**Igniting Our App**

**Q1. What is ‘NPM’?**

**Answer**: NPM stands for Node Package Manager. It is a library and registry of various JavaScript software packages. Basically, it’s a tool that helps you add and manage packages in the project. For example, when connecting to a database I can use NPM to download the necessary code that does that for me. It also keeps track of the code to make sure everything is working properly.

**Q2. What is ‘Parcel/Webpack’? Why do we need it?**

**Answer**: Parcel/Webpack is a tool called “bundlers” which is usually used in web development. Their main task is to take all the code files and packages and bundle them together into a smaller, optimized package. This package is then sent to the browser so that the browser can load our app or website faster. Parcel and Webpack can make our code smaller and faster by removing any unnecessary parts and this process is called minification. Not all JavaScript code works in all the browsers, but bundler can help translate my code to work on the older versions of the browser.

**Key Features of Parcel**:

1. Zero Configuration:
   * Parcel is good because it works out of the box with little to no configuration. A developer doesn’t need to set up a complex configuration file like Webpack’s webpack.config.js
   * This makes it easy to start quickly, especially for small projects.
2. Automatic Dependency Management:

**Q3. What is `.parcel-cache`?**

**Answer**: It’s a folder created by parcel to store temporary files that are used to speed up the build process. Since bundling takes a lot of work like compiling and optimizing files and code, instead of redoing it again after every change it saves some of the processed data in the .parcel-cache folder. This way, it can reuse the data the next time we build the project, fasting the process. This also helps remember Parcel which files have been optimized and processed so that it only needs to work on the files that have changed. Most of the time, we don’t need to touch this file unless we have some build issues, deleting this .parcel-cache file can help resolve things.

**Q4. What is `npx`?**

**Answer**: npx is a tool that lets you run a program without installing it. Usually when we want to use a tool, we have to install it, but npx can execute that tool right away for that moment without having to keep it installed in your system. This is very useful when you don’t need the tool and only want to use it once. It also makes sure that you’re using the latest version of the tool each time you run it.

**Q5. What is difference between `dependencies` vs `devDependencies`?**

**Answer**:

* **dependencies**: These are the packages that go into production. In other words, these are the files that our project will need to run when it is in a production environment. So, for example, if I want to include a pdf viewer library to show pdf files then that library would be included in dependencies. So, when someone installs our web app or clones our repo, they will install these packages ensuring that the application has everything it needs to run.
* **devDependencies**: These are the packages that are only needed when we are developing and testing the application. They don’t go into the production environment. Tools like parcel for bundling code and babel for compiling modern JS code into a version compatible with older browsers are typically listed under devDependencies.

**Q6. What is Tree shaking?**

**Answer**: Basically, Tree shaking is a concept in JavaScript specially in ES6 (EcmaScript6), where the bundler (Webpack, Parcel, or Rollup) will remove unused code either be it a function or a variable or modules from the final bundle. The term comes from the idea of “shaking” of a tree to drop the dead or unnecessary branches while keeping only the ones that are used.

Tree shaking works best with ES6 modules because they use static import/exports.

For example:

A screenshot of a computer

Description automatically generated

A black rectangular object with a grey stripe

Description automatically generated

With tree shaking, when you bundle your app using a bundler (Webpack, parcel or Rollup) the unused function will be “shaken out” (In other words – “deleted”) (function to be deleted: subtract, multiply and divide) and only the used function (In this case - add) will not be removed in the final bundle.

**Benefits of Tree Shaking**:

1. Smaller bundler size: unused code will be deleted and only used code will stay, hence reducing its the bundler size.
2. Faster Load Times: Smaller bundles load faster in the browser, leading to better performance.
3. Optimized Code: Helps keep the codebase clean by eliminating dead or unused code.

**Key points to remember**:

1. Tree shaking relies on the use of ES6 modules(import and export)
2. Bundlers like Webpack and Rollup support tree shaking by default if configured properly.
3. It works well with libraries that use ES6 module syntax, such as React, Lodash-es, etc.

