NPM

What is Npm?

- npm is the **package manager** for the Node.js JavaScript runtime environment. It is an online repository for publishing and installing packages (libraries or modules) that can be used in **Node.js** applications, a popular JavaScript runtime environment.
- npm also has **command-line tools** to help developers run the npm command-line tool to install or uninstall packages and manage versions or dependencies.
- npm is free and relied on by over 11 million developers worldwide. They're **open-source** and have become the center of JavaScript code sharing. There are more than a million packages available on npm.
- Npm started as a private organization; GitHub acquired npm in 2020.

Why Use npm?

- While we can certainly manage the **dependencies** for our project manually, as the project grows in size and complexity, this task can become overwhelming. This is where a package manager like **NPM** comes in. **npm** solves this problem by handling **dependency** and **package management** for our project.
- We can define all our project's dependencies inside our **package.json** file. Anytime we or a team member needs to get started with our project, all they have to do is run "**npm install"** or **"npm I".**
- This will immediately **install** all the necessary **dependencies** for our project. In the **package.json** file, we can also specify which versions our project depends upon. This is useful to prevent updates from these packages from breaking our project.

Here are some of the reasons why we should use npm:

- It enables us to install **libraries**, **frameworks**, and other **development tools** for our project, similar to installing a mobile application from an app store.
- It helps us **speed up the development phase** by using **prebuilt dependencies**.
- npm has a wide variety of tools to choose from for no cost.
- Using **npm commands** doesn't require a lot of learning, as they're easy to understand and make use of.

The npm Command Line Interface (CLI): -

- The **command line interface** for npm is used to run various commands like **installing** and **uninstalling** packages, to check **npm version**, **run package scripts**, create **package.json** file, and so much more.

Essential npm Commands: -

npm init:

- The **init** command is used to **initialize a project**. When we run this command, it creates a package.json file.
- When running **npm init**, they'll be asked to provide certain information about the project we're initializing. This information includes the **project's name**, the **license type**, the **version**, and so on.
- To skip the process of providing the information our self, we can simply run the "**npm init -y"** command.

npm install or (npm i):

- This command is used to **install packages**. We can either install packages **globally** or **locally**. When a package is installed **globally**, we can make use of the package's functionality from any directory in our computer.
- On the other hand, if we install a **package locally,** we can only make use of it in the directory where it was installed. So, no other folder or file in our computer can use the package.

npm uninstall:

- **npm uninstall <package-name>**: - This command is used to uninstall a package.

npm update:

- **npm update <package-name>: -** We can Use this command to update a npm package to its **latest version.**

What Is an Npm Package?

- A package is simply a prebuilt project published on the **npm directory**. What packages can do depends solely on the **creator** of, and **contributors** to, the package.
- With **npm**, we can access numerous projects created by other developers. Imagine creating our own **CSS framework**; that would take a lot of time to do. So, developers create these projects and put them on the **npm** registry so we can easily use them and ease the **development process**.
- One example of such a **npm package is Tailwind CSS**, a utility-first CSS framework for building web pages. Other popular npm packages include **React**, **Chalk**, **Gulp**, **Bootstrap**, **Express**, and **Vue.js**, among many more.

How to Install an NPM Package Globally: -

- When we install an npm package **globally**, we're able to access it from any directory on our computer.
- To install a package **globally**, use this command:
  "**npm install -g [package name]"**

Note that the **-g** flag in the command is what enables the **npm CLI** to install the package **globally**.

How to Install an NPM Package Locally: -

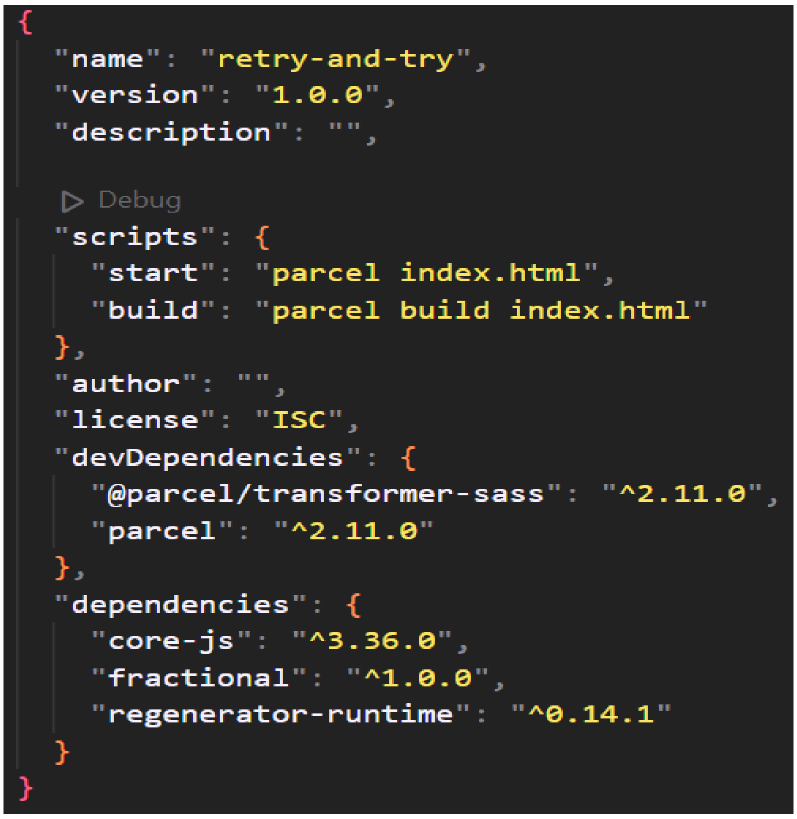
- To install a package **locally** means we can only use the package's functionality in the **current directory**. To do this, we'd have to navigate to the directory we want to install the package and run this command in the terminal:

  "npm install [package name]"

What is a package.json?

