**What is `NPM`?**

NPM (Node Package Manager) is a package manager for the JavaScript programming language. It is a command-line tool that comes bundled with Node.js, a JavaScript runtime environment. NPM is used to manage and install packages, libraries, and dependencies required for JavaScript projects, both on the client-side (in web browsers) and the server-side (in Node.js applications).

Here are some key features and functionalities of NPM:

1. Package Management: NPM provides a vast registry of public JavaScript packages, where developers can find and install packages to use in their projects. The registry contains thousands of open-source libraries and tools that can be easily integrated into JavaScript applications.
2. Dependency Management: NPM simplifies dependency management by allowing developers to specify the required packages and their versions in a configuration file called **package.json**. NPM uses this file to manage and install all the project dependencies, ensuring that the correct versions are installed and resolving any conflicting dependencies.
3. Installation and Updating: NPM provides a simple command-line interface to install packages. Developers can use the **npm install** command to download and install all the project dependencies specified in the **package.json** file. Additionally, NPM allows updating packages to their latest versions with the **npm update** command.
4. Scripts and Lifecycle Hooks: NPM enables developers to define custom scripts in the **package.json** file. These scripts can be executed using the **npm run** command and are commonly used for various tasks like running tests, building the project, starting a development server, and more. NPM also provides lifecycle hooks (preinstall, postinstall, etc.) that allow developers to perform specific actions during the package installation process.
5. Versioning and Semantic Versioning: NPM uses semantic versioning (SemVer) to manage package versions. Semantic versioning consists of three parts: MAJOR.MINOR.PATCH. By following this versioning scheme, packages can communicate the impact of new releases (whether they introduce breaking changes, add new features, or provide bug fixes) and allow developers to specify version constraints in their **package.json** file.
6. Private Packages and Registries: NPM supports the creation and publishing of private packages for organizations or individual developers. It allows you to set up private registries to host and distribute proprietary code or packages that are not intended to be publicly available.

NPM has become a widely adopted tool in the JavaScript ecosystem and is used by developers around the world to manage dependencies and streamline JavaScript development workflows.

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

Parcel and Webpack are both popular bundlers for JavaScript and web development. They serve similar purposes but have some differences in terms of configuration and features.

1. Parcel:
   * Parcel is a zero-configuration bundler, meaning it requires minimal setup and can automatically handle most tasks without additional configuration.
   * It aims to simplify the development process by eliminating the need for complex configuration files. It automatically detects and bundles assets like HTML, CSS, JavaScript, images, and more.
   * Parcel offers a built-in development server with hot module replacement, which allows for live reloading and fast development iterations.
   * It supports various languages and frameworks out of the box, making it easy to use with projects built with React, Vue.js, TypeScript, and others.
   * With its automatic optimization features, Parcel optimizes the bundled code for production, including minification, tree shaking, and cache busting.
2. Webpack:
   * Webpack is a highly configurable bundler that provides more flexibility and customization options compared to Parcel.
   * It allows developers to define custom entry points, specify loaders for different file types, and configure various optimization options.
   * Webpack supports a wide range of plugins and loaders, enabling advanced features such as code splitting, lazy loading, and advanced transformations for different file types.
   * It has a mature ecosystem and extensive community support, with a wide range of plugins and loaders available to extend its functionality.
   * Webpack is widely adopted and well-suited for complex projects and large-scale applications that require fine-grained control over the build process.

**Why do we need bundlers like Parcel or Webpack?**

1. Code Organization: Bundlers help organize and manage the codebase by allowing developers to split their code into modules and separate files. This modular approach promotes code reuse, maintainability, and scalability.
2. Dependency Management: Bundlers handle the resolution and management of dependencies between different modules or files. They ensure that the correct order of importing modules is maintained and handle bundling and transforming dependencies as needed.
3. Asset Bundling: Bundlers can process and bundle various types of assets, including JavaScript, CSS, images, fonts, and more. This simplifies the inclusion of different types of assets in the application and ensures efficient delivery to the browser.
4. Code Optimization: Bundlers optimize the bundled code for production, including techniques like minification, dead code elimination, and tree shaking. This helps reduce file size, improve performance, and optimize the loading time of the application.
5. Development Features: Bundlers offer development features like hot module replacement, which allows for live reloading and faster development iterations. They also provide development servers for local development and debugging.

