1. What is `NPM`?

NPM (Node Package Manager) is the default package manager for Node.js and the largest ecosystem of JavaScript libraries.

It provides:

- **Package installation** (e.g., React, Lodash)
- **Dependency management**
- **Versioning + semantic versioning support**
- **Scripts runner** (`npm start`, `npm build`)
- **Registry** where developers publish and share JS packages

Short version:
NPM is the tool that installs, manages, and runs JavaScript packages for Node.js projects.

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

Parcel and Webpack are **bundlers** — tools that process your JavaScript, CSS, images, and other assets and bundle them into optimized files for the browser.

Why we need them:

- **Bundle multiple modules** into fewer files for faster loading
- **Transpile modern JS (ES6+)** to browser-compatible code
- **Handle assets** (CSS, images, fonts) with loaders/plugins
- **Tree shaking** to remove unused code
- **Hot Module Replacement (HMR)** for fast development
- **Minification & optimization** for production builds

Difference (high-level):

- **Webpack:** highly configurable, plugin-heavy, industry standard.
- **Parcel:** zero-config, faster setup, smart defaults, built-in optimizations.

Short version:
Parcel/Webpack bundle and optimize your code so it runs efficiently in browsers.

3. What is `.parcel-cache`?

`.parcel-cache` is a folder created by **Parcel** to store cached build artifacts.

Why it exists:

Parcel uses a smart caching system so it doesn't need to rebuild everything every time.
It stores:

- processed files
- transformed modules
- dependency graphs
- intermediate build outputs

Benefits:

- Much faster rebuilds
- Better incremental development performance

Deleting `.parcel-cache` forces Parcel to do a full rebuild.

Short version:
`.parcel-cache` is Parcel's build cache that speeds up subsequent builds**.**

4. What is `npx` ?

npx is a tool that comes with npm (since npm 5.2+) and is used to run packages without installing them globally.

Why it's useful:

- Executes a package directly from the npm registry or from `node_modules`.
- Avoids global installs → no version conflicts.
- Great for one-off commands.

Examples:

```
npx create-react-app my-app
```

Runs the latest `create-react-app` **without** installing it globally.

Short version:
`npx` lets you run npm packages instantly without installing them globally.

5. What is difference between `dependencies` vs `devDependencies`?

dependencies

- Packages your app *needs at runtime*.
- Included when you deploy/build for production.
- Examples: React, Axios, Lodash.

devDependencies

- Packages used *only during development*.
- NOT included in production builds.
- Examples: Webpack, Babel, ESLint, Jest, TypeScript.

Short version:
dependencies = required by the running app.
devDependencies = required only for development/build tooling.

6. What is Tree Shaking?

Tree shaking is a build optimization technique that removes **unused (dead) code** from your JavaScript bundles.

How it works:

- Relies on **ES modules (ESM)** because they allow static analysis.
- The bundler (Webpack, Rollup, Parcel) detects exports that are never imported/used.
- Those unused exports are removed during the production build.

Benefits:

- Smaller bundle size
- Faster load times
- Better performance

Short version:
Tree shaking = automatically removing unused code from the final bundle.

7. What is Hot Module Replacement?

Hot Module Replacement (HMR) is a development feature (in tools like Webpack, Vite, Parcel) that lets you **update modules in the browser without reloading the entire page**.

What it does:

- Replaces only the changed modules (JS/CSS/components).
- Keeps the app state intact (no full refresh).
- Makes development much faster.

Example:

If you update a React component, HMR updates just that component instead of reloading the whole app.

Short version:
HMR updates changed code instantly in the browser without a full reload, preserving state and speeding up development.

8. List down your favourite 5 superpowers of Parcel and describe any 3 of them in your own words?

Here are **5 Parcel superpowers** (staff-level, concise, interview-ready):

My Top 5 Parcel Superpowers

1. Zero Configuration Setup
2. Blazing-Fast Bundling with Multicore + Caching
3. HMR (Hot Module Replacement) out of the box
4. Automatic Code Splitting
5. Built-in Support for TypeScript, JSX, Sass, Images, etc. (no plugins needed)

Describe Any 3 (short, high-signal)

1. Zero Configuration Setup

Parcel works with almost no config — it auto-detects file types, sets up transforms, and creates an optimized build. This makes onboarding extremely fast and reduces time spent maintaining bundler config files.

2. Blazing-Fast Bundling with Smart Caching

Parcel parallelizes work across all CPU cores and uses a persistent `.parcel-cache` to avoid reprocessing unchanged files. This makes incremental builds dramatically faster, especially in large projects.

