Igniting our App

▼ What is npm ?

npm is a package manager for the JavaScript programming language which is used to install and manage packages(modules, libraries) that are published on the npm registry.

A plethora of Node.js libraries and applications are published on npm, and many more are added every day. These applications can be searched for on https://www.npmjs.com/

```
npm install <pcakage-name>

        OR

 npm i <package-name>
```

▼ What is Parcel/ webpack ? Why do we need it ?

Parcel and Webpack are bundlers, where bundlers are tools that take our code and its dependencies, package them together into a single file or few files, and can be run on a browser or a web server.

They can be used to transform code written in one language (such as TypeScript or JSX) into regular JavaScript that can be run in a web browser, or to optimize and minimize the bundled code to improve performance.

▼ What is .parcel-cache ?

When we want to bundle our app with parcel:

```
npx parcel index.html
```

A folder called .parcel-cache will be generated. Parcel uses this for perform Hot Module Replacement (HMR).

Caching

Parcel caches everything it builds to disk. If you restart the dev server, Parcel will only rebuild files that have changed since the last time it ran. Parcel automatically tracks all of the files, configuration, plugins, and dev dependencies that are involved in your build, and granularly invalidates the cache when something changes. For example, if you change a configuration file, all of the source files that rely on that configuration will be rebuilt.

By default, the cache is stored in the `.parcel-cache` folder inside your project. You should add this folder to your `.gitignore` (or equivalent) so that it is not committed in your repo. You can also override the location of the cache using the `--cache-dir` CLI option.

Caching can also be disabled using the `--no-cache` flag. Note that this only disables *reading* from the cache – a `.parcel-cache` folder will still be created.

Development

Parcel includes a development server out of the box supporting hot reloading, HTTPS, an API proxy, and more.

https://parceljs.org/features/development/

▼ What is npx ?

npx is a command-line tool that is included with npm (Node Package Manager) that allows you to **execute Node.js packages**. It is designed to be an easy and convenient way to run packages without having to install them globally on your system.

```
# Exceute pracel using npx for the entry point index.htmnl

npx parcel index.html
```

If you install something globally, it's not installed in a project's node_modules folder. Instead, it's installed in a computer-specific directory, so it's available for all Node.js projects on the computer. To install a global package, add the `-g` flag to the `install command`, so the command looks like this one: `npm install <dependency name> -g`.

However, the npx tool was created in part to address the fact that globally installed dependencies take up space after many of them have been installed over time. Also, on many operating systems, a globally installed tool needs elevated permissions for installation. The main point is that global dependencies complete tasks that you most likely want to perform infrequently. So there's little need for these tools to be on your computer for more than a temporary period of time.

The npx tool allows you to load the dependency into the Node.js process and run the command from there. After the command is run, the dependency is cleaned up and removed from your system. The npx tool has been shipped with all major versions of npm since version 5.2. This tool is the preferred way to use dependencies that are meant to be run infrequently. To use the npx tool, enter `npx <name of package>`. It will fetch the dependency, run the command, and clean up.

▼ What is the difference between `dependencies` vs `devDependencies` ?

The package.json file is a manifest file for our project. It contains metadata information on our project such as the project name, description, author, keywords, and so on. It also governs things like how dependencies are managed, what files go into a package, and much more.

Initialize a package.json

A package.json file isn't something you author by hand. It's the result of running the npm `init` command. There are two main ways to run this command:

- `npm init`: This command starts a wizard that prompts you for information about a project's name, version, description, entry point, test command, Git repository, keywords, author, and license.
- `npm init -y`: This command uses the `-y` flag, and is a faster version of the `npm init` command because it's *not* interactive. Instead, this command automatically assigns default values for all values you're prompted to enter by using the `npm init`.