- The package.json file is in **JSON** format and it is a **metadata file** that describes the **project's dependencies**, **scripts**, **configuration**, and other details.
- It typically contains information about the project such as its **name, version, author, and license**. It also lists the project's dependencies. along with their version numbers, so that these dependencies can be automatically installed when the project is set up or updated.
- The package.json file is typically located at the root directory of a **Node.js/JavaScript** project and is automatically generated when we run "**npm init"** command to initialize a new project. We can also manually create and modify this file to manage our project's dependencies and configuration**.**
- This file is essential for **Node.js/JavaScript** projects as it provides a standardized way to manage **dependencies** and define **project configuration**, making it easier to share and collaborate on projects with others.

Here's a simple example:



```
{
  "name": "retry-and-try",
  "version": "1.0.0",
  "description": "",

  ▷ Debug
  "scripts": {
    "start": "parcel index.html",
    "build": "parcel build index.html"
  },
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "@parcel/transformer-sass": "^2.11.0",
    "parcel": "^2.11.0"
  },
  "dependencies": {
    "core-js": "^3.36.0",
    "fractional": "^1.0.0",
    "regenerator-runtime": "^0.14.1"
  }
}
```

There are two ways of generating a package.json file using "npm" or by using "yarn":

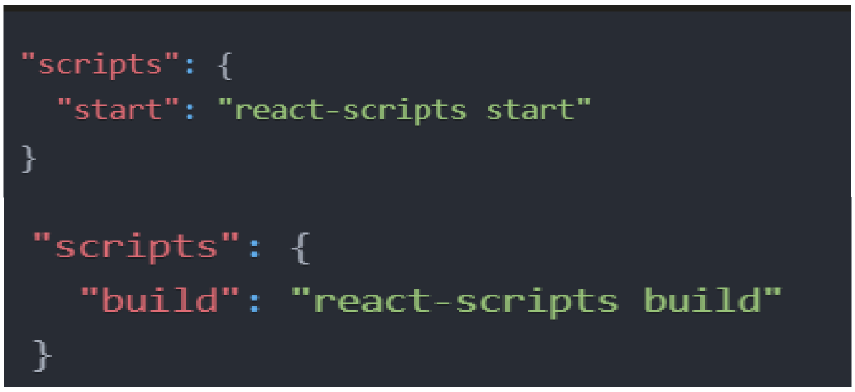
- For using **npm**, run "**npm init**" on your terminal
- For using **yarn**, run "**yarn init**" on your terminal.
- we can also run npm "**init -y**" or "**yarn init -y"** to generate the **package.json** file with default values.

npm scripts:

In the package.json file there is also a **scripts** property. npm scripts are custom commands defined in a project's package.json file that **automate** common development tasks and processes. This can be used to **run command line tools** that are installed within the **project's local context**. Common scripts we might use are things like:

- **npm test**—to run our tests
- **npm build**—to build our project
- **npm start**—to run our project locally

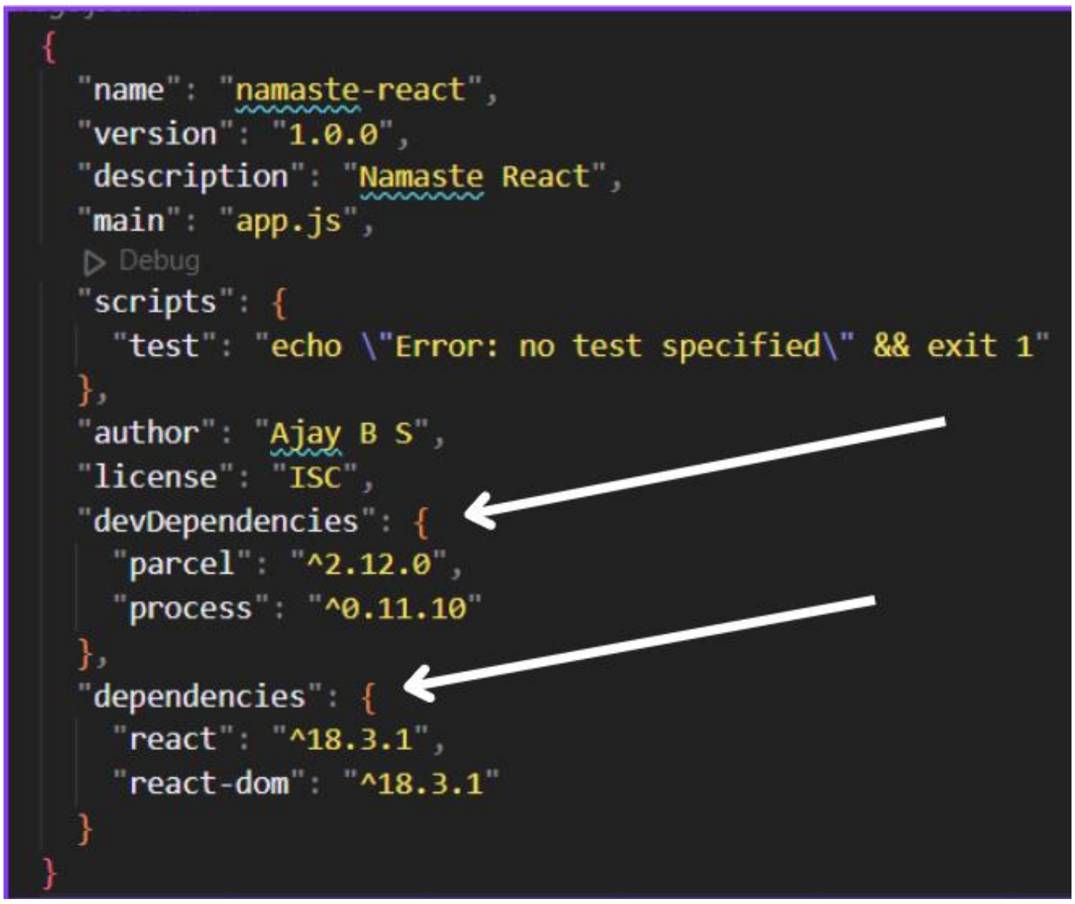
Of course, we are flexible to customize scripts that make sense for our specific project.



```
"scripts": {
  "start": "react-scripts start"
}
```

```
"scripts": {
  "build": "react-scripts build"
}
```

