# Chapter 2 – Igniting our App

# Making an Application Production-Ready

When