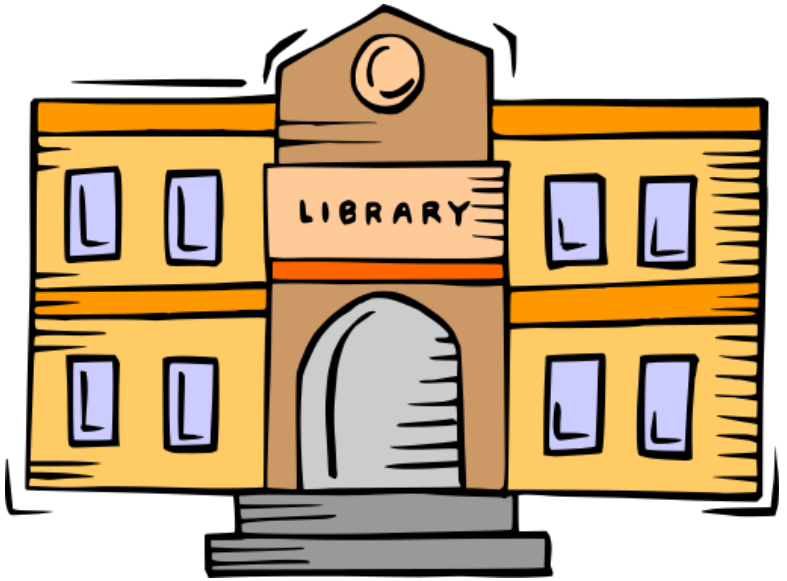
<WA/>
2025

JavaScript: Libraries and Packages



“The” language of the Web

Enrico Masala
Antonio Servetti

Outline

- Modules, libraries, packages
- Package manager
- Package example: handling dates
- Issues and security concerns

Definition

- **Module**: a Javascript file, having its own scope isolated from the rest
  - Will be described later in the course
- **Library**: a group of modules and/or scripts that implements a desired function (e.g., handling dates/times)
  - Similarly to other programming languages
- **Package**: one or more libraries organized so that they can be easily integrated into projects

Handling Packages

- **Registry:** a database of software packages that facilitates searching, download, and dependency management
- In the Javascript community: *NPM Registry* is the largest (2M+ packages)
  - https://www.npmjs.com

- **Package manager**: software tools to automate managing packages in a standardized manner
  - An assistant to the developer to install the packages required by the developer while automatically managing package versions and their dependencies
  - **Dependencies**: other packages (software and libraries) needed to run a package

Javascript Package Managers

- **`npm:`** the default for the node.js environment
  - npm is the “node package manager”
  - npm is also a “repo platform” now maintained by GitHub (owned by Microsoft)

In the rest of the course: we focus on **`npm`**

- **`yarn:`** alternative to npm developed by Facebook
  - faster, can work offline in certain cases, but some drawbacks: not compatible with older node versions, issues with native code packages
- **`pnpm:`** npm improvement (“performant npm”)
  - Not so popular, compatibility issues with some packages

https://romanglushach.medium.com/comparing-npm-yarn-and-pnpm-package-managers-which-one-is-right-for-your-distributed-project-to-4d7de2f0db8e

How does a package manager work?

Example 1: Developer tells the package manager to import a new package in the project. The package manager **automatically**:

1. Queries a software registry to find the package
2. If found, obtains the information about to download it
3. Downloads it
4. Includes it in the project
   - Typically in a standardized form, e.g. in `package.json` for the case of `npm`
5. Checks dependencies, and restart from 2 until all dependencies are met

How does a package manager work?

Example 2: Developer tells the package manager to install all packages that are required by a project

1. The package manager reads the list of packages required by the project
   - From `package.json` for the case of `npm`
2. Then it proceeds to include them in the project as in previous example

The developer:

- share only the application code
- does not need to care about other code (packages) that will be managed automatically

Example: Day.js Package

DAY.JS https://day.js.org/

- Install (from command line)

```
# initialize the package manager files in the project
# if not already done (choose a name and default for the rest)
npm init

# download from registry, add to project package list
# make it available to the scripts in the project
npm install dayjs
```

package.json

```
{
  "name": "my-project",
  "version": "1.0.0",
  "main": "index.js",
  . . .
  "dependencies": {
    "dayjs": "^1.11.10"
  }
}
```

Folder Structure after running npm

```
my-project
├── node_modules
├── package.json
├── package-lock.json
└── index.js
```

- `my-project` is the project root
- `node_modules` is the folder where packages are installed. This is automatically managed/reconstructed by npm, do not touch!
- `package.json` contains (also) the list of packages needed by the project, with their minimum version
- `package-lock.json` contains the list of packages actually installed in the project, with more details (version, package hash)
- `index.js` is the code of the project
  - **Develop here!**
  - Insert the `require()` statement here to use the package

Example: Day.js Package usage in node

- In the Javascript file, after the package has been installed

index.js

```
// import (using name of my choice)
const dayjs = require ('dayjs');

// use (depends on the specific package)
let now = dayjs();
console.log(now.format());
```

Security and Critical Issues with External Code

Security: The package code is:

- running in your project
- with the **same privileges as your code**
- same possibility to access data
- how can it be trusted?

Availability: If the package and/or a specific package version is not available anymore:

- how can the project be run?

Checking for Bugs/Vulnerabilities

- **First**: do not forget to check for known bugs/vulnerabilities!