Dependencies vs. devDependencies-:



```
{
  "name": "namaste-react",
  "version": "1.0.0",
  "description": "Namaste React",
  "main": "app.js",
  ▷ Debug
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Ajay B S",
  "license": "ISC",
  "devDependencies": {
    "parcel": "^2.12.0",
    "process": "^0.11.10"
  },
  "dependencies": {
    "react": "^18.3.1",
    "react-dom": "^18.3.1"
  }
}
```

Dependencies are external **libraries**, **frameworks,** or **modules** that are required for our project to run. They provide pre-built functionality, saving our time and effort by allowing us to leverage existing code.

However, dependencies can be categorized into two main types:

1. **dependencies**
2. **dev dependencies.**

Dependencies: -

- Dependencies, also known as **runtime dependencies**, are the essential packages that our **application** or **project** relies on to run correctly in a **production environment**. They are necessary for the execution and functionality of our code. For example, if we are building a web application using a JavaScript framework like **React**, **react** itself would be a **runtime dependency**. These dependencies are usually specified in a **package.json** file or an equivalent configuration file in other programming languages.
- The purpose of **runtime dependencies** is to ensure that our application has all the required components to work as intended. When deploying our project to a **production environment** or sharing it with others, these dependencies need to be installed for the code to execute correctly**. Runtime dependencies** are typically included in the production build or deployed along with our application.

Following is the syntax for dependencies –

"npm install <package name>"

Dev Dependencies: -

- Dev dependencies, short for **development dependencies**, are packages that are not required for the actual execution of our code in a production environment. They are used during the **development process** and help developers with tasks such as **testing, linting, bundling, and building the project**. Dev dependencies are not bundled or shipped with our application when it is deployed or shared.
- Dev dependencies include tools like **testing frameworks** (e.g., **Jest**), build tools (e.g., **Babel**, **Webpack**), and other development-specific utilities. These packages assist developers in **writing high-quality code**, automating tasks, and ensuring the project's maintainability and reliability. Dev dependencies are typically installed during the **development phase** and are not necessary when running the project in a **production environment**.

Following is the syntax for Dev dependencies –

"npm install <package name> --save-dev"

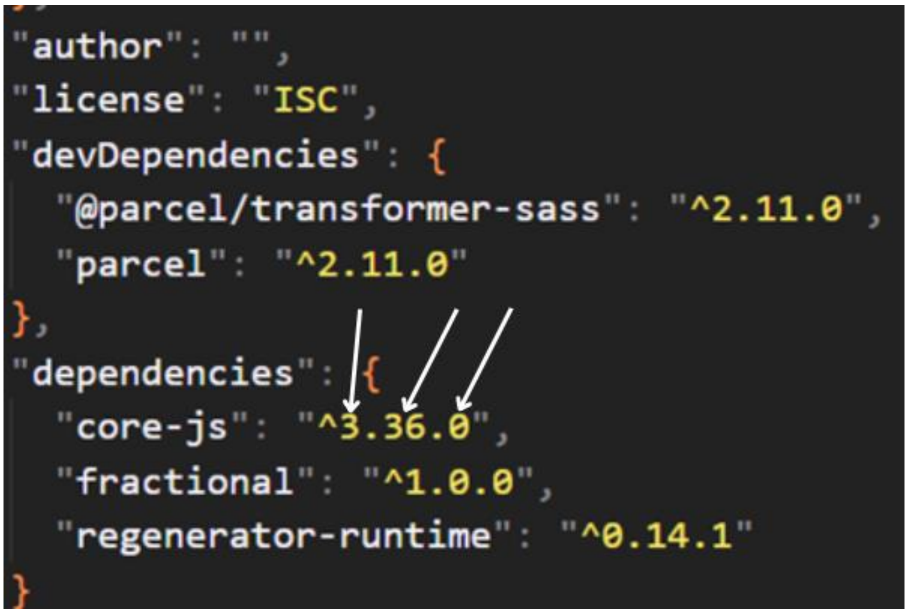
Or

"npm install -D <package name>"

What are Transitive dependencies:

Transitive dependency is a **dependency** of a **dependency**. If our React project depends on **Package A**, and **Package A** depends on **Package B**, then **Package B** is a transitive dependency for our project. React itself has several dependencies, and when we use additional libraries in our React project, each of those libraries may have its own dependencies. This creates a **chain** or **tree of dependencies.**

Semantic versioning: -



```
"author": "",
"license": "ISC",
"devDependencies": {
  "@parcel/transformer-sass": "^2.11.0",
  "parcel": "^2.11.0"
},
"dependencies": {
  "core-js": "^3.36.0",
  "fractional": "^1.0.0",
  "regenerator-runtime": "^0.14.1"
}
```

The versions above are in the format **MAJOR.MINOR. PATCH**. So, what does this mean for us?

- A **MAJOR** version involves breaking changes—likely we will need to update the package in our project when a major version has changed.

- A **MINOR** version change is backwards compatible, meaning it should update without breaking things.
- A **PATCH** version change is backwards compatible bug fixes, or other small fixes.

For example, if we have a project with the version **1.2.3** and we add a new feature without breaking any existing functionality, we can increase the **MINOR** part and make it **1.3.0.** If we fix a bug without changing the API, we can increase the **PATCH** part and make it **1.3.1**. If we change the API in a way that breaks existing code, we can increase the **MAJOR** part and make it **2.0.0**.

