Introduction to Node.js

Baraneetharan Ramasamy

What is Nodejs

Node.js is a powerful tool for building fast and scalable web applications. It's an open-source, cross-platform JavaScript runtime environment that runs on the V8 engine and executes JavaScript code outside a web browser. This allows developers to use JavaScript to write command-line tools and for server-side scripting—running scripts server-side to produce dynamic web page content before the page is sent to the user's web browser.

Consequently, Node.js represents a "JavaScript everywhere" paradigm, unifying web application development around a single programming language, rather than different languages for server- and client-side scripts.

Key Features of Node.js:

- **Asynchronous and Event-Driven**: All APIs of the Node.js library are asynchronous, meaning non-blocking. It essentially means a Node.js based server never waits for an API to return data.
- **Very Fast**: Being built on Google Chrome's V8 JavaScript Engine, Node.js library is very fast in code execution.

- **Single Threaded but Highly Scalable**: Node.js uses a single-threaded model with event looping. This event mechanism helps the server to respond in a non-blocking way and makes the server highly scalable as opposed to traditional servers which create limited threads to handle requests.
- **No Buffering**: Node.js applications never buffer any data. These applications simply output the data in chunks.

> https://nodejs.org/en/learn/getting-started/introduction-to-nodejs

Where to Use Node.js?

Node.js can be used for the following kinds of applications:

- Real-time chat applications
- Complex single-page applications
- REST API servers
- Real-time collaboration tools
- Streaming applications
- Microservices architecture

Where Not to Use Node.js?

It's not advisable to use Node.js for CPU-intensive operations; it's built to handle asynchronous events. Using it for heavy computation will annul nearly all of its advantages.

Remember, Node.js is not a silver bullet that can tackle all kinds of software problems. But for certain domains, such as building fast, scalable network applications, it is an excellent choice that can make your development process easier and more fun.

JavaScript runtimes

- **Bun**: Bun is a new, up-and-coming runtime that prioritizes speed and performance. It uses the JavaScriptCore Engine, which has been optimized for faster startup times and overall higher performance compared to

Node.js and Deno, which use the V8 Engine. Bun is designed as an all-in-one runtime and toolkit, offering bundling, testing, and compatibility with Node.js packages. It includes bundling and task-running features for JavaScript and TypeScript projects. While Bun is still in its early stages and may have some bugs, it shows promising potential, especially for those seeking the fastest option available.

- **Node.js**: Node.js is a well-established, mature runtime with a large ecosystem and community support. It has proven its success and stability over the years. Node.js offers built-in features such as a test runner, and discussions are ongoing about adding built-in TypeScript support. One of its advantages is the vast community and the abundance of online guidance available. However, Node.js lags behind Bun and Deno in terms of performance, particularly in database speed.
- **Deno**: Deno is a modern runtime created by the original developer of Node.js, Ryan Dahl. It emphasizes security and fine-grained access controls, operating within a secure sandbox environment. Deno also offers native TypeScript support and better web compatibility. Deno has a rich feature set that facilitates smoother development and makes it easier to build complex projects with high quality. While Deno is faster than Node.js, it is not as fast as Bun. Deno has its own community, but it is smaller compared to Node.js due to being a newer technology.

In summary:

- Choose Bun if you prioritize speed, performance, and the latest technology.
- Opt for Node.js if you value stability, a large community, and a proven track record.
- Consider Deno if security and modern features are essential for your project, and you're willing to work with a newer but feature-rich runtime.

> https://blog.bitsrc.io/should-you-use-bun-or-node-js-or-deno-in-2024-b7c21da085ba

How to create a project?

1. **Install Node.js and npm**:
    - Download and install Node.js from the official website.