In summary, bundlers like Parcel and Webpack help streamline the development workflow, optimize code, and improve the performance of web applications by efficiently bundling and managing dependencies, organizing code, and providing additional development features. The choice between Parcel and Webpack depends on the specific project requirements, complexity, and customization needs.

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

The **.parcel-cache** directory is a cache folder created by the Parcel bundler. When you use Parcel to bundle your project, it automatically generates a cache of compiled assets, dependencies, and other intermediate files to improve build performance and avoid recompiling unchanged code.

The cache folder, named **.parcel-cache**, is typically located in the root directory of your project. It stores cached data related to the bundling process, including compiled JavaScript, CSS, and other assets.

The purpose of the cache is to speed up subsequent builds by reusing previously compiled assets. When you make changes to your code and rebuild the project, Parcel compares the modification timestamps of the source files with the cached versions. If there are no changes, it can skip recompiling those files, resulting in faster build times.

The **.parcel-cache** directory is usually safe to ignore in version control systems (e.g., Git) as it can be regenerated by Parcel when needed. However, it's recommended to include the **.parcel-cache** directory in your project's **.gitignore** file to avoid unnecessary commits and keep the repository clean.

If you encounter any issues with the bundling process or cache inconsistencies, deleting the **.parcel-cache** directory and re-running the build can often help resolve such problems. Parcel will recreate the cache as necessary during the next build.

What is `npx` ?

**npx** is a command-line tool that comes bundled with NPM (Node Package Manager) since version 5.2.0. It stands for "Node Package Executor" and provides a convenient way to run Node.js packages without globally installing them on your system.

Here are some key features and use cases of **npx**:

1. Package Execution: **npx** allows you to execute a specific version of a package without needing to install it globally. It automatically installs the required package locally (in the current project directory) and runs the command associated with it.
2. Latest Version: By default, **npx** installs and uses the latest version of a package from the NPM registry. This ensures that you're always using the most up-to-date version when running a package.
3. Running Binaries: **npx** can execute binaries provided by a package. It automatically resolves the binary associated with a package and runs it, eliminating the need to specify the full path to the binary.
4. Temporary Environment: **npx** creates a temporary environment for executing packages. This allows you to avoid polluting your global or project-specific dependencies with packages that are only needed for a one-time task or script.
5. Installing Packages: If a requested package is not found in the local or global package registry, **npx** will attempt to install it before executing the command. This simplifies the workflow of trying out new packages or running one-off scripts.

Here are some examples of using **npx**:

* Running a package command:

cssCopy code

npx eslint src/

This command executes the locally installed **eslint** package, running it on the **src/** directory.

* Running a specific package version:

cssCopy code

npx webpack@4.0.0

This command runs the **webpack** package version 4.0.0.

* Trying out new packages:

luaCopy code

npx create-react-app my-app

This command installs the latest version of **create-react-app** and uses it to create a new React application in the **my-app** directory.

The **npx** tool simplifies package execution and helps manage project dependencies without the need for global installations. It is particularly useful in scenarios where you want to run a package temporarily or use the latest version without explicitly installing it.

**What is the difference between `dependencies` vs `devDependencies`**

In the context of NPM (Node Package Manager), **dependencies** and **devDependencies** are two different sections in the **package.json** file that serve different purposes for managing project dependencies.

1. Dependencies:
   * The **dependencies** section is used to specify the packages that are required for the production runtime of your application.
   * Packages listed in the **dependencies** section are installed when someone installs your project as a dependency or when you deploy your application to a production environment.
   * These packages are essential for the application to run properly and include libraries, frameworks, and utilities that the application relies on during runtime.
2. DevDependencies:
   * The **devDependencies** section is used to specify the packages that are only required during development and are not necessary for the production runtime of your application.
   * Packages listed in the **devDependencies** section are typically development tools, testing frameworks, build systems, and other dependencies that aid in the development and testing processes.
   * These packages are not included when your application is published or deployed for production, and they are primarily used by developers during the development and testing phases.