**Q7. What is Hot Module Replacement?**

**Answer**: Hot module replacement replaces, adds, or removes modules while an application is running, without a full reload. This speeds up the development process in a few ways:

1. Retain application state which is lost during a full reload
2. Save valuable development time by only updating what’s changed
3. Immediately update the browser when modifications are made to CSS/JS in the source code.

How the HMR works?

1. The application asks the HMR runtime to check for updates
2. The runtime asynchronously downloads the updates and notifies the application
3. The application asks the runtime to apply the updates
4. The runtime synchronously applies to the updates.

HMR doesn’t reload the entire application. It only updates the specific parts that were modified.

Only module that contain HMR code will get updated. If a module doesn’t have HMR code, the update will “bubble up” to its parent module until a handler is found, or the app is fully reloaded.

HMR is important because we don’t need to rebuild or reload everything which speeds up the development process and saves time and preserves things like user input, form data or current app state.

**Q8. List down your favorite 5 superpowers of Parcel and describe any 3 of them in your own words.**

**Answer**:

**Hot Module Reload (HMR)**: It automatically updates the code when a user makes some changes without fully reloading the page.

**Minification**: Minification reduces the file size of output bundles by removing white space, removing unused variables, renaming variables to shorter names and much more.

**Image Optimization**: Parcel has built in support for image resizing, converting and optimizing. Images can be referenced from CSS, HTML, JS or any other file types.

**Code Splitting**: Code splitting in parcel means to break a large code into smaller, more manageable chunks. This means that instead of loading the entire application at once, Parcel only loads the parts of the code that are needed when the user interacts with a specific feature.

**Q9**. What is ‘*.gitignore*’? What should we add and not add into it?

**Answer**: .gitignore is a file which specifies the files that are unnecessary to add to the git repository. Basically, it ignores the files which have been specified in the .gitignore file.

The .gitignore file itself is a plain text document. Here’s an example .gitignore file:

A white screen with red text

Description automatically generated

* \* is used as a wildcard match. So \*.exe will ignore any file with .exe extension
* / will ignore directories with the name. vendor/ will ignore the whole vendor directory.
* # used for comment
* […] will ignore values with any of the values.
  + \*.[abc] ignores files file.a, file.b, file.c
  + \*.[a-\*.[oa]d] the dash will include a range, in this case, file extension a-d.

We should use gitignore to:

* Ignore files that contain sensitive data
* The files which are system specific and shouldn’t go to a git repository
* Node modules are very heavy so we should add node modules to .gitignore.
* Any file that is associated with vscode workspace.

Remember we always ignore node modules and package-lock.json. These are sensitive files, and we should not touch them.

How to create the global .gitignore for your system – below are the steps:

* Create the file:
  + touch ~/.gitignore\_global
* Add the file to the Git configuration:
  + git config --global core.excludesfile ~/.gitignore\_global
* Edit the file with your text editor and add your rules to it.
* To see all ignored files use this command in the terminal: *git status -ignored*

**Q10**. What is the difference between ‘package.json’ and ‘package-lock.json’

**Answer**: **package.json**: package.json file contains essentials information like project name, author, license, description, version number, etc. It also specifies the project’s dependencies and devDependencies, often using versions ranges (e.g. ^1.0.0) to allow flexibility in updating to compatible versions. This file is maintained by developers and is used by npm to identify what packages are required to run or build the project.

**package-lock.json**: package-lock.json file ensures consistency and reproducibility across different environments. It records the exact versions of all installed dependencies and their nested dependencies. package-lock.json defines the “how” by locking down the exact package tree. This guarantees that anyone installing the project gets the same dependency versions, preventing potential issues from unintentional updates or mismatches in nested dependencies.

Here’s a detailed explanation of the differences between package.json and package-lock.json, focusing on their roles, functionality, and importance:

**1. What Are They?**

**package.json**

* The **manifest file** of a Node.js project.
* Contains high-level metadata such as:
  + Project name, version, description, and author details.
  + Scripts to automate tasks (e.g., npm start, npm test).
  + Dependencies (dependencies and devDependencies) required to run and develop the project.
  + Version constraints for dependencies, like:
    - ^1.0.0 – Install compatible versions with minor updates.
    - ~1.0.0 – Install compatible versions with patch updates.
    - 1.0.0 – Lock to exactly version 1.0.0.