Semantic versioning helps us communicate the changes in our project and avoid compatibility issues with other projects that depend on ours.

Version ranges: -

- Npm versions are written in **0.0.0** formats, where first number (from left) stands for **major release,** second for the **minor release** and the third for the latest **patch release** of this particular version.
- When we specify the versions of our **dependencies** in the **package.json** file, we can use different symbols to indicate the range of acceptable versions. These symbols are:
  - **~ (tilde):** This means that we accept any **patch updates** within the same **minor** version. For example, **~1.4.0** means that we accept any version from **1.4.0** to **1.4.x**, but not **1.5.x** or higher.
  - **^ (caret):** This means that we accept any **minor** or **patch** updates within the same **major** version. For example, **^1.4.0** means that we accept any version from **1.4.0** to **1.x.x,** but not **2.x.x** or higher.
  - No symbol before the **version** means the version of the package must match exactly and should not be updated.

What is the purpose of package.json file?

Here are some of the main purposes of the package.json file: -

- **Dependency management**: The package.json file specifies the **dependencies** and **dev-dependencies** required by the project, as well as their version ranges**.**
- **Metadata**: The file contains **metadata** about the project, such as the **name**, **version**, **description**, **author**, **license**, and other relevant information.
- **Scripts**: It can define scripts that can be run using the **npm run command**, such as **build**, **test**, **start**, and others.
- **Configuration**: The package.json file can be used to configure various aspects of the project, such as the **main entry point**, the **repository**, the engines required to run the project, and more.
- **Publishing**: If the project is intended to be published to the npm registry, **the package.json** file provides information about how the package should be published, such as the **name**, **version**, **author**, **license**, and other relevant information.

What is package-lock. json?

In earlier versions of the **package.json** file did not provide a way to lock down the specific version of each dependency that a project was using. This meant that when a project was deployed or shared with others, there was a risk that different developers or machines would use different versions of the same dependency, which could cause compatibility issues or unexpected behavior.

- **package-lock.json** file is like a one-stop solution to our entire problem. **package-lock. json** is a file that is automatically generated by npm when a package is installed. **It records the exact version of every installed dependency**, including its **sub-dependencies** and their **versions**.
- The purpose of package-lock.json is to ensure that the same **dependencies** are installed consistently across **different environments**, such as **development** and **production environments**. It also helps to prevent issues with installing different package versions, which can lead to **conflicts** and **errors**.
- **package-lock. json** is created by npm when we run the **npm install** command. It contains a detailed list of all the packages, their **dependencies**, their **specific version numbers**.

If we are working in a team, it is important to commit **package-lock.json** to our **version control system** along with our code so that all team members have the same **dependencies** installed. When another developer **clones the project**, they can simply run **npm-install** to install the same packages and versions specified in the **package-lock.json** file.

This file allows **npm CLI(Command Line Interface)** to identify and start our project, run **scripts**, install **dependencies**, and **co-dependencies** and to publish the npm registry among many other tasks.

Additionally, **package-lock.json** has just transformed the way packages are installed in a program. Earlier, when these libraries and packages were installed from third-party vendors, tracing them back was a big problem.

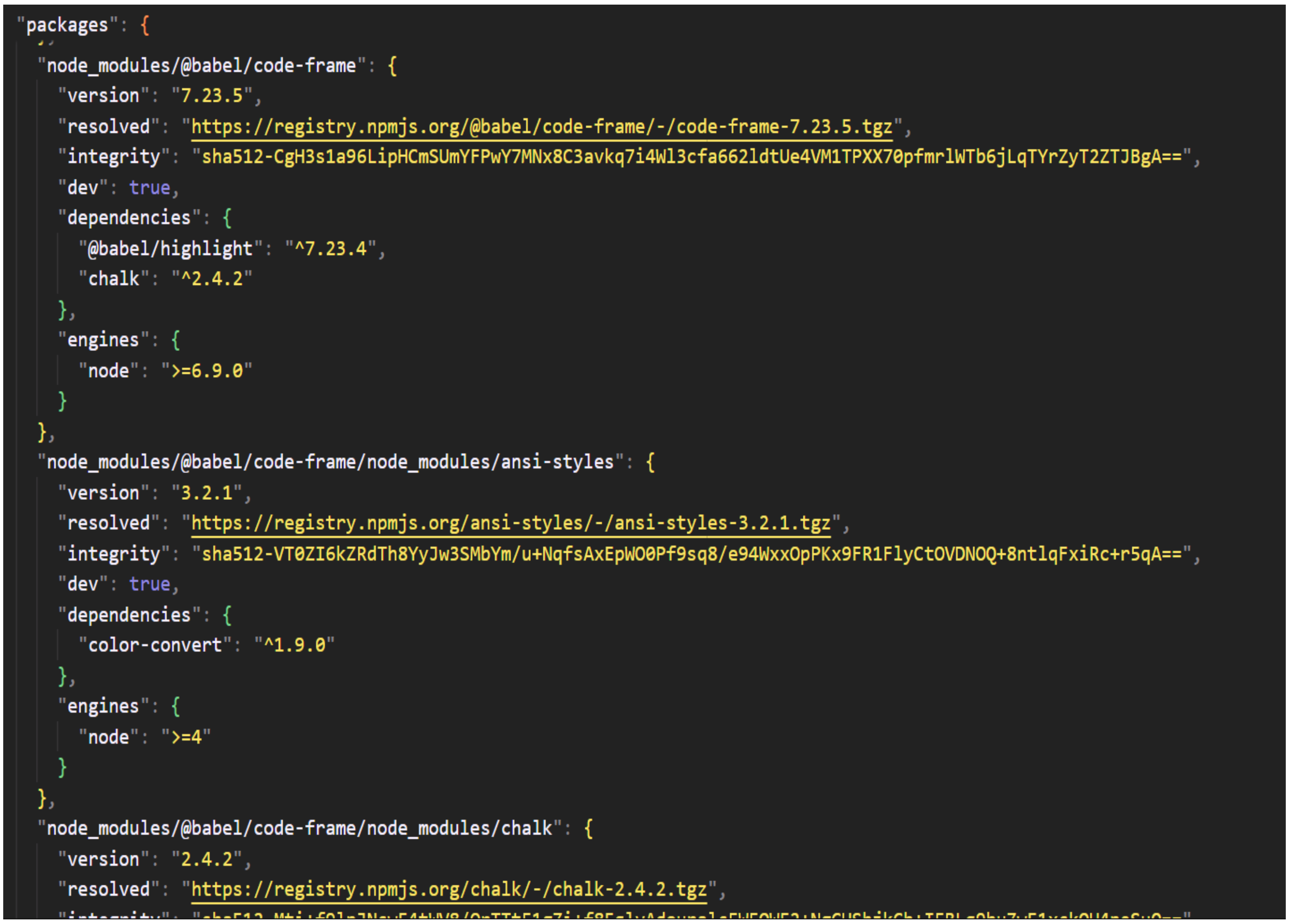
Now, with **npm registry** and the **package-lock.json** at hand, we no longer need to look for the source code of the **dependencies**. They are made available at the tip of our fingertips when all the dependencies are linked in the package.json file.

