JavaScript Part 10: Node.js and npm Basics

So far, JavaScript has lived inside the browser. Node.js is a free program that runs JavaScript on an ordinary computer or server, with no web page at all. Four steps take us from the browser to real automation: run, install, serve and automate. Each step answers one simple question. In this part, Samer, an IT coordinator at a logistics company, uses them to find out what is filling the shared drive, which is almost full.

1. Run

Node.js runs a JavaScript file from the terminal: we type node, then the file name, and the output appears in the terminal. The question is: what does my script need from the computer? There is no page, so there is no document and no window, but there are built-in modules for the computer itself. The fs module reads and writes files, the path module works with file names and extensions, and the http module speaks to the network. We bring a module in with an import statement, written with the prefix node and a colon, and the file name ends in dot mjs. Whatever we type after the file name arrives in the process dot argv list, so one script can work on any folder we name. Code in a terminal has real power: it can change and delete real files, so we test on a copy first.

Example: A bakery recipe works in any kitchen, and it does not need the shop window. The baker follows the steps written on a card, and the same bread comes out every time. A Node.js script is that recipe without the shop window: the file holds the steps and the terminal is the kitchen. We run it, and the result appears right there.

At work: Samer writes a file called report dot mjs. It imports the fs and path modules, takes the folder name from process dot argv, and prints a first line for every file it finds. He runs it on a small test folder, not on the shared drive, and the list appears in the terminal in a moment.

2. Install

Node.js comes with npm, the package manager. A package is reusable code that someone else wrote and published to the npm registry. The question is: should I write this myself, or use a package that already exists? A project starts with a package dot json file, which records its name, its scripts and its dependencies. When we install a package, npm downloads it into the node modules folder, adds its name to dependencies in package dot json, and records the exact versions it installed in package lock dot json. The node modules folder is not shared; a colleague who has the project runs npm install and rebuilds it. Tools needed only during development are saved as dev dependencies. Every package is code written by strangers that runs with our permissions, so we prefer well-known, maintained packages with few dependencies, and we run npm audit, which checks our dependencies against a list of known vulnerabilities.

Example: A builder who needs a window buys a finished one from a supplier instead of making glass. But the builder checks the supplier first, because a bad window is built into the house for good. A package is a finished window for code: it saves work, but it becomes part of our project, so we check who wrote it.

At work: Samer wants sizes shown as megabytes and gigabytes, not as long numbers of bytes. He decides that a tiny function of his own is safer than a new dependency, and he writes it. For the next project, he does install a well-known testing tool as a dev dependency. A colleague copies the project without node modules, runs npm install, and gets exactly the same versions from package lock dot json.

3. Serve

A server is a program that waits for requests and sends answers. The http module of Node.js creates one in a few lines. We give it a function that receives the request and the response; the function sets a status code,

This document is a property of Asaseeyat Academy - © Copy Right Protected.

such as two hundred for success or four hundred and four for a page that does not exist, and ends the response with the text to send. The server then listens on a port, such as three thousand. The question is: who needs to ask this program a question, and what should it answer? While it runs, the address localhost followed by the port opens in a browser on the same computer. Other computers can reach it only if the network and the firewall allow it, so a server for colleagues must be agreed with IT first. A tiny server teaches how the web works, but a real public service needs much more: security, secure connections and monitoring.

Example: A reception desk answers one kind of question: where is the meeting room? A visitor asks, the desk answers, and the next visitor can ask. The desk never starts the conversation; it waits and replies. An http server is a desk like that: it waits for a request, sends one answer, and stays ready for the next.

At work: Samer's team lead wants to check the drive without running a script. Samer creates a server that, for each request, builds the report and sends it as a short page. He opens localhost on port three thousand on his own computer, and the numbers appear in the browser. He stops the server with control and C, and he agrees with the network team before anyone else connects.

4. Automate

A script that we run by hand is useful, but a named command is better. In package dot json, the scripts section maps a short name to a command. For example, a script named report can run node report dot mjs, and then everyone starts it with `npm run report` and does not need to remember file names. This is the question: how can anyone repeat this task exactly, tomorrow or next month? Reports that run automatically, for example every week, are the usual next step, and the operating system or the company's tools can do the scheduling.

Example: A company prints the same weekly sales sheet. Instead of explaining how to build it every time, the manager gives it a name on a shelf and says, take the weekly sheet. New staff ask for it by name and get the same sheet. A script entry is that name on the shelf: one short word that starts the whole job.

At work: Samer finishes the report. It reads the shared folder and its subfolders with the fs module, asks for the size of every file, uses the path module to find the extension, and adds the totals for each extension in an object. It prints them in a table with console dot table, largest first. He adds a script named report to package dot json, and the team can run npm run report. The table shows that exported training videos from a finished campaign fill about two thirds of the drive. The team archives them, and the purchase of new storage waits.

Conclusion

The shared drive was not a mystery after all. Run took JavaScript out of the browser, and the built-in modules let it read files. Install showed how npm brings in other people's code through package dot json, and why we choose carefully and run npm audit. Serve turned a script into a tiny program that answers requests. Automate gave the report a name, so anyone can repeat it. We keep each script small, test on copies, and decide before every install whether we really need the package.

Sources

- Node.js Documentation: File system (fs) module, readdir, stat and Stats size – https://nodejs.org/api/fs.html
- Node.js Documentation: HTTP module (createServer, writeHead, end, listen) – https://nodejs.org/api/http.html
- Node.js Documentation: Process (process.argv) – https://nodejs.org/api/process.html
- Node.js Documentation: ECMAScript modules (mjs extension and the type field in package.json) – https://nodejs.org/api/esm.html
- npm Docs: package.json (dependencies, devDependencies, scripts) – https://docs.npmjs.com/cli/v10/configuring-npm/package-json
- npm Docs: npm audit – https://docs.npmjs.com/cli/v10/commands/npm-audit
- Node.js and npm are names of their respective owners. This video is independent training material.

This document is a property of Asaseeyat Academy - © Copy Right Protected.