    - Verify the installation by running `node -v` and `npm -v` in your terminal.
2. **Set Up Your Project Directory**:
    - Create a new directory for your project: `mkdir my-node-app`.
    - Navigate into your project directory: `cd my-node-app`.
3. **Initialize Your Node.js Project**:
    - Run `npm init` to create a `package.json` file.
    - Follow the prompts to set up your project details.
4. **Create Your Main Application File**:
    - Create a file named `app.js` or `index.js` as your entry point.
5. **Install Dependencies** (if needed):
    - Use `npm install <package_name>` to install any packages your project requires.
6. **Write Your Application Code**:
    - Add your JavaScript code to the main application file you created.
7. **Run Your Node.js Application**:
    - Execute `node app.js` to start your application.

> https://nodejs.org/en/learn/getting-started/introduction-to-nodejs

list of common npm commands in bullet points:

- `npm init`: Initialize a new Node.js project and create a `package.json` file.
- `npm install`: Install all the dependencies listed in `package.json`.
- `npm install <package_name>`: Install a specific package.
- `npm install <package_name> -g`: Install a package globally.
- `npm update`: Update all the packages to the latest version.
- `npm update <package_name>`: Update a specific package to the latest version.
- `npm uninstall <package_name>`: Remove a specific package.
- `npm start`: Run the script defined as `start` in `package.json`.

- `npm test`: Run the script defined as `test` in `package.json`.
- `npm run <script>`: Run a script defined in `package.json`.
- `npm run-script <script>`: Alias for `npm run`.
- `npm list`: List installed packages.
- `npm list -g`: List globally installed packages.
- `npm outdate`: Check for outdated packages.
- `npm search <term>`: Search for packages.
- `npm publish`: Publish a package to the registry.
- `npm cache clean --force`: Clear the npm cache.
- `npm audit`: Run a security audit to analyze the dependencies.
- `npm audit fix`: Automatically fix vulnerabilities in dependencies.
- `npm version <update_type>`: Bump the version of your package.
- `npm pack`: Create a tarball from a package.
- `npm link`: Symlink a package folder for local development.
- `npm ci`: Install a project with a clean slate based on `package-lock.json`.
- `npm rebuild`: Rebuild a package.
- `npm dedupe`: Reduce duplication in the node_modules directory.
- `npm doctor`: Check your environments and diagnose common issues.
- `npm config list`: List all npm configuration options.
- `npm config set <key> <value>`: Set a configuration option.
- `npm config get <key>`: Get the value of a configuration option.
- `npm config delete <key>`: Delete a configuration option.
- `npm owner add <user> <package_name>`: Add a user as an owner of a package.
- `npm owner rm <user> <package_name>`: Remove a user from the list of owners of a package.
- `npm access public <package_name>`: Set a package to be publicly accessible.
- `npm access restricted <package_name>`: Set a package to be accessible only to users with permission.

package.json

The **package.json** file is a fundamental component in the Node.js ecosystem and modern JavaScript development. It serves as a manifest file that contains important metadata and configuration details about a project. Here's an overview of its key aspects:

Purpose of package.json:

- **Metadata Storage**: The file holds human-readable metadata about the project, such as the project name, version, description, author, license, and keywords. This information provides a clear understanding of the project's purpose and basic details.
- **Dependency Management**: package.json lists the project's dependencies, specifying the packages required by the application. It helps manage and track these dependencies, ensuring that the project has the necessary components to function properly.
- **Project Configuration**: The file is used to configure how to interact with and run the application. It provides instructions to tools like the npm CLI (Command-Line Interface) or yarn on how to handle the project's dependencies and scripts.

Common Fields in package.json:

- **"name"**: The name of the module or project.
- **"version"**: The current version of the module.
- **"description"**: A human-readable description of what the module does.
- **"main"**: The entry point to the module, specifying the file to be returned when the module is called.
- **"repository"**: Details about the source code repository, including the type of version control and the URL.
- **"dependencies"** and **"devDependencies"**: Lists of production and development dependencies, respectively, specifying the packages required by the project.