The key differences between **dependencies** and **devDependencies** can be summarized as follows:

* **dependencies** are required for the application to run in a production environment, while **devDependencies** are only needed during development and testing.
* **dependencies** are installed when the project is deployed or installed as a dependency, whereas **devDependencies** are not included in production deployments.
* **dependencies** include packages essential for the runtime functionality of the application, while **devDependencies** include tools and dependencies for development and testing purposes.

By separating dependencies into these two categories, it allows for a clear distinction between the packages needed for the production environment and those needed for development and testing workflows. It helps keep the production environment lean and ensures that unnecessary development dependencies are not included, reducing the overall package size and potential security risks.

**What is Tree Shaking?**

Tree shaking is a technique used in modern JavaScript bundlers to eliminate unused code from the final bundled output. It is an optimization process that helps reduce the size of the JavaScript bundle by removing any dead or unreachable code.

Here's how tree shaking works:

1. Static Analysis: During the bundling process, the bundler performs static analysis of the JavaScript code and its dependencies to determine which parts of the code are actually used and which are not.
2. Dependency Graph: The bundler builds a dependency graph that represents the relationships between different modules and their dependencies.
3. Dead Code Removal: Using the information from the dependency graph and static analysis, the bundler can identify and remove any code that is not referenced or used by the application. This includes functions, variables, and entire modules that are not required.
4. Minimized Bundle Size: By removing the unused code, tree shaking reduces the size of the JavaScript bundle that is sent to the browser. This results in faster downloads, quicker parsing and execution by the browser, and improved overall performance of the application.

Tree shaking is particularly effective with modern JavaScript module systems like ES modules (ESM) that have static import/export statements. It leverages the fact that static imports and exports can be analyzed at build time to determine the dependencies and eliminate dead code.

To enable tree shaking, it is important to use a bundler that supports this optimization, such as Webpack, Rollup, or Parcel, and ensure that your code is written using ES modules. Additionally, tree shaking works best with pure functions and modules that have minimal side effects, as it can safely remove unused code without impacting the application's behavior.

Overall, tree shaking is a powerful optimization technique that helps reduce the size of JavaScript bundles, improves application performance, and allows developers to ship leaner code to end users.

**What is Hot Module Replacement?**

Hot Module Replacement (HMR) is a feature provided by bundlers like Webpack and Parcel that allows developers to update modules in a running application without requiring a full page reload. It enables faster development iterations by applying code changes immediately and preserving the application's state.

Here's how Hot Module Replacement works:

1. Initial Application Load: When you start your application with HMR enabled, the bundler generates a so-called "HMR runtime" that runs alongside the application.
2. Detecting Module Changes: As you make changes to your code and save the files, the bundler's development server compares the modified modules with the previous versions. It uses various mechanisms like file hashing or time stamps to detect which modules have changed.
3. Replacing Modules: Once a change is detected, the bundler sends an update signal to the HMR runtime and triggers a process to update the affected modules.
4. Applying Changes: The HMR runtime applies the updated modules by patching the running application in memory. It only replaces the modified modules and their dependencies, keeping the rest of the application intact.
5. Preserving Application State: During the module replacement process, HMR tries to preserve the application's state as much as possible. It retains the current state of components and avoids losing user input or any other dynamic data.
6. Refreshing the UI: After the module replacement, the HMR runtime updates the UI of the application with the new code. This update is typically seamless and instantaneous, as only the changed modules are replaced.

Hot Module Replacement provides several benefits for developers:

* Faster Iterations: With HMR, developers can see the changes they make to the code immediately without having to manually refresh the browser or restart the application.
* Improved Developer Experience: HMR speeds up the development process, making it more interactive and productive. It eliminates the need to manually navigate to a specific state or retrace the steps to reach a particular point in the application after a full reload.
* Preserved Application State: HMR attempts to maintain the current state of the application during module updates, ensuring a smooth development experience and reducing the need to re-enter data or recreate application states.
* Increased Efficiency: HMR reduces the feedback loop time by eliminating the waiting time for a full rebuild and refresh cycle. It allows developers to experiment, test, and iterate on their code more efficiently.

It's important to note that while HMR is a powerful tool for development, it is not meant to replace proper testing and quality assurance processes. It primarily serves as a productivity tool during development to speed up the feedback loop and improve developer experience.