**package-lock.json**

* An **automatically generated file** that records the exact versions of dependencies and sub-dependencies installed.
* Contains a **snapshot of the dependency tree** with all packages resolved to their specific versions.
* Ensures **deterministic builds** by locking the dependency versions used.

**2. How They Work**

**package.json**

* You define dependencies and their version ranges here.
* When you run npm install, it fetches the latest compatible versions based on these constraints.
* Example:
* {
* "dependencies": {
* "express": "^4.17.0"
* }
* }

This tells npm to fetch any version of express that is >=4.17.0 and <5.0.0.

**package-lock.json**

* When you install packages, npm creates or updates package-lock.json to include the **exact versions** that were installed.
* It also records the dependency hierarchy, ensuring consistency across environments.
* Example (snippet):
* {
* "name": "my-app",
* "version": "1.0.0",
* "lockfileVersion": 2,
* "dependencies": {
* "express": {
* "version": "4.17.3",
* "resolved": "https://registry.npmjs.org/express/-/express-4.17.3.tgz",
* "integrity": "sha512-..."
* }
* }
* }

**3. Key Differences**

| **Feature** | **package.json** | **package-lock.json** |
| --- | --- | --- |
| **Purpose** | Project metadata and dependency declaration. | Exact dependency resolution for reproducibility. |
| **Version Flexibility** | Allows version ranges for dependencies (e.g., ^1.0.0). | Locks exact versions of dependencies. |
| **Manual vs Auto-Generated** | Manually created and edited by developers. | Automatically created and updated by npm. |
| **Scope** | Tracks direct dependencies only. | Tracks both direct and nested dependencies. |
| **Reproducibility** | Cannot ensure the same versions are installed everywhere. | Ensures exact versions are used across systems. |

**4. Why Both Files Are Needed**

* **package.json**: Acts as a blueprint for the project, defining what dependencies are needed.
* **package-lock.json**: Serves as the implementation details, specifying exactly how the blueprint is executed.

For example:

* In package.json, you might specify:
* "lodash": "^4.17.0"

This means "any compatible version of lodash starting from 4.17.0 to before 5.0.0."

* In package-lock.json, it will lock the version to something specific like:
* "lodash": {
* "version": "4.17.21",
* "resolved": "https://registry.npmjs.org/lodash/-/lodash-4.17.21.tgz"
* }

This ensures that even if a newer version of lodash (e.g., 4.18.0) is released, your project will continue using version 4.17.21 unless explicitly updated.

**5. Benefits**

**package.json**

* Easy to read and maintain.
* Lets developers define the dependency ranges and the scripts to run.
* Supports cross-platform usage with scripts and metadata.

**package-lock.json**

* Guarantees identical dependency trees in all environments.
* Improves install speed by caching resolved versions.
* Helps diagnose issues by locking dependencies at specific versions.

**6. Common Scenarios**

1. **Installing Dependencies**  
   When you run npm install, npm checks package.json for the dependencies and installs them:
   * If package-lock.json exists, npm uses the exact versions specified in it.
   * If it doesn’t exist, npm resolves the dependencies from package.json and creates a new package-lock.json.
2. **Sharing Projects**
   * If a new developer clones the project and runs npm install, the exact same dependency versions (from package-lock.json) will be installed.
3. **Updating Dependencies**
   * You can update a package by editing package.json or running commands like npm update.
   * This updates package-lock.json with the new version while maintaining its lock functionality.

**7. Key Notes**

* **Should You Commit package-lock.json?** Yes, always commit package-lock.json to version control. It ensures everyone working on the project uses the same dependency versions.
* **What Happens Without package-lock.json?** Dependency resolution might differ across environments, leading to potential bugs or mismatches.
* **When Should You Edit Them?**
  + Edit package.json when you need to add, remove, or change dependencies.
  + Do not manually edit package-lock.json; let npm handle it.

Q11. Why should I not modify ‘package-lock.json’?