Example package.json Structure:

```
{
  "name": "my-project",
  "version": "1.0.0",
  "description": "A sample project for demonstration.",
  "main": "app.js",
  "repository": {
    "type": "git",
    "url": "<https://github.com/user/my-project.git>"
  },
  "dependencies": {
    "package1": "^1.0.0",
    "package2": "^2.3.1"
  },
  "devDependencies": {
    "test-framework": "^1.2.0"
  }
}
```

Benefits of package.json:

- **Centralized Configuration**: It serves as a central place to configure and describe how to run and interact with the application.
- **Dependency Management**: package.json enables tools like npm to install, update, and manage the project's dependencies efficiently.
- **Project Publishing**: It is used to publish projects to registries like the NPM registry, making them accessible to other developers.
- **Script Execution**: The file allows for the execution of scripts, such as build scripts or test runners.

package.json is a crucial file in the Node.js ecosystem, providing essential information about a project, managing dependencies, and facilitating interactions with the application. It is a fundamental concept to understand when working with Node.js and modern JavaScript.

package.json vs package-lock.json

1. `package.json`:
    - Contains project metadata and configuration.
    - Lists the packages (dependencies) your project depends on.

    - Specifies the minimum version of each package your project requires.
    - Allows you to define scripts, author information, and other project-related details.
    - Typically maintained manually by developers.
    - Does not record specific package versions.
2. `package-lock.json`:
    - Automatically generated by npm (Node Package Manager).
    - Records the **exact** version of each installed package.
    - Ensures consistency across different environments by locking down dependency versions.
    - Helps re-install the same versions of packages during future installations.
    - Useful for maintaining an identical dependency tree.
    - Not meant for manual editing; npm handles it internally.

`package.json` provides high-level information about your project, while `package-lock.json` ensures precise package versions for consistent builds and deployments.

Versioning ranges

In a package.json file, the characters ^ and ~ are known as semantic versioning ranges, and they are used to specify version ranges for package dependencies. They allow you to define a range of acceptable versions for a particular dependency, providing flexibility in updating and managing dependencies. Here's an explanation of their usage:

- ^ (Caret) —
    - The caret symbol allows for updates that do not change the leftmost non-zero digit in the version number.
    - For example, ^2.3.4 would allow all 2.x.x versions but would not allow 3.0.0 or higher.
    - This is often used when you want to keep up with feature additions and minor updates but avoid potential breaking changes introduced in major

version updates.

- ~ (Tilde) —
    - The tilde symbol allows patch-level updates for the specified version. ... For example, ~1.2.3 would allow all 1.2.x versions but would not allow 1.3.0 or higher.
    - This is useful when you want to receive bug fixes and patch-level updates but avoid potential breaking changes in minor or major version updates.

In addition to ^ and ~, there are other special characters and ranges used in package.json to specify version constraints:

- (Asterisk) —
    - The asterisk is a wildcard character that matches any version.
    - For example, * would match any version of the package.
    - This is generally not recommended for production use as it can lead to unexpected updates and potential compatibility issues.
- (Greater Than) and < (Less Than) —
    - These symbols allow you to specify a range of versions.
    - For example, >1.0.0 <2.0.0 would match any version greater than 1.0.0 but less than 2.0.0.
    - This is useful when you want to ensure compatibility with a specific range of versions.
- = (Equal) —
    - The equal sign specifies an exact version match.
    - For example, =1.2.3 would only match version 1.2.3 exactly.
    - This is useful when you want to lock a dependency to a specific version, ensuring consistency across different installations or environments.

By using these special characters and ranges, you can control how your project's dependencies are updated, balancing between receiving the latest features, bug fixes, and maintaining compatibility with the rest of your codebase. It's important to carefully consider the versioning strategy for each dependency to avoid unexpected behaviour or breaking changes.