The `npm init` and `npm init -y` commands both generate a package.json file. Here's an example:

```
{
  "name": "namaste-react",
```

```
  "version": "1.0.0",
  "description": "Namaste React Course",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "repository": {
    "type": "git",
    "url": "git+https://github.com/NishantGautam023/Namaste-React.git"
  },
  "keywords": [
    "Namaste-React",
    "React",
    "JavaScript"
  ],
  "author": "Nishant Gautam",
  "license": "ISC",
  "bugs": {
    "url": "https://github.com/NishantGautam023/Namaste-React/issues"
  },
  "homepage": "https://github.com/NishantGautam023/Namaste-React#readme",
  "devDependencies": {
    "parcel": "^2.8.2",
    "process": "^0.11.10"
  },
  "dependencies": {
    "react": "^18.2.0",
    "react-dom": "^18.2.0"
  }
}
```

- **Production dependencies**: Production dependencies are dependencies that we need to run when your application is in production. A production dependency must be built into our application so the functionality is available when the application is running. Examples include a web framework with which we can build a web application.
- **Development dependencies**: Development dependencies are dependencies that we need only when we develop our application. Think of these dependencies as of scaffolding for a building. When we are done building, we don't need them anymore. Examples of development dependencies are test libraries, linting tools, or bundling tools. These dependencies are an important part of ensuring our application works well, but we don't need to ship our application with them.

▼ What is Tree Shaking ?

In production builds, Parcel statically analyzes the imports and exports of each module, and removes everything that isn't used. This is called "tree shaking" or "dead code elimination". Tree shaking is supported for both static and dynamic `import()` , CommonJS and ES modules, and even across languages with CSS modules.

▼ What is Hot Module Replacement ?

HMR improves the development experience by updating modules in the browser at runtime without needing a whole page refresh. This means that the application state can be retained as you change small things in your code. The parcel has a file watcher algorithm (written in c++ very fast, keeps track of files that are changing in real-time, and tells server to reload)

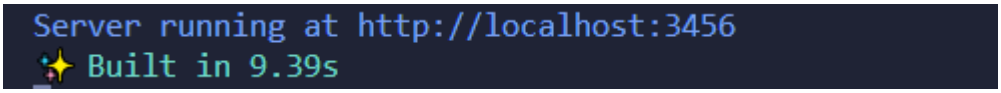
▼ List down the 5 superpowers of Parcel and describe any 3 of them in your own words.

As we know the parcel is a beast, it has many features,

- Dev Server: By default when we open our server it opens at port: 1234. If port `1234`
  is already in use, then a fallback port will be used. After Parcel starts, the location where the dev server is listening will be printed to the terminal. We can even open in our favourite web browser.

```
npx parcel index.html --port 3456

npx parcel index.html --open safari.
```



- Hot Reloading: Whenever we make changes to our code, parcel automatically detects it and does the changes in the browser. Parcel has HMR (Hot module Replacement) which improves the development experience in the browser by updating modules in the browser at runtime without needing a whole page refresh.
- HTTPS: Sometimes we need to test code which are only possible in HTTPS like authentication cookies etc. Parcel provides us HTTPS support

  ```
  parcel src/index.html --https
  ```

  Other features includes

  - Bundling
  - Caching While Development

▼ What is `.gitignore`? What should we add and not add into it?

`.gitignore` is a file that you can create in a Git repository to tell Git which files or directories to ignore when you make a commit. This can be useful if you have files that you don't want to track, such as temporary files or build artifacts.

Anything that can be autogenerated should be put in .gitignore. For ex: With the help of package.json we can generate node_modules folder, thus it should be put in .gitignore

▼ What is the difference between `package.json` and `package-lock.json`

- **Major version**: The leftmost number. For example, the 1 in 1.0.0. A change to this number means you can expect breaking changes in the code. You might need to rewrite part of your code.
- **Minor version**: The middle number. For example, the 2 in 1.2.0. A change to this number means features have been added. Your code should still work. It's generally safe to accept the update.
- **Patch version**: The rightmost number. For example, the 3 in 1.2.3. A change to this number means a change has been applied that fixes something in the code that should have worked. It should be safe to accept the update.

This table shows how the version number changes for each version type:

Type	What happens
Major version	1.0.0 changes to 2.0.0
Minor version	1.2.9 changes to 1.3.0
Patch version	1.0.7 changes to 1.0.8

Pattern	Update level
x.0.0 or * (asterisk)	Update to the highest *major* version.
1.x.1 or ^ (insert or caret)	Update to only the *minor* version.
1.1.x or ~ (tilde)	Update to the latest *patch* version.1

When we install packages the package.lock.json file is automatically generated, which contains detailed record of the exact versions of all of the packages and their dependencies that were installed in the project. This can be useful for ensuring that the same versions of packages are installed on different machines, even if the latest versions of those packages have changed.

```
For example, the package.json file might specify that the project
depends on lodash version ^4.17.15, which would allow npm to install
any version of lodash that is compatible with version 4.17.15.
The package-lock.json file, on the other hand, would specify the exact
version of lodash that was installed, such as 4.17.19.
```

▼ Why should I not modify `package-lock.json`?