Also, the **carat (^)** sign just adds to the bounty! It just locks the dependency at the current version (the one we are creating the program in) and even helps in automatic **updating** when a newer version is released.

And optimize the installation process by allowing npm to skip repeated metadata resolutions for previously-installed packages.

Example: -

This is what a package-lock.json file looks like -



```
"packages": {
  "node_modules/@babel/code-frame": {
    "version": "7.23.5",
    "resolved": "https://registry.npmjs.org/@babel/code-frame/-/code-frame-7.23.5.tgz",
    "integrity": "sha512-CgH3s1a96LipHCmSUmYFPwY7MNx8C3avkq7i4Wl3cfa662ldtUe4VM1TPXX70pfmrlWTb6jLqTYrZyT2ZTJBgA==",
    "dev": true,
    "dependencies": {
      "@babel/highlight": "^7.23.4",
      "chalk": "^2.4.2"
    },
    "engines": {
      "node": ">=6.9.0"
    }
  },
  "node_modules/@babel/code-frame/node_modules/ansi-styles": {
    "version": "3.2.1",
    "resolved": "https://registry.npmjs.org/ansi-styles/-/ansi-styles-3.2.1.tgz",
    "integrity": "sha512-VT0ZI6kZRdTh8YyJw3SMbYm/u+NqfsAxEpWO0Pf9sq8/e94WxxOpPKx9FR1FlyCtOVDNOQ+8ntlqFxiRc+r5qA==",
    "dev": true,
    "dependencies": {
      "color-convert": "^1.9.0"
    },
    "engines": {
      "node": ">=4"
    }
  },
  "node_modules/@babel/code-frame/node_modules/chalk": {
    "version": "2.4.2",
    "resolved": "https://registry.npmjs.org/chalk/-/chalk-2.4.2.tgz",
```

You must have observed that the **package-lock.json** code is a bit longer than the **package.json** file.

Why is that so?

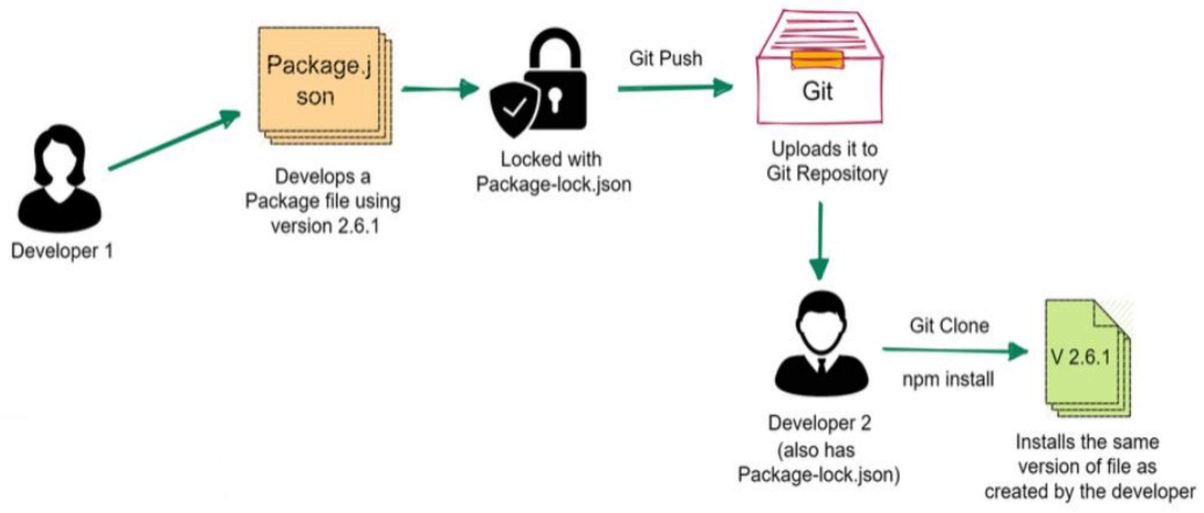
Well, because while **package.json** shows us a particular property and its version, **package-lock.json** would show the even the **dependency** of that particular property, the **sub-dependencies** etc. Thus, it locks the package data so deeply that we would not have to worry about any **dependencies** being missed out of sight.

What is the purpose of package-lock.json?