> https://medium.com/@anjusha.khandavalli/decoding-commonly-used-symbols-in-package-json-file-e08f3939c9e4

NPM, NVM, and NPX

- **NPM (Node Package Manager)**: NPM is the package manager for the Node.js platform. It is automatically installed when you set up Node.js on your computer. NPM handles the installation, updating, and management of Node.js packages and their dependencies. It offers a vast repository of packages that can be easily downloaded and installed. NPM also manages dependency conflicts intelligently and allows for package upgrades or version changes. For example, you can use NPM to install a package globally on your system: "npm install -g <package_name>".
- **NVM (Node Version Manager)**: NVM is a tool that allows you to manage multiple versions of Node.js on your system. It lets you install, uninstall, and switch between different versions of Node.js. This is particularly useful if you want to test your code against different Node.js versions or need to maintain projects with varying Node.js version requirements. NVM gives you more flexibility in managing your Node.js environment. For instance, you can use NVM to install a specific version of Node.js: "nvm install <version>".
- **NPX (Node Package Executor)**: NPX is a tool that comes bundled with NPM starting with version 5.2.0. It is a package executor that helps you execute packages from the NPM registry without installing them. NPX simplifies the process of running packages directly, making it easier to test and use packages without cluttering your system with globally installed tools. For example, you can use NPX to run a package without installing it: "npx <package_name>".

In summary:

- Use NPM to install, update, and manage Node.js packages and their dependencies.
- Utilize NVM if you need to manage multiple versions of Node.js on your system.

- Leverage NPX to execute packages from the NPM registry without installing them globally.

JavaScript Modules

AMD – one of the most ancient module systems, initially implemented by the library require.js.

CommonJS – the module system created for Node.js server.

UMD – one more module system, suggested as a universal one, compatible with AMD and CommonJS.

https://www.freecodecamp.org/news/modules-in-javascript/

CommonJS is a standard for structuring and organizing JavaScript code into reusable modules. Here’s an explanation with different examples:

What is CommonJS?

- CommonJS is a module specification used in JavaScript, particularly on the server-side with Node.js.
- It allows you to encapsulate code within modules, which can then be imported and used in other files.

Why Use CommonJS?

- **Modularity**: Breaks down code into manageable pieces.
- **Reusability**: Modules can be reused across different parts of an application.
- **Maintainability**: Easier to maintain and update code in a modular structure.

CommonJS Syntax:

- **Exporting a Module**: You can export objects, functions, or variables from a module using `module.exports`.
- **Importing a Module**: You can import the exported module using the `require()` function.

Examples:

Exporting a Module:

JavaScript

```
// math.js

function add(a, b) {
  return a + b;
}

function subtract(a, b) {
  return a - b;
}

module.exports = {
  add,
  subtract
};
```

Importing a Module:

JavaScript

```
// app.js

const math = require('./math.js');

console.log(math.add(2, 3)); // Output: 5console.log(math.s
ubtract(5, 2)); // Output: 3
```

Exporting Multiple Modules:

JavaScript

```
// utils.js

module.exports.add = function(a, b) {
  return a + b;
};

module.exports.subtract = function(a, b) {
  return a - b;
};
```

Importing Specific Functions:

JavaScript

```
// app.js

const { add, subtract } = require('./utils.js');

console.log(add(10, 5)); // Output: 15console.log(subtract
(10, 5)); // Output: 5
```

Notes:

- **CommonJS** is mainly used in Node.js environments as browsers do not natively support it.
- With the advent of ES6, **ESModules** have become the standard for JavaScript modules, which use `import` and `export` syntax.

ESModules, or ECMAScript Modules, are the official standard format for packaging JavaScript code for reuse. Here's an explanation with different examples:

What are ESModules?

- **ESModules** provide a way to export and import functions, variables, or classes from one JavaScript file to another.
- They use the `import` and `export` statements to share code between different files, promoting modularity and maintainability.

Why Use ESModules?

- **Code Organization**: Helps in organizing code into small, manageable pieces.
- **Reusability**: Makes it easy to reuse code across different parts of an application or even across different projects.