Answer:

* The package-lock.json file is automatically generated by the npm when you install or update dependencies in your project.
* It locks down the specific versions of every package and sub-package (transitive dependency) that are installed, ensuring that every developer or CI/CD pipeline working on the project uses the same versions of dependencies.
* Modifying package-lock.json manually can break this consistency. When dependencies change, it’s the job of the package manager to update the lock file properly. By modifying it directly, you risk introducing inconsistencies or conflicts that could make it harder for other developers (or automation systems) to install the correct versions of dependencies.
* Since package-lock.json contains a detailed tree of all dependencies and their versions, modifying it manually might lead to a situation where a specific package version or its sub-dependencies are broken or incompatible with your project, without any visibility into why or how it happened.
* The package manager is directly to handle updates to package-lock.json. When you add, update or remove dependencies using commands like npm install, npm update, or npm uninstall, npm ensures that package-lock.json reflects the changes accurately. If you modify it manually, npm might not be aware of the changes or could even overwrite them when you run commands that update the lock file.
* If you encounter merge conflicts in package-lock.json, you can use the npm ci command to regenerate the lock file based on the package.json, or resolve the conflict manually, but always let npm handle the process of managing the file.

Q12. What is ‘node\_modules’? Is it a good idea to push that on git?

Answer: node\_modules is a folder in your Node.js project where all the installed dependencies (i.e. libraries or packages) are stored. When you run npm install, npm downloads and saves all the required dependencies for your project into this folder.

For example, if your project depends on packages like express, lodash, or react, those libraries will be downloaded and placed inside the node\_modules folder.

node\_modules can be huge – sometimes hundreds of megabytes or more – and adding this folder to git repositories will make the project very large unnecessary. This can slow down the process of cloning of the repositories, and it makes your version control history unnecessarily bloated.

Some dependencies might include platform-specific binaries (e.g., compiled code for macOS vs. Windows). If you push node\_modules to Git, someone on a different platform might face issues when trying to install or run the project.

You don’t need to store node\_modules in Git because it can always be recreated by running npm install (or yarn install). The package.json and package-lock.json files contain all the information about what dependencies your project needs. These files allow anyone to re-install the correct dependencies without needing to push them to Git.

The best practice is to ignore the node\_modules folder by adding it to a special file called .gitignore. This tells Git to not track changes to node\_modules, keeping your repository clean.

To do this, simply add the following line to your .gitignore file:

node\_modules/

**Q13. What is ‘dist’ folder?**

**Answer**: dist stands for distributable and this folder refers to a folder in the directory where all the build files are located and can be usually used by others without the need to compile or minify the source code. Usually, the minified and obfuscated code goes into dist folder so that when someone else wants to use your project they don’t need to minify your code again. It’s the folder we get after we modify our source code and run “npm run build” or “ng build” or “ng build --prod” for production.

Just for more clarity I will provide a point base note below:

* The /dist stands for distributable
* The /dist folder contains the minified version of the source code.(Note: Minified and Minimized are the same thing, referring to the process of removing unnecessary characters from source code without changing its functionality. This process is also known as code minification).
* The code present in dist folder is the code used in production of web applications.
* The /dist folder also comprises of all the compiled modules that may or may not be used with other systems.
* It is easier to add files in the dist folder as it is an automatic process. All the files are automatically copied to the dist folder on save.
* The /dist folder also includes files that are used to run/build a module in other platforms.

To give an example: The source code of any program or library is present in the /src directory. Now, if one wishes to use the source code of a certain library(C, C++, Java, etc.) which was written by another person, then they need to compile the source code first before being able to use it. If this source code does not comply then it would not be possible to use them. However, if somehow, a precompiled version of the source code is already available, then one does not need to go through the task of compiling the source code files and it can be directly used. Such an already compiled version is saved into the /dist directory.

Likewise, if one wishes to share a JavaScript library, one should add the original (not minified) source code into the src/ folder and the minified (precompiled) version into the dist folder. By doing so, anyone can use the minified version of the code right away without having to minify it themselves.

Q14. What is `browserlists`?

Answer: