Web Applications

Node.js

Suhel Hammoud

Node.js

Introduction

- Node.js is an open-source and cross-platform JavaScript runtime environment.
- It is a popular tool for almost any kind of project!

Node.js Runtime

- Node.js runs the V8 JavaScript engine, the core of Google Chrome, outside of the browser.
- This allows Node.js to be very performant.

Node.js

Single Process Architecture

- A Node.js app runs in a single process, without creating a new thread for every request.
- Node.js provides asynchronous I/O primitives in its standard library to prevent blocking.

Non-blocking I/O

- When Node.js performs I/O (e.g., reading from network or filesystem),
- it resumes operations when the response returns, avoiding blocked threads.

Efficient Concurrency

- This architecture allows Node.js to handle thousands of concurrent connections without the complexity of managing thread concurrency.

Node.js

JavaScript on Server-side

- Millions of frontend developers can now write server-side code using JavaScript, without learning a new language.

Modern JavaScript Compatibility

- You can use the latest ECMAScript standards in Node.js.
- Choose your supported features simply by changing the Node.js version or enabling flags.

A Simple Node.js Application

```
import { createServer } from 'node:http';
//const { createServer } = require('node:http');

const hostname = '127.0.0.1';
const port = 3000;

const server = createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World');
});

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

A Simple Node.js Application

Running the Application

- Save the file as `server.js` (or `server.mjs` for ES module).
- Run using the command:

```
node server.js
```

Explanation of the Code

- Includes the `http` module from Node.js standard library.
- Uses `createServer()` to create a new HTTP server.
- The server listens on a specified port and host.
- When a request comes in, it sends a plain text response.

A Simple Node.js Application

The Response Logic

Inside the server callback:

```
res.statusCode = 200;
res.setHeader('Content-Type', 'text/plain');
res.end('Hello World\n');
```

- `statusCode = 200` indicates success.
- `Content-Type = text/plain` sets response type.
- `res.end()` sends the response to the client.

Differences Between Node.js and the Browser

- Both the browser and Node.js use JavaScript.
- But building apps for them is a very different experience.

Single Language Advantage

- Node.js allows frontend developers to build backend apps using the same language they use for the frontend: JavaScript.
- This simplifies the developer experience dramatically.

Language Mastery = Productivity

- Learning one language deeply is hard.
- Node.js lets you use JavaScript everywhere – frontend and backend giving you a unique productivity edge.

Ecosystem Differences

The main difference lies in the ecosystem:

- In the **browser**: you use the **DOM**, **Web APIs**, **Cookies**, etc.
- In **Node.js**: those APIs do **not** exist.

Node.js APIs

Node.js offers modules and APIs the browser doesn't have:

- File system access
- Process management
- Network utilities
- Built-in server capabilities

Ecosystem Differences

Environment Control

- In Node.js, **you control the environment**.
  - You know what version of Node.js your app runs on.
  - In contrast, browser developers cannot control what browser users will use.

Modern JavaScript Without Transpiling

In the browser:

- You often need **Babel** to **transpile** modern JS to ES5 for compatibility.

In Node.js:

- Just use modern ES2015+ features directly, no transpilation needed.

Ecosystem Differences

Module Systems

Node.js supports:

- `require()` — CommonJS
- `import` — ES Modules (since Node.js v12)

Browsers are only **starting** to adopt `import` with ES Modules.

Summary

Feature	Browser	Node.js
DOM/Web APIs	✅ Available	❌ Not available
File system access	❌ Not available	✅ Available
Module support	`import` only	`require()` and `import`
Env control	❌ User-dependent	✅ Developer-controlled
Modern JS	Needs transpilation (Babel)	Runs natively

Introduction to npm

- npm is the standard package manager for Node.js.
- As of September 2022, over 2.1 million packages were listed in the npm registry the largest single-language code repository on Earth.

History and Scope

- npm started as a tool to manage Node.js dependencies.
- It is now also widely used in frontend JavaScript development.
- Alternatives include **Yarn** and **pnpm**.

Introduction to npm

What Does npm Do?

- Installs packages (dependencies)
- Updates packages
- Manages package versions
- Runs scripts defined in `package.json`

Dependencies are reusable libraries your app relies on.

Installing Dependencies

Installing All Dependencies (in package.json):

```
npm install
```

- This creates the node_modules/ directory (if it doesn't exist) and installs everything the project needs.

Installing a Single Package

```
npm install <package-name>
```

- Since **npm v5**, this also adds the package to package.json.
- Before v5, you had to use --save.

npm: Flags for Installing Packages

Common flags:

- `--save-dev`: adds to `devDependencies`
- `--no-save`: installs without adding to `package.json`
- `--save-optional`: adds to `optionalDependencies`
- `--no-optional`: skips optional dependencies

Shorthands:

- `-S`: `--save`
- `-D`: `--save-dev`
- `-O`: `--save-optional`

Dependencies

devDependencies vs dependencies

- **dependencies**: needed in production
- **devDependencies**: only needed in development

Use `--save-dev` for tools like test runners or bundlers.

optionalDependencies

- Will not break install if they fail to build
- Are up to you to handle in your code

Useful when a feature is non-critical or platform-specific.

npm:Uninstalling & Updating Packages

- Uninstall a specific package:

```
npm uninstall <package-name>
```

- Update all packages:

```
npm update
```

- Update a specific package:

```
npm update <package-name>
```

npm respects version constraints defined in package.json.

Versioning in npm

npm uses **semantic versioning (semver)**: MAJOR.MINOR.PATCH

You can:

- Lock to a specific version
- Define version ranges

This ensures compatibility and prevents issues with breaking changes or bugs.

Installing Specific Versions

```
npm install <package-name>@<version>
```

This helps ensure consistency across your team and builds.

Running npm Scripts

- Define custom scripts in package.json:

```
{
  "scripts": {
    "start-dev": "node lib/server-development",
    "start": "node lib/server-production"
  }
}
```

- Run scripts with:

```
npm run start-dev
npm run start
```

Example: Webpack Scripts

Define scripts for dev and prod Webpack builds:

```
{
  "scripts": {
    "watch": "webpack --watch --progress --colors --config webpack.conf.js",
    "dev": "webpack --progress --colors --config webpack.conf.js",
    "prod": "NODE_ENV=production webpack -p --config webpack.conf.js"
  }
}
```

Run with:

```
npm run watch
npm run dev
npm run prod
```

References:

- https://nodejs.org/en/learn/getting-started/introduction-to-nodejs
- https://github.com/suhelhammoud/web-course