- **Tree Shaking**: Allows for more efficient bundling by eliminating unused code.

ESModules Syntax:

- **Exporting**: Use `export` to make parts of the module available to other files.
- **Importing**: Use `import` to bring in exports from other modules.

What is the use of "type": "module" in package.json

The "type": "module" field in a package.json file is used to indicate that the package contains ECMAScript (ES) modules and should be treated as such by Node.js and other tools in the JavaScript ecosystem. Here's an explanation of its use:

1. ES Modules Support —
    - In modern JavaScript, the concept of modules has been introduced, allowing developers to encapsulate code into reusable and independent units.
    - By specifying "type": "module", you are indicating that the package contains ES modules, which use the import and export keywords for module loading and sharing functionality.
2. Improved Tree-Shaking —
    - Tree-shaking is an optimization technique used by bundlers and module loaders to eliminate unused code from the final bundle or application.
    - When "type": "module" is specified, tools like webpack or Rollup can perform more effective tree-shaking, resulting in smaller bundle sizes and improved performance.
    - With ES modules, the static nature of imports and exports makes it easier for bundlers to analyze and remove unused portions of the code.
3. Better Code Organization —
    - Using ES modules promotes better code organization and separation of concerns.
    - By dividing your code into smaller, focused modules, you can improve readability, maintainability, and reusability.

    ... Each module can have its own set of exports, making it easier to understand and reason about the code.

4. Interoperability with Modern Tools —
    - Specifying "type": "module" ensures better interoperability with modern tools and environments that support ES modules.
    - For example, when using Node.js with the --experimental-modules flag or in environments that natively support ES modules (such as modern browsers), the "type": "module" field signals that the package should be treated as an ES module.
5. Improved Security —
    - ES modules provide improved security compared to traditional CommonJS modules.
    - With ES modules, the import statements are statically analyzed, allowing bundlers and loaders to perform static analysis and detect potential security issues, such as missing or incorrect dependencies.

By setting "type": "module" in your package.json, you are explicitly declaring that your package follows the ES modules specification, enabling better tooling support, improved performance, and enhanced security for your project. It also ensures that your package can take advantage of the latest features and improvements in the JavaScript ecosystem related to module loading and interoperability.

Examples:

Exporting a Single Value:

JavaScript

```javascript
// utils.js

export const PI = 3.14159;
```

Importing a Single Value:

JavaScript

```
// app.js
import { PI } from './utils.js';

console.log(PI); // Output: 3.14159
```

Exporting Multiple Values:

JavaScript

```
// math.js

export function add(a, b) {
  return a + b;
}

export function subtract(a, b) {
  return a - b;
}
```

Importing Multiple Values:

JavaScript

```
// app.js
import { add, subtract } from './math.js';

console.log(add(5, 3)); // Output: 8console.log(subtract(5,
3)); // Output: 2
```

Default Export:

JavaScript

```
// defaultExport.js
export default function() {
  console.log('This is the default export');
}
```

Importing Default Export:

JavaScript

```
// app.js

import myDefaultFunction from './defaultExport.js';

myDefaultFunction(); // Output: This is the default export
```

Renaming Exports and Imports:

JavaScript

```
// math.js

export const squareRoot = Math.sqrt;

// app.jsimport { squareRoot as sqrt } from './math.js';

console.log(sqrt(16)); // Output: 4
```

Dynamic Import:

JavaScript

```
// This is useful for code-splitting and lazy-loading modul
esif (condition) {
  import('./module.js').then((module) => {
    module.doSomething();
  });
}
```

Notes:

- **Browser Support**: Modern browsers support ESModules natively.
- **File Extension**: JavaScript files using ESModules typically have the `.mjs` extension when used on the server-side with Node.js.

- **Static & Dynamic Analysis**: ESModules allow for static analysis of dependencies and can also support dynamic imports.