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

Here are five commonly appreciated features of Parcel:

1. Zero Configuration: Parcel requires little to no configuration out of the box. It automatically detects and bundles various file types, including HTML, CSS, JavaScript, images, and more. This makes it easy to get started with a new project without spending time on complex setup.
2. Fast Development Server: Parcel comes with a built-in development server that supports hot module replacement (HMR). This allows for live reloading and fast development iterations, enabling developers to see changes immediately as they code.
3. Automatic Asset Optimization: Parcel optimizes the bundled code for production. It performs optimizations such as minification, tree shaking (removing unused code), and cache busting (generating unique filenames for updated assets). These optimizations help reduce the bundle size, improve performance, and ensure efficient delivery to the browser.
4. Support for Multiple Languages and Frameworks: Parcel supports a wide range of languages and frameworks, making it versatile and suitable for different project requirements. It works well with popular frameworks like React, Vue.js, and Angular, as well as languages like TypeScript and SCSS.
5. Extensibility through Plugins: Parcel offers a plugin system that allows developers to extend its functionality. There are various community-maintained plugins available to enhance the bundler's capabilities, enabling features like code splitting, static site generation, and more.

Now, let's delve deeper into three of these features:

1. Zero Configuration: Parcel's zero-configuration approach simplifies the development process by eliminating the need for extensive configuration files. Developers can start working on their projects right away without spending time on complex setup steps. Parcel intelligently detects dependencies and handles bundling tasks automatically, reducing the learning curve and making it beginner-friendly.
2. Fast Development Server: Parcel's built-in development server is a valuable feature during the development phase. With hot module replacement (HMR) support, developers can see instant updates as they modify their code, without the need for manual page reloading. This feature speeds up the development iterations, enhances productivity, and provides a smooth development experience.
3. Automatic Asset Optimization: Parcel's ability to optimize assets during the bundling process is a significant advantage. It applies various optimizations such as minification, tree shaking, and cache busting by default. These optimizations result in smaller bundle sizes, faster loading times, and improved performance of the web application. By automating these optimization steps, developers can focus more on coding and rely on Parcel to handle the best practices for optimizing their assets.

These features collectively contribute to making Parcel a popular choice among developers, offering a seamless and efficient development experience with minimal configuration and powerful optimization capabilities

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

The **.gitignore** file is a text file used to specify files and directories that should be ignored by Git, a version control system. When you create a **.gitignore** file in your project's root directory or any specific subdirectory, Git uses it as a guide to determine which files and directories should not be tracked or committed.

The purpose of the **.gitignore** file is to exclude certain files and directories from being included in version control. This is useful for preventing sensitive or unnecessary files from being shared with others or cluttering the repository. It also helps to improve the performance and cleanliness of the repository.

Here are some guidelines on what to include and not include in a **.gitignore** file:

1. Include:
   * Build artifacts: Compiled binaries, object files, or build output directories should be included to avoid tracking unnecessary generated files. Examples include **/build/**, **/dist/**, or **\*.exe**.
   * Dependency directories: Include directories where external dependencies or packages are installed. Examples include **/node\_modules/** or **/vendor/**.
   * Environment-specific files: Configuration files specific to certain environments (e.g., development, testing, or production) should be included. Examples include **.env**, **.env.local**, or **\*.env.\***.
2. Exclude:
   * Source code and project files: Files and directories containing source code, project configuration files, or important development files should typically be committed and not added to the **.gitignore** file. These files are essential for collaborating on the project and building it from scratch on different machines.
   * Personal and sensitive information: Avoid committing files that contain personal credentials, API keys, passwords, or sensitive data. These files should be kept out of version control entirely, even if they are listed in the **.gitignore** file.

It is important to carefully consider what should be added or excluded in the **.gitignore** file based on the specific needs of your project. It is a good practice to review and update the **.gitignore** file periodically as your project evolves and new files or directories are introduced.

Additionally, it's worth noting that the **.gitignore** file can use wildcards, patterns, and comments to specify files and directories to ignore. You can refer to the Git documentation for more details on the syntax and usage of **.gitignore** files.