Here are some of the main purposes of the package-lock.json file:

- **Dependency version tracking**: The package-lock.json file keeps track of the exact **versions of dependencies** and **sub-dependencies** that are currently installed in a Node.js/JavaScript project.
- **Consistent builds**: Because the **package-lock.json** file **records specific versions of dependencies**, it ensures that everyone working on the project has the same dependencies installed, which helps to avoid version conflicts and ensures that the project can be built and run consistently across different environments.

- **Faster and more reliable installs**: The **package-lock.json** file allows **npm** to quickly and accurately **install the same versions of dependencies across different machines**, which makes the installation process **faster** and more **reliable**.
- **Security**: The **package-lock.json** file helps to prevent **malicious code** injection by ensuring that only verified and secure versions of dependencies are installed.
- **Reproducible builds**: By recording the specific versions of **dependencies** in the **package-lock.json** file, it makes it easier to reproduce builds at a later time or on a different machine, as we can simply use the same **package-lock.json** file to install the same **versions of dependencies**.


Developer 1
Package.json
Develops a Package file using version 2.6.1
Locked with Package-lock.json
Git Push
Git
Uploads it to Git Repository
Developer 2 (also has Package-lock.json)
Git Clone
npm install
V 2.6.1
Installs the same version of file as created by the developer

Comparing package.json and package-lock.json: -

package.json	package-lock.json
It is a metadata file that describes the project's dependencies, scripts, configuration, and other details.	It is a lockfile that provides an exact, deterministic list of all the installed packages and their dependencies, including their exact version numbers.
It is typically created and modified manually by the developer to manage the project's dependencies and configuration.	It is automatically generated by npm and updated whenever you install or update packages.
It lists the required dependencies and their version ranges, but not the exact versions to be installed.	It is used to ensure that the same dependencies are installed consistently across different environments and prevent conflicts due to different versions being installed.
It can be easily shared and committed to version control systems.	It is not meant to be manually modified and should be committed to the version control system to ensure consistency across all team members.

What is node_modules: -

The **node_modules directory is a folder** that contains all the **installed packages** (**dependencies**) required by our project. When we use **npm (Package Manager)** to install **packages** or **modules** for our project, these dependencies are typically stored in the **"node_modules"** directory.

"Node modules are very large, so we should always include them in the `.gitignore` file."
NOTE: Never touch **node_modules** and **package-lock. Json.**

NPX

What is NPX?

The npx stands for **Node Package Execute** and it comes with the npm, when we installed npm above **5.2.0 version** then automatically npx will be installed. It is a **npm package runner** that can execute any package that we want from the **npm registry without even installing that package**. The **npx** is useful during a **single time use package**. If we have installed npm **below 5.2.0** then **npx** is not installed in our system.

What is the advantage of NPX?

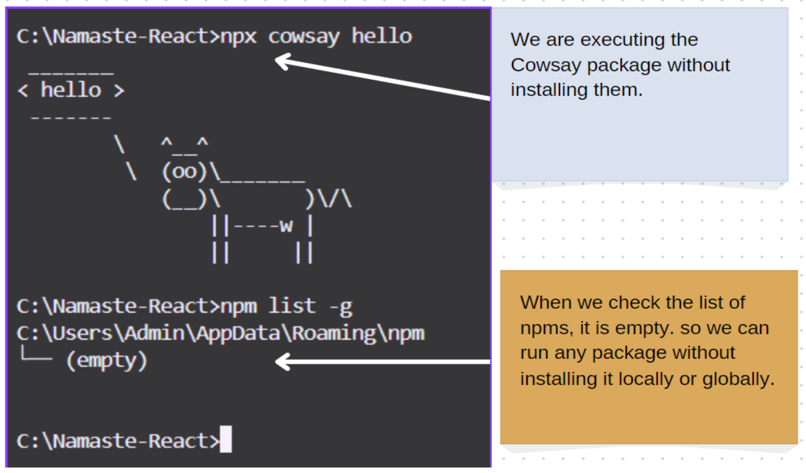
NPX allows us to **run** and **use packages** without having to install them **locally** or **globally**. If a package is installed while using NPX to run NPM executables, NPX will search for the package binaries (**locally** or **globally**) and then run the **package**.

What is the difference between NPX and NPM medium?

NPM is a package management that is used to install, uninstall, and update Javascript packages on our workstation, whereas **NPX** is a package executer that is used to **directly execute Javascript packages without installing them.**

Does NPX install locally?

No, **NPX** does not locally install software. It enables us to run packages straight from the **npm registry without installing them globally or locally**, maintaining a clean and segregated execution environment.



```
C:\Namaste-React>npx cowsay hello
 _______
< hello >
 -------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||

C:\Namaste-React>npm list -g
C:\Users\Admin\AppData\Roaming\npm
└── (empty)

C:\Namaste-React>
```

We are executing the Cowsay package without installing them.

When we check the list of npms, it is empty. so we can run any package without installing it locally or globally.

Bundler

What is a Bundler?

A bundler is a tool used in web development, particularly in modern JavaScript applications, to **bundle multiple JavaScript files** and **their dependencies into a single file** or a few files. This process is often referred to as "**bundling**" or "**code bundling**."

The primary purpose of a bundler is to improve the **performance and efficiency of web applications by reducing the number of requests required to load the necessary code**. Instead of having the browser load multiple individual files, a bundler combines them into one or a few larger files, which can be loaded more efficiently.

What do bundlers do?