3. Automatic Code Splitting

Parcel automatically splits code into smaller chunks when it detects dynamic imports or separate entry points. This leads to smaller initial bundles and faster load times without manual configuration.

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

A **`.gitignore`** file tells Git which files or folders it should *not track* or commit to the repository.

✅ What we SHOULD add to `.gitignore`

These are files that are:

1. Auto-generated

- `dist/` or `build/`
- `.parcel-cache/`
- `coverage/`

2. Environment-specific or secret

- `.env`
- API keys, config files with credentials

3. Local machine or editor-specific

- `node_modules/`
- `.DS_Store`
- `.vscode/`
- `*.log`

4. System files

- OS temp files
- Swap files

❌ What we SHOULD NOT add to `.gitignore`

These are files required for teammates, CI/CD, or production builds:

1. Source code

- `src/`
- Components, logic, configs
  (These must be committed.)

2. Project configuration

- `package.json` / `package-lock.json`
- `.eslintrc`, `.prettierrc`
- `tsconfig.json`

3. Build-related scripts

- CI/CD files (`.github/workflows/*`)
- Deployment configs

Short version:

`.gitignore` tells Git what NOT to track. We ignore generated files, environment files, dependencies, and editor/system files — but we keep all source code and project configs in the repo.

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

package.json

- Defines your project's **metadata**, scripts, and dependency **ranges** (e.g., `^1.2.0`).
- Describes what your project *depends on*, but not the exact versions installed.

package-lock.json

- Records the **exact versions** of every installed dependency (and sub-dependency).
- Ensures reproducible installs across machines.
- Automatically generated; you shouldn't edit it manually.

Short version:

`package.json` = dependency *rules* + project info.
`package-lock.json` = exact dependency *versions* for consistent installs.

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

You should **not modify `package-lock.json` manually** because:

1. It ensures exact, reproducible installs

It locks the precise dependency graph. Manual edits can break determinism and cause inconsistent environments across machines or CI/CD.

2. It is auto-generated

The file is structured based on npm's internal logic. Editing it by hand can corrupt the file or cause npm to reinstall everything incorrectly.

3. It ensures security + dependency integrity

It stores integrity hashes. Editing it manually can break those hashes, causing install failures or security mismatches.

4. It can introduce subtle, hard-to-debug bugs

Manually tweaking versions can cause dependency mismatches that only show up for some developers or environments.

Short version:

`package-lock.json` guarantees consistent, secure, exact installs. Manual editing can corrupt the dependency graph and break the build.

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

`node_modules` is the folder where npm (or yarn/pnpm) installs all your project's dependencies and their sub-dependencies.

It contains:

- thousands of packages
- compiled binaries
- platform-specific builds

❌ Should you push it to Git?

No — never.

Why not:

1. **Huge size** → slows down cloning and repository storage.
2. **Machine-specific files** → contains OS/architecture-specific builds.
3. **Reproducibility is already guaranteed** by `package-lock.json` or `yarn.lock`.

4. **npm can regenerate it** using `npm install`.

Short version:

`node_modules` is where all dependencies live. It should *not* be pushed to Git because it's large, machine-specific, and fully rebuildable from the lock file.

13. What is the `dist` folder?

The **`dist`** folder (short for *distribution*) is the output directory generated by your bundler (Parcel, Webpack, Vite, etc.) when you build your project for production.

It typically contains:

- **Minified JavaScript bundles**
- **Optimized CSS files**
- **Images, fonts, and static assets**
- **HTML files prepared for deployment**
- **Source maps (optional)**

These files are:

- optimized
- compressed
- ready to be deployed to a server or CDN

Short version:

`dist` is the production-ready output folder generated after building your project. It contains optimized files that are actually shipped to users.

14. What are `browserlists`?

Browserslist (often written as `browserslist`) is a configuration that tells your frontend build tools **which browsers your project needs to support**.

Tools like:

- **Babel** → decides how much to transpile
- **Autoprefixer** → adds necessary CSS vendor prefixes
- **Webpack/Parcel** → optimize output based on target browsers

…all use Browserslist to generate the correct, compatible code.

Example (in package.json):

"browserslist": [

">0.5%",

"last 2 versions",

"not dead"

]

This means:
Support browsers used by >0.5% of users, the last 2 versions of each major browser, and ignore outdated/unsupported ones.

Short version:

Browserslist tells build tools which browsers to target so your code compiles correctly for those environments.