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

**package.json** and **package-lock.json** are both files used in Node.js projects to manage dependencies, but they serve different purposes and have distinct roles in the dependency management process.

1. package.json:
   * **package.json** is the primary file used to define a Node.js project's metadata, configuration, and dependencies.
   * It includes information such as the project's name, version, entry points, scripts, dependencies, and more.
   * Developers manually edit this file to specify the project's dependencies, including their version ranges or specific versions.
   * **package.json** is committed to version control and shared with other developers to ensure consistent dependency installations across different environments.
2. package-lock.json:
   * **package-lock.json** is an automatically generated file that provides deterministic and reproducible dependency resolution.
   * It is created when dependencies are installed or modified using package managers like NPM (Node Package Manager).
   * **package-lock.json** contains a complete record of the exact dependency tree generated by the package manager, including the resolved versions of all dependencies and their transitive dependencies.
   * This file ensures that subsequent installations of the project use the exact same versions of the dependencies, thereby preventing unexpected or non-reproducible dependency installations.
   * **package-lock.json** is intended to be committed to version control so that all developers working on the project have access to the exact same dependency tree.

In summary, **package.json** is the manually maintained configuration file that defines the project's metadata and lists the desired dependencies and their version ranges, while **package-lock.json** is an automatically generated file that provides a detailed and reproducible snapshot of the dependency tree with resolved versions. Together, these files facilitate consistent dependency installations and ensure that all developers working on the project have the same set of dependencies installed.

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

It is generally recommended not to modify the **package-lock.json** file manually. Here are a few reasons why:

1. Dependency Consistency: The purpose of the **package-lock.json** file is to provide a consistent and reproducible snapshot of the exact dependency tree with resolved versions. Modifying this file directly can lead to inconsistencies between different development environments, making it challenging to reproduce the same dependencies across different machines.
2. Dependency Resolution: The **package-lock.json** file is automatically generated by package managers like NPM based on the dependencies listed in the **package.json** file. It takes into account various factors like version ranges, transitive dependencies, and compatibility constraints to resolve the dependencies. Modifying the **package-lock.json** file manually can disrupt the correct resolution process and result in unexpected behavior or incompatible dependencies.
3. Collaboration and Version Control: The **package-lock.json** file is meant to be shared and committed to version control. When multiple developers work on the same project, having a consistent **package-lock.json** file ensures that they all have the same set of dependencies. Modifying the file directly can introduce inconsistencies and create conflicts when collaborating with others.
4. Automatic Updates: When you run dependency installation commands (**npm install**, **npm update**, etc.), the package manager uses the **package-lock.json** file to determine the exact versions of the dependencies to install or update. If you modify the **package-lock.json** file manually, these automatic updates may not work as expected, and you may not get the desired versions or updates.

If you need to make changes to the project's dependencies, it is recommended to modify the **package.json** file instead. After updating the **package.json** file, you can run the appropriate package manager command (**npm install**, **yarn**, etc.) to update the **package-lock.json** file automatically based on the changes in **package.json**.

By following this recommended approach, you can ensure dependency consistency, maintain reproducibility across different environments, and avoid potential conflicts or unexpected behavior caused by manual modifications to the **package-lock.json** file.

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

**node\_modules** is a directory created by package managers like NPM to store the dependencies of a Node.js project. When you run the **npm install** command or equivalent, the package manager reads the **package.json** file, resolves the dependencies specified in it, and downloads and installs them into the **node\_modules** directory.

It is generally not considered a good practice to include the **node\_modules** directory in version control systems like Git. Here's why:

1. Large Size: The **node\_modules** directory can be quite large, especially for projects with numerous dependencies. Including it in version control can significantly increase the repository size, making it more cumbersome to clone or download.
2. Redundancy: The dependencies listed in the **package.json** file are sufficient to install the required packages on any machine. Including the entire **node\_modules** directory in version control is redundant and unnecessary.
3. Dependency Updates: Dependencies often receive updates, bug fixes, and security patches. By including **node\_modules** in version control, you may miss out on these updates. Instead, developers can rely on the **package.json** file and the **package-lock.json** file (or **yarn.lock** for Yarn) to reproduce the exact dependency tree with the correct versions when cloning or setting up the project.

