1. **What is `NPM`?**

* npm is the world's largest software registry.
* npm consists of three distinct components:
* the website - Use to discover packages, set up profiles, and manage other aspects of your npm experience.
* the Command Line Interface (CLI) - runs from a terminal, and is how most developers interact with npm.
* the registry - is a large public database of JavaScript software and the meta-information surrounding it.

1. **What is `Parcel/Webpack`? Why do we need it?**

* A module bundler is a tool that takes pieces of JavaScript and their dependencies and bundles them into a single file, usually for use in the browser.
* Parcel is a web application bundler, differentiated by its developer experience.
* We need parcel for building an optimized, minified and bundled production apps.
* We also need them for image compression, old browser support, cleaning up of code, caching and creating servers.

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

* The .cache folder (or .parcel-cache in parcel v2) stores information about your project when parcel builds it, so that when it rebuilds, it doesn't have to re-parse and re-analyze everything from scratch. It's a key reason why parcel can be so fast in development mode.

1. **What is `npx` ?**

* It is an npm package runner that can execute any package that you want from the npm registry without even installing that package.
* The npx is useful during a single time use package.
* Npx is a tool that use to execute packages

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

* Dev dependencies are modules which are only required during development whereas dependencies are required at runtime. If you are deploying your application, dependencies has to be installed, or else your app simply will not work. Libraries that you call from your code that enables the program to run can be considered as a dependencies.

Eg- React , React - dom

* Dev dependency modules need not be installed in the production server since you are not going to develop in that machine .compilers that covert your code to javascript , test frameworks and document generators can be considered as dev-dependencies since they are only required during development .

Eg- ESLint , Babel , webpack

1. **What is Tree Shaking?**

* When we import and export modules in JavaScript, most of the time there is unused code floating around. [Tree shaking or dead code elimination](https://webpack.js.org/guides/tree-shaking/) means that unused modules will not be included in the bundle during the build process.
* Tools like [webpack](https://webpack.js.org/" \t "_blank) will detect dead code and mark it as “unused module” but it won’t remove the code. Webpack relies on minifiers to cleanup dead code, one of them is [UglifyJS plugin](https://webpack.js.org/plugins/uglifyjs-webpack-plugin/" \t "_blank), which will eliminate the dead code from the bundle.
* Utilizing the tree shaking and dead code elimination can significantly reduce the code size we have in our application. The less code we send over the wire the more performant the application will be.

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

* As you make changes to your code, Parcel automatically rebuilds the changed files and updates your app in the browser.
* By default, Parcel fully reloads the page, but in some cases it may perform Hot Module Replacement (HMR).
* HMR improves the development experience by updating modules in the browser at runtime without needing a whole page refresh.
* This means that application state can be retained as you change small things in your code.
* CSS changes are automatically applied via HMR with no page reload necessary.
* HMR works by replacing the code for a module, and then re-evaluating it and along with all of its parents.

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

* 5 superpowers of Parcel –
  1. Hot Module Replacement
  2. File Watcher Algorithm
  3. Caching
  4. HTTPS
  5. Asset Compression and Optimization, Bundling, Minification
* File Watcher –

1. To support an optimal caching and development experience, Parcel utilizes a very fast watcher written in C++ that integrates with low-level file watching functionality of each operating system.
2. Using this watcher Parcel watches every file in your project root (including all node\_modules).
3. Based on events and metadata from these files, Parcel determines which files need to be rebuilt.

* Caching –

1. Parcel caches everything it builds to disk.
2. If you restart the dev server, Parcel will only rebuild files that have changed since the last time it ran.
3. 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.

* Image Optimization –

1. Parcel supports resizing, converting, and optimizing images. You can use query parameters when referencing an image in HTML, CSS, or JavaScript to specify which format and size the image should be converted to.
2. You can request multiple sizes or formats from the same source image, which helps support different types of devices or browsers efficiently.
3. Resizing and converting images occurs both in development and production mode, so you can test with the correct image dimensions and formats as well.
4. Parcel also includes lossless image optimization for JPEGs and PNGs by default in production mode, which reduces the size of images without affecting their quality.
5. **What is `.gitignore`? What should we add and not add into it?**

