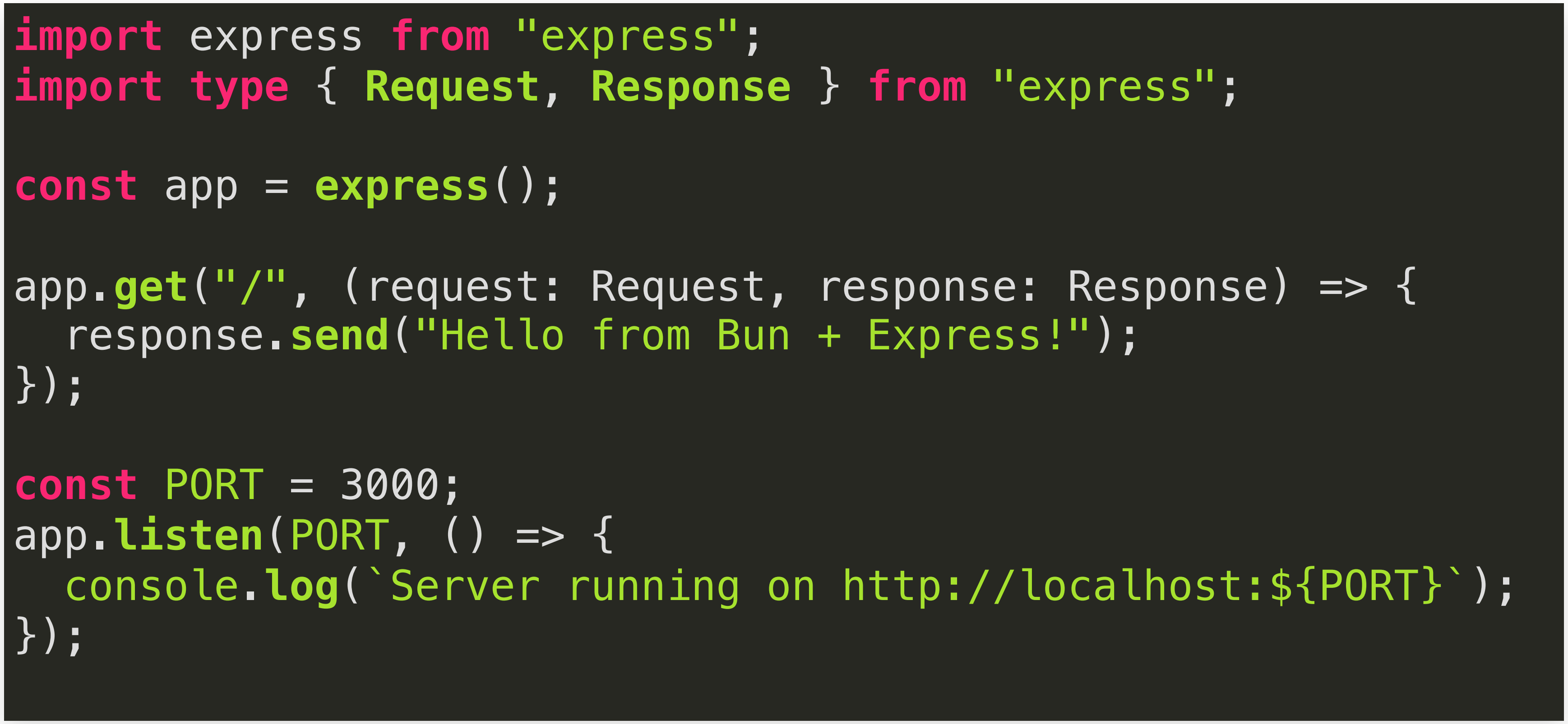
- Save code to `server.js`
- Run with `bun run server.js`

IMPLEMENTATION

- JS and TS autodetection (based on file extension)
- bun object
- Single threaded, single process by default
- Executed via `bun run` on the command line

PACKAGE MANAGEMENT

- Bun is fully npm-compatible and can use the entire npm registry
- Initialization via `bun init -y`
- Supports `package.json` and automatic installation of dependencies
- Packages are stored locally in a `node_modules` directory
- Lockfile support (`bun.lock`) ensures reproducible builds



```typescript
import express from "express";
import type { Request, Response } from "express";

const app = express();

app.get("/", (request: Request, response: Response) => {
  response.send("Hello from Bun + Express!");
});

const PORT = 3000;
app.listen(PORT, () => {
  console.log(`Server running on http://localhost:${PORT}`);
});
```

BUN CLI COMMANDS

- `bun run [file]` - Run a JavaScript/TypeScript file
- `bun install` - Install npm dependencies from `package.json`
- `bun add [package]` - Add a package to `package.json` and install it
- `bun remove [package]` - Remove a package from `package.json`
- `bun build [file]` - Bundle code for production
- `bun test` - Run tests with Bun's native test runner
- `bun create [template]` - Scaffold a new project from templates
- `bun dev [file]` - Run a file with hot reload for development
- `bun upgrade` - Upgrade Bun to the latest version
- `bun --help` - List all available commands and options

NODE COMPATIBILITY

- Bun aims to be largely compatible with Node.js, supporting many core modules and npm packages.
- You can use most Node.js libraries directly in Bun without modification.
- Some Node.js APIs may not be fully supported or behave differently, so testing is recommended.
- npm packages can be used mostly without any issues.

CONCLUSION: NODE.JS VS DENO VS BUN

- **Node.js:** Battle-tested, huge ecosystem, continuously (but slowly) evolving
- **Deno:** Modern, secure by default, ES Modules & TypeScript first, URL imports, built-in tooling
- **Bun:** Ultra-fast, native npm support, built-in bundler & transpiler, still evolving
- Each runtime has trade-offs — understanding **strengths, weaknesses, and use cases** is key

QUESTIONS?

Thank you for your attention!
Any questions?