To avoid including **node\_modules** in version control, it is recommended to:

1. Add **node\_modules** to the **.gitignore** file to prevent Git from tracking it.
2. Ensure that **package.json** and **package-lock.json** (or **yarn.lock**) are committed to version control. These files define the project's dependencies and their exact versions, allowing other developers to install the correct dependencies when setting up the project.
3. Use the appropriate package manager command (**npm install**, **yarn**, etc.) to install the dependencies on each developer's machine based on the **package.json** and **package-lock.json** files.

By following these best practices, you can keep your repository lean, maintain consistency across different environments, and allow for easier dependency management and updates.

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

The **dist** folder is a commonly used convention in web development projects, especially those that involve build processes or bundling. "Dist" stands for "distribution" and typically contains the final, production-ready version of your project.

When you build your project, whether it's a web application, a library, or any other software, the output or artifacts generated by the build process are often placed in the **dist** folder. The contents of the **dist** folder are typically optimized and ready to be deployed or distributed to end-users.

Here are a few key aspects of the **dist** folder:

1. Output of the Build Process: The **dist** folder contains the transformed, bundled, or compiled files that are the result of the build process. These files are often optimized for production use, which may include minification, concatenation, transpilation, or other transformations.
2. Standalone and Self-contained: The files in the **dist** folder are designed to be self-contained and standalone. They should include all the necessary assets, such as HTML, CSS, JavaScript, images, or any other resources required to run the application or use the library. This allows the contents of the **dist** folder to be easily deployed or distributed without any additional dependencies or build steps.
3. Ready for Deployment: The **dist** folder typically represents the final stage of the development process, where the project is ready for deployment to a web server, hosting platform, or any other target environment. The contents of the **dist** folder can be copied or deployed to a production server or shared with others as a deployable package.
4. Not Meant for Version Control: Similar to the **node\_modules** directory, the **dist** folder is usually excluded from version control systems like Git. Since the **dist** folder contains the output files generated during the build process, it can be regenerated from the source code and build configuration. It is more efficient to exclude it from version control to avoid unnecessary repository bloat and potential conflicts.

It's important to note that the specific structure and contents of the **dist** folder can vary depending on the project's build system, bundler, or framework used. It's a convention rather than a strict requirement, and different projects may use alternative names or locations for the distribution output.

Overall, the **dist** folder serves as the destination for build artifacts, containing the optimized, standalone version of your project that is ready for deployment or distribution.

**What is `browserlists`**

The **browserslist** is a configuration file or entry in the **package.json** file that defines the list of target browsers or environments that a project intends to support. It is used by various front-end tools, such as Autoprefixer and Babel, to determine the specific browser versions or environments that require compatibility adjustments or transpilation during the build process.

Here's how **browserslist** works:

1. Configuration File: The **browserslist** configuration can be specified in a separate **.browserslistrc** file or as a **"browserslist"** key in the **package.json** file. It follows a specific syntax and format.
2. Target Browsers: The **browserslist** configuration defines the target browsers or environments using a range of criteria, such as browser names, versions, and usage statistics. For example, you can specify specific versions like **"last 2 versions"**, **"> 1% in US"**, or **"defaults"** to represent a pre-defined set of browsers.
3. Compatibility Adjustments: Front-end tools like Autoprefixer and Babel use the **browserslist** configuration to determine which CSS prefixes or JavaScript language features need to be polyfilled or transpiled to ensure compatibility with the specified target browsers. This ensures that your web application or site behaves consistently across different browsers and environments.
4. Tool Integration: Various front-end tools and frameworks support **browserslist** by reading the configuration and applying the necessary adjustments during the build process. For example, Autoprefixer uses **browserslist** to add vendor prefixes to CSS properties, while Babel uses it to determine which JavaScript transformations are required based on browser support.

By defining the target browsers or environments in the **browserslist** configuration, developers can optimize their development workflow by automatically applying compatibility adjustments only where needed. This helps ensure that the final output of the build process is tailored to the target browsers or environments specified in the configuration.

It's worth noting that different tools may have their own specific ways of reading and interpreting the **browserslist** configuration. Therefore, it's essential to consult the documentation of the specific tools you're using to understand how they utilize **browserslist** and ensure compatibility in your project.