- They allow developers to bundle their code, including **libraries** and **frameworks**, into a **single file**. This makes it easier to **deploy** and **run** the application, as well as making the code more **efficient** and **maintainable**.
- In other words, a **bundler is a tool that helps us to manage the dependencies and assets of our React application**, and **package them into a single file** (or a few files**) that can be served to the browser.**

Here are some key features and benefits of using a bundler:

- **Dependency Management**: Bundlers can analyze the **import statements** in our code and automatically include the required dependencies from **external libraries** or **modules**. This ensures that all the necessary code is included in the final bundle.
- **Code Splitting**: Some bundlers support **code splitting**, which allows them to **split** the **bundled code** into **smaller chunks** or **separate files**. This can improve the **initial load time** by only loading the essential code upfront, while **lazy-loading** other parts of the application as needed.
- **Transpilation and Polyfilling**: Bundlers can integrate with **transpilers** (like **Babel**) and **polyfills** to **convert** modern **JavaScript code** into a format that can be understood by **older browsers**, ensuring better **cross-browser compatibility**.
- **Minification and Optimization**: Bundlers can minify the bundled code by removing unnecessary **whitespace**, **comments**, and **renaming variables** to reduce the overall file size. They can also apply other **optimizations**, such as **dead code elimination** and **tree-shaking**, to **remove unused code**.
- **Source Maps**: Bundlers can generate source maps, which provide a mapping between the bundled code and the original source code, making it easier to **debug** the application in the browser's developer tools.
- **Hot Module Replacement (HMR):** Some bundlers support **Hot Module Replacement**, which allows for updating module files in the running application without a full page reload, improving the development experience.

How does a bundler work?

- **Entry Point:** The bundler starts from an **entry point,** which is typically the main file of our **application** or **library**. This entry point serves as the **starting point** for the bundler to discover and process all the required dependencies.
- **Dependency Resolution:** The bundler analyzes the **entry point** file and its **imports/requires** statements to identify and resolve the dependencies. It recursively follows these dependencies to build a **dependency graph**, representing the relationships between the files and modules in our project.
- **Code Transformation**: Once the **dependency graph** is built, the bundler can apply various transformations to the code. This might include **transpiling** modern JavaScript code (e.g., ES6+) to a format compatible with **older browsers** using tools like **Babel**. It may also involve processing **CSS**, **HTML**, or other asset types through loaders or plugins.
- **Module Resolution:** The bundler resolves the **modules,** and their **dependencies** based on the configured resolution rules. This typically involves searching for modules in the local **node_modules directories**, as well as any specified external paths or aliases.
- **Code Splitting:** Some bundlers support **code splitting**, which involves splitting the bundled code into smaller chunks or separate files. This can improve the **initial load time** by loading only the essential code upfront and **lazy-loading** other parts of the application as needed.
- **Minification and Optimization:** The **bundler** can minify the bundled code by removing unnecessary **whitespace**, **comments**, and **renaming variables** to **reduce** the overall **file size**. It can also apply other **optimizations**, such as dead code elimination and **tree-shaking**, to **remove unused code.**
- **Bundling**: After all the **transformations** and **optimizations**, the **bundler** generates one or more output files, depending on the configuration. These output files contain the bundled code, including the entry point file and all its resolved dependencies.
- **Source Maps:** The **bundler** can generate **source maps**, which provide a mapping between the bundled code and the original source code, making it easier to debug the application in the browser's developer tools.
- **Output**: The **bundler outputs** the **bundled files**, which can be served to the **client** (e.g., a **web browser**) or used as input for further processing or deployment.

Parcel

What is Parcel?

Parcel is a popular **open-source JavaScript bundler** for web applications. A bundler is a tool that takes various pieces of **code**, **assets**, and **dependencies** from a **web application** and **bundles** them together into a **format** that is **optimized forward deployment on the web**.

Parcel is known for its simplicity and ease of use. Some key features and aspects of Parcel include:

- **Zero Configuration**: One of the standout features of Parcel is its **zero-configuration** setup. In many cases, we can start using Parcel without having to write **complex configuration files**. This makes it easy for **developers** to get started quickly without the need for extensive setup.
- **Blazing Fast Bundling:** Parcel claims to be faster than other bundlers due to its **parallel processing** and **caching mechanisms**, which can significantly speed up the **bundling process**, especially for larger projects.
- **Automatic File Transformations:** Parcel can automatically transform various file types, such as **transpiling modern JavaScript** (**ES6+**) to a **compatible format**, compiling **Sass/LESS** to **CSS**, and **optimizing images** and other **assets**.
- **Automatic Code Splitting:** Parcel automatically splits the bundled code into **smaller chunks**, enabling **lazy-loading** and **improving** the **initial load time** of our **application**.
- **Hot Module Replacement (HMR):** Parcel provides **Hot Module Replacement**, allowing developers to **see changes in real-time without needing to manually refresh the entire page.** This feature enhances the development experience by providing instant feedback during the coding process.
- **Tree-Shaking and Minification:** Parcel performs **tree-shaking** and **minification** out of the box, **removing unused code and minimizing the bundle size for production builds**.
- **Out-of-the-box Support for Multiple Languages:** Parcel can handle a variety of languages and file types right out of the box, including JavaScript, TypeScript, CSS, HTML, images, and more.

Difference between live loading and Hot Module Replacement (HMR): -

"Live loading" and **"hot reloading**" are terms often used in the context of web development to describe different approaches to updating and **reflecting code changes in real-time.** While both aim to improve the development experience, there are distinctions between the two:

Live Loading:

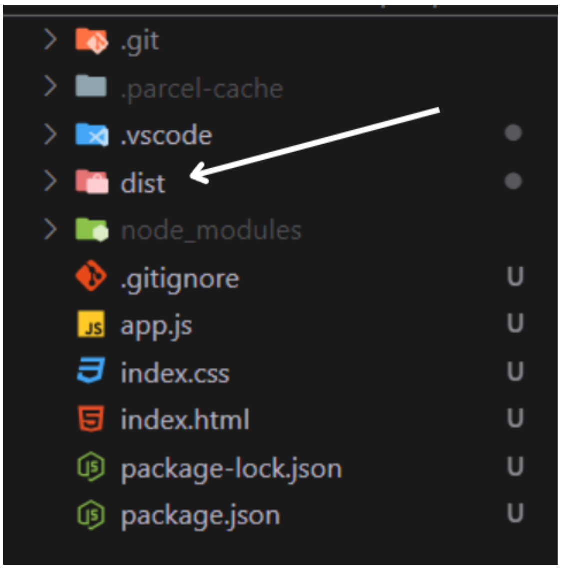
- **Definition**: Live loading, sometimes referred to as "**live reloading**" or "**auto-reloading**," involves automatically refreshing the entire page or application whenever there is a code change.