This file is intended to be managed by npm and is used to ensure that the exact same versions of packages are installed on different machines.

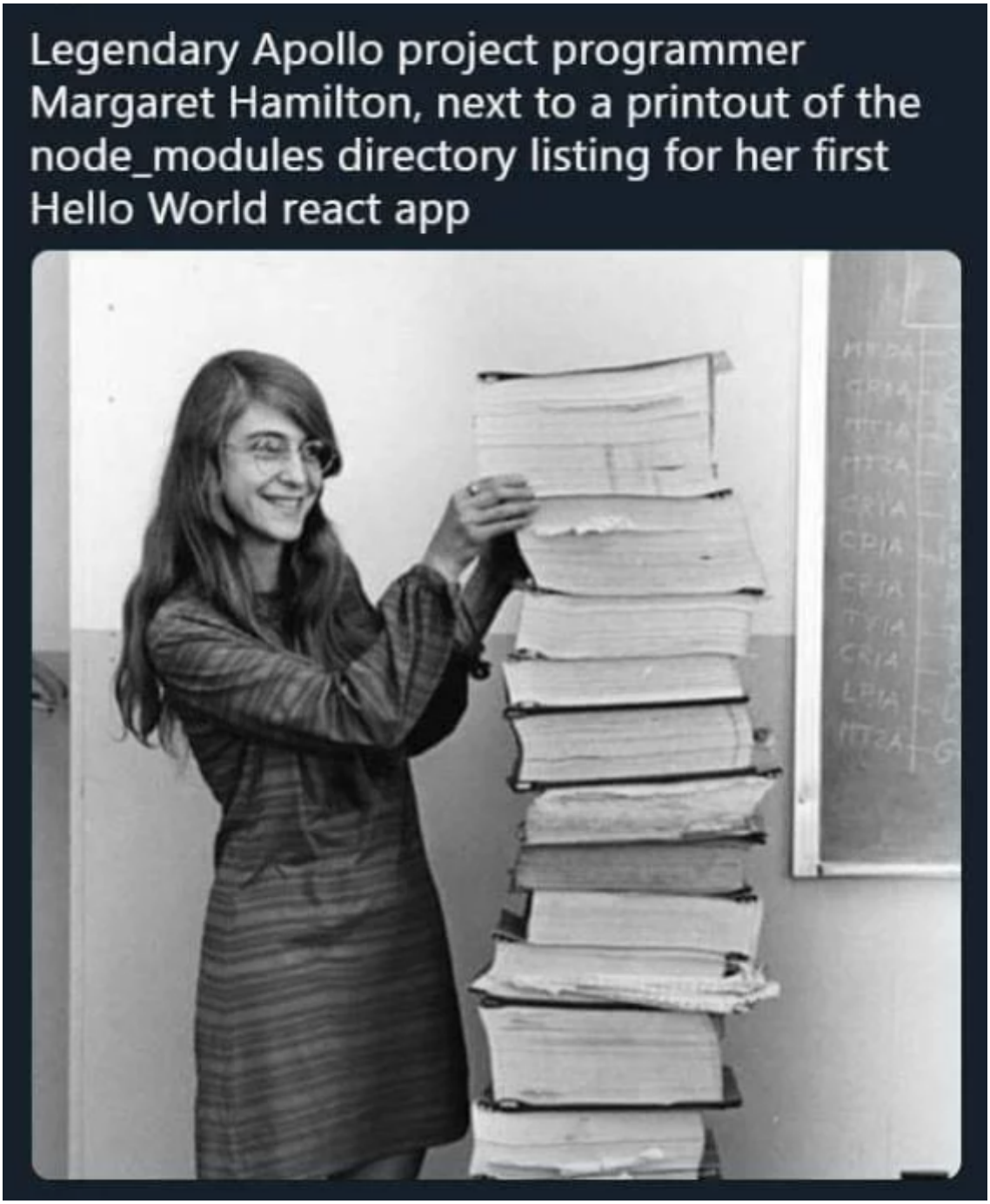
Modifying this file can cause conflicts with the dependencies that are specified in the `package.json` file, and can lead to issues when you try to install or update packages in the future.