  `npm audit`

  NB: `npm audit fix` ONLY updates the package version(s)! https://docs.npmjs.com/cli/v10/commands/npm-audit

```
@babel/traverse  <7.23.2
Severity: critical
Babel vulnerable to arbitrary code execution when compiling specifically crafted malicious code - https://github.com/advisories/GHSA-67hx-x53-jw92
fix available via `npm audit fix`
node_modules/@babel/traverse
```

...

```
vite  4.2.0 - 4.2.2
Severity: high
Vite Server Options (server.fs.deny) can be bypassed using double forward-slash (//) - https://github.com/advisories/GHSA-353f-5xf4-qw67
fix available via `npm audit fix`
node_modules/vite

4 vulnerabilities (2 moderate, 1 high, 1 critical)

To address all issues, run:
  npm audit fix
```

Security

How to trust packages written by others?

- By reading and checking it yourself or by some trusted entity (e.g. your organization / company)
- And by using formal verification tools
- And …
- Do not rely only on unknown / untrusted entities, and/or reputation

Result

- An *approved* package that can be securely used in your projects / organization / company

Secure distribution of packages

How to force package managers use only approved packages?

- Search/install only from a private registry containing only approved packages (serious companies always have one)
  - The private registry/database solves also the availability issue because:
    - the package is stored in computers systems you control thus it cannot disappear
    - the package cannot change its content due to malicious attacks
- A cryptographically secure hash applied to the package helps in checking if the content of the package changed (even if name and version are the same!)
  - `package-lock.json` contains both the hash and the version of the package

Forcing specific packages

- Always use `package-lock.json`, distribute it with your code
  - npm will always check hash, will always try to install dependencies whose version matches the ones listed in the `package-lock.json` file.
- Note the subtle differences in npm install commands

```
npm install
# if dependencies already exist
in node_modules, use them;
it creates package-lock.json if it
doesn't exist, no hash/version check!

npm ci
# ci = clean install,
for reproducible builds in automated
environments: delete node_modules
then start from scratch
```

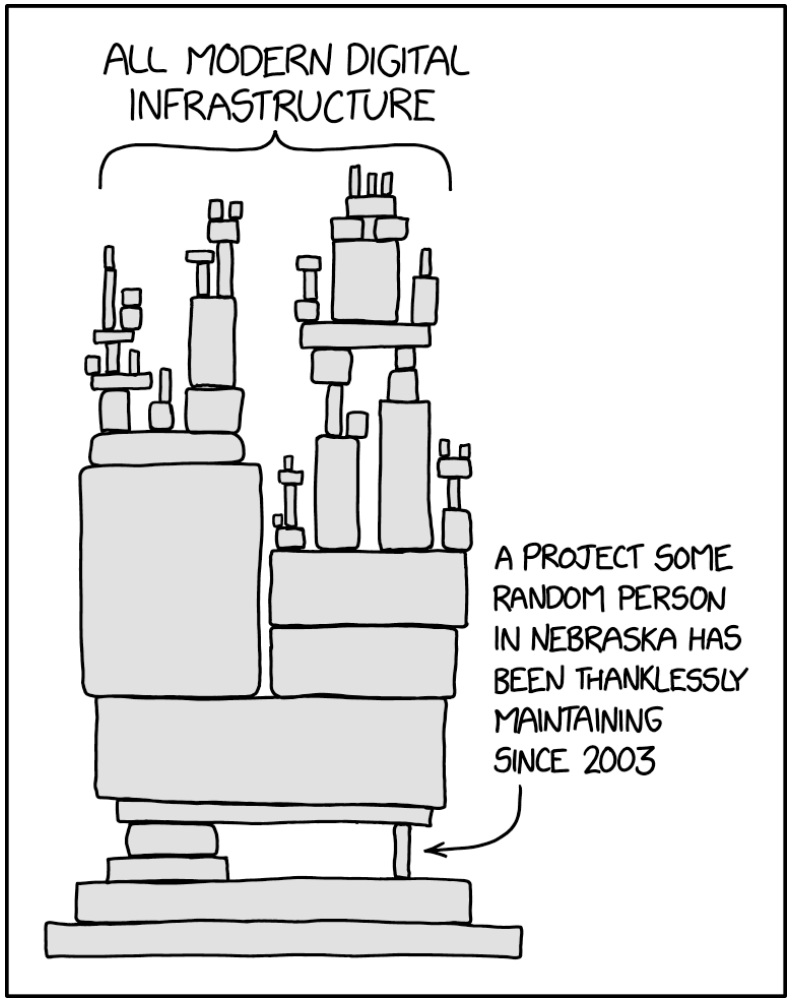
`package-lock.json`

```
{ "name": "my-project",
  . . .
  "packages": {
    . . .
    "node_modules/dayjs": {
      "version": "1.11.10",
      "resolved": "https://registry.npmjs.org/dayjs/-
                                          /dayjs-1.11.10.tgz",
      "integrity": "sha512-vjAczensTgRcqDERK0SR2XMwsF/ … "
    }
} }
```

https://cheatsheetseries.owasp.org/cheatsheets/NPM_Security_Cheat_Sheet.html

Choosing libraries/packages

- Beware: before choosing libraries, packages, etc. consider:
  - Licensing
  - Community / long term support
  - Security
  - Documentation and code quality



https://xkcd.com/2347/

License

- These slides are distributed under a Creative Commons license “**Attribution-NonCommercial-ShareAlike 4.0 International (CC BY-NC-SA 4.0)**”
- **You are free to:**
  - **Share** — copy and redistribute the material in any medium or format
  - **Adapt** — remix, transform, and build upon the material
  - The licensor cannot revoke these freedoms as long as you follow the license terms.
- **Under the following terms:**
  - **Attribution** — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
  - **NonCommercial** — You may not use the material for commercial purposes.
  - **ShareAlike** — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.
  - **No additional restrictions** — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.
- https://creativecommons.org/licenses/by-nc-sa/4.0/