- **Process**: When we make changes to **our source code,** the **entire webpage or application is reloaded in the browser**. This ensures that the latest code modifications take effect.
- **State Preservation**: Live loading typically does not **preserve the state of the application**, meaning that if we were in a specific part of our application, such as a **form** or a particular view, the **entire page is reloaded**, and we might lose that state.

Hot Module Replacement:

- **Definition:** Hot Module Replacement, on the other hand, is a **technique that updates the application in real-time without requiring a full-page refresh**.
- **Process:** When we make changes, only the specific **module** or **component** being modified is updated, and the application attempts **to keep its current state intact**. This allows developers to see the impact of their changes instantly, **without disrupting the entire application.**
- **State Preservation:** Hot reloading aims to **preserve the state** of the application during code changes, making it more seamless and less disruptive than a **full live reload**.
- There is **File Watcher Algorithm** (written in **C++**). It keeps track of all the files which are changing real-time and it tells the **server** to **reload**. These are all done by **Bundler**.

What is the `dist` folder?



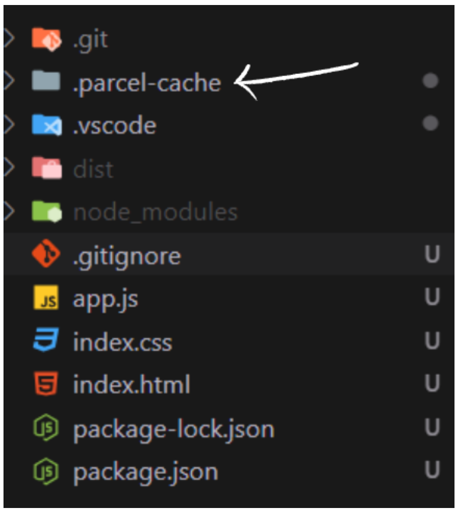
The **dist folder** in React (and many other **JavaScript projects**) stands for "**distribution**". It contains the **production-ready**, **minified**, and **optimized version** of our application that's ready to be **deployed** to a **web server**. Here's why it's important:

- **Optimized code**: The files in **dist** are typically **minified** and **bundled**, **reducing** file size and improving **load times**.
- **Browser compatibility**: The code may be **transpiled** to ensure compatibility with a wider range of **browsers**.

- **Asset management**: It often includes processed assets like **compressed images** or **compiled CSS**.
- **Separation of concerns**: It keeps our **source code** separate from the **production-ready code**.
- **Easier deployment**: we can simply upload the contents of the **dist folder** to our **web server**.

The **dist folder** is usually created by **build tools** like **Webpack**, **Parcel**, or **Create React App's** build process. **we don't typically work directly in this folder**; instead, we develop in our source folders and use build commands to generate the **dist contents**.

What is parcel-cache?

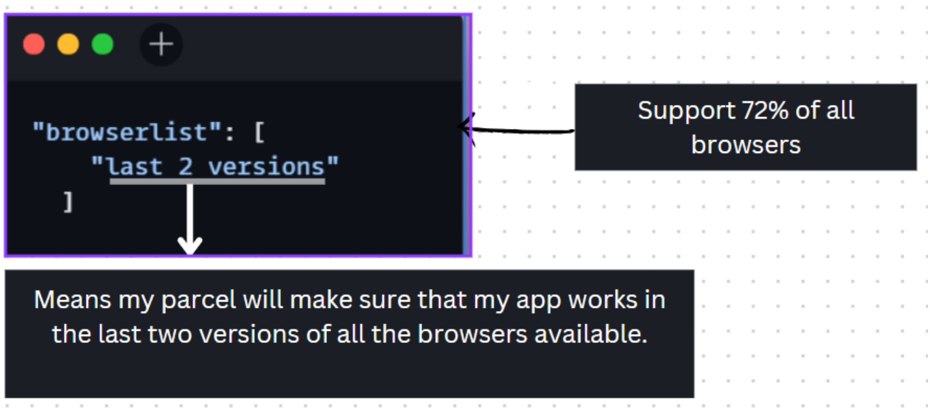

```
.git
.parcel-cache
.vscode
dist
node_modules
.gitignore          U
app.js              U
index.css           U
index.html          U
package-lock.json   U
package.json        U
```

The **parcel-cache** directory in Parcel is a **storage location** used by the **Parcel bundler** to **cache** (**cache is temporary storage for quick access to frequently used data**), intermediate build results and **metadata**. This **caching mechanism** improves **build performance** by avoiding redundant work, allowing Parcel to quickly rebuild only the parts of the project that have changed. The **parcel-cache** directory is automatically managed by **Parcel** and should typically be included in the **.gitignore** file to prevent it from being committed to **version control**. **Parcel** gives faster build, faster developer experience because of **caching**.

Babel: -

Babel is a **JavaScript compiler** that takes our modern JavaScript code and returns **browser compatible JavaScript**. It translates JSX into **React.createElement**. This is one of the reasons why setting up a React environment requires a bit of configuration.

What is Browserslist?



```
"browserlist": [
    "last 2 versions"
  ]
```

Support 72% of all browsers
Means my parcel will make sure that my app works in the last two versions of all the browsers available.

The **Browserslist** package is used to define the list of **target browsers** and **their versions** that our project should support. It's a configuration file that determines which **browser versions** our **application** needs to be compatible with for **CSS** and **JavaScript transpilation** and **prefixing**.
It makes our code compatible for a lot of **browsers**.