▼ What is `node_modules` ? Is it a good idea to push that on git?

`node_modules` is a directory that is created by the Node package manager (npm) when you install packages in a Node.js project. It contains all of the packages and their dependencies that are required by your project, as well as any scripts that are associated with the packages.

When we are building a production-ready app we need a lot of things like compression, minification, optimization, caching, bundling, hot reload etc . we can’t do this alone and we need some dependencies and those dependencies

are also dependent on other dependencies it creates a dependencies tree. this is called **transitive dependency.** **Thats why node_modules is heavy.**

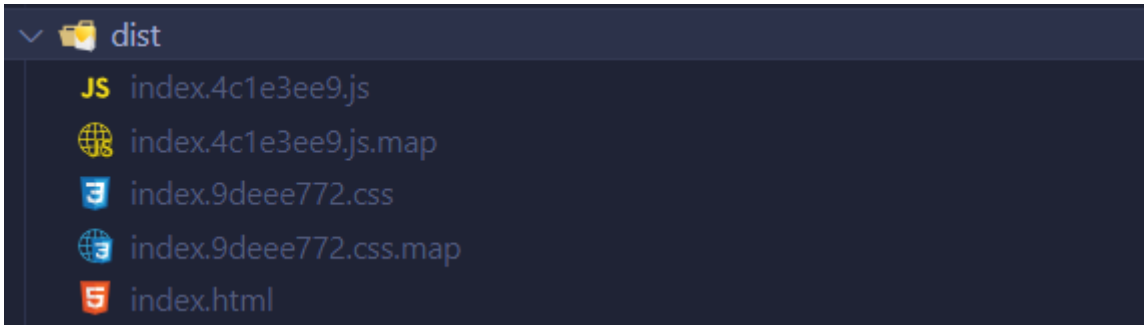


▼ What is the `dist` folder?

When we bundle our app using parcel

```
npx parcel index.html
```

A folder called dist will be generated. This is a directory that contains the final, compressed version of our project and this is the directory we deploy to our production environment.



The `dist` folder typically contains the HTML, CSS, and JavaScript files that make up your project, as well as any assets such as images or fonts. These files are generated by Parcel from the source code and assets in your project, and they are optimized for production use.

▼ What is `browserlists`

`browserlist` is a configuration file that is used to specify which browsers and versions should be targeted by a project that is built using tools like Autoprefixer, Babel, or PostCSS. It is typically used to ensure that the project is compatible with a specific set of browsers, and to automatically add vendor prefixes to CSS rules or polyfills to JavaScript code as needed.

A `browserlist` file is typically a plain text file with a `.browserlistrc` extension that is located in the root directory of a project. It consists of a list of browser names and version ranges, separated by commas. For example, a `browserlist` file might contain the following:

```
last 2 versions
```

Browserslist

Use if you're building a web application for the global audience. Use if you're building a Node.js application, e.g., for server-side rendering. Autoprefixer, Babel and many

https://browsersl.ist/#q=last+2+versions

Shared browser and platform compatibility config for popular JavaScript tools

Browserslist

Check compatible browsers

```
last 2 versions
```

The **not dead** query skipped when using **last N versions** query Fix

Less than 80% coverage in **China**, **United States**, **Russia**, **Japan**, **Germany**, and 15 more regions Fix

Audience coverage: 75.9 %

Region Global

Chrome for Android | Chrome

Chrome for Android	108	41.7 %	Safari	16.2	0.00 %
Chrome	108	0.06 %		16.1	0.57 %
	107	18.6 %	IE	11	0.42 %
Edge	108	0.00 %		10	0.01 %
	107	4.0 %	Firefox for Android	107	0.29 %
Safari on iOS	16.2	0.00 %	QQ Browser	13.1	0.12 %
	16.1	3.1 %	KaiOS Browser	2.5	0.04 %
Samsung Internet	19.0	1.5 %	IE Mobile	11	0.02 %
	18.0	0.66 %		10	0.01 %
Firefox	107	0.87 %	Opera Mobile	72	0.01 %
	106	1.3 %		12.1	0.00 %
Opera	92	0.68 %	Android Browser	108	0.00 %
	91	0.41 %	Baidu Browser	13.18	0.00 %
Opera Mini	all	0.96 %	Blackberry Browser	10	0.00 %
UC Browser for Android	13.4	0.68 %		7	0.00 %