* When you’re working in your copy, Git [watches every file](https://www.atlassian.com/git/tutorials/saving-changes/gitignore) in and considers it in three ways:
* Tracked: You’ve already staged or committed the file.
* Untracked: You’ve not staged or committed.
* Ignored: You’ve explicitly told Git to ignore the file(s).
* The .gitignore file tells Git which [files to ignore](https://github.com/github/gitignore) when committing your project to the GitHub repository.
* We can add system specific files, vscode workspaces, security and API keys/secrets.
* dependency caches, such as the contents of /node\_modules or /packages
* compiled code, such as .o, .pyc, and .class files
* build output directories, such as /bin, /out, or /target
* files generated at runtime, such as .log, .lock, or .tmp
* hidden system files, such as .DS\_Store or Thumbs.db
* personal IDE config files, such as .idea/workspace.xml

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

| package.json | package.lock.json |
| --- | --- |
| It contains basic information about the project. | It describes the exact tree that was generated to allow subsequent installs to have the identical tree. |
| It is mandatory for every project. | It is automatically generated for those operations where npm modifies either node\_modules tree or package.json. |
| It records important metadata about the project. | It allows future devs to install the same dependencies in the project. |
| It contains information such as name, description, author, script, and dependencies. | It contains the name, dependencies, and locked version of the project. |

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

* This file is automatically created and used by npm to keep track of your package installations and to better manage the state and history of your project’s dependencies. You shouldn’t alter the contents of this file.
* One important thing to mention as well is the security improvement that comes with the package-lock file.
* Since it keeps all the hashes of the packages if someone would tamper with the public npm registry and change the source code of a package without even changing the version of the package itself it would be detected by the package-lock file.
* package.json contains only your direct dependencies, not the dependencies of your dependencies (sometimes called nested dependencies). This means with the standard package.json you can't control the versions of those nested dependencies, referencing them directly
* The solution to all this is the lock file which as described above locks in the versions of the full dependency tree. This allows you to guarantee your dependency tree for other developers or for releases whilst still allowing testing of new dependency versions (direct or indirect) using your standard package.json.

1. **What is `node\_modules` ? Is it a good idea to push that on git?**

* You can think of the *node\_modules* folder like a cache for the external modules that your project depends upon.
* When you npm install them, they are downloaded from the web and copied into the *node\_modules* folder and Node.js is trained to look for them there when you import them (without a specific path).
* This is because there's no reason to store copies of all your dependent modules in your own GitHub project.
* The exact version you were using is known and stored in your package.json or package-lock.json so at any time you or anyone else using your project can download your code and then fetch all the other dependent modules from their original source (including even the exact same versions you were using).
* So, there isn't any reason to store a separate duplicate copy of all those dependent modules in your own project. That would just be wasteful and would complicate upgrading to a newer version of all those dependent modules.

1. **What is the `dist` folder?**

* The /dist stands for distributable.
* The /dist folder contains the minimized version of the source code.
* The code present in the /dist folder is actually the code which is used on production web applications.
* Along with the minified code, 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 to 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 contains all those files which are required to run/build a module for use with other platforms- either directly in the browser, or in an AMD system (eg. require.js).

1. **What is `browserlists`?**

* Browserslist is a tool that allows specifying **which browsers** should be supported in your frontend app by specifying "queries" in a config file.
* It's used by frameworks/libraries such as React, Angular and Vue, but it's not limited to them.
* As javascript evolves, browsers won't support new features at the same pace, for instance not all browsers have built-in support for ES6 (aka ES2015).
* By using browserslist, transpilers/bundlers know what browsers you want to support, so they can "group" browsers in different categories and generate separate bundles, for example:
* Legacy Bundle: Contains polyfills, larger bundle size, compatible with old browsers without ES6 support.
* Modern Bundle: Smaller bundle size, optimized for modern browsers.

1. **Diff between tilde and caret.**

|  |  |
| --- | --- |
| Tilde(~) notation | Caret(^) notation |
| Used for Approximately equivalent to version. | Used for Compatible with version. |
| It will update you to all future patch versions, without incrementing the minor version. ~1.2.3 will use releases from 1.2.3 to <1.3. | It will update you to all future minor/patch versions, without incrementing the major version. ^2.3.4 will use releases from 2.3.4 to <3.0.0 |
| It gives you bug fix releases. | It gives you backwards-compatible new functionality as well. |
| It will update in decimals. | It will update to its latest version in numbers. |
| Not a default notation used by NPM. | Used by NPM as default notation. |
| Example: ~1.0.2 | Example: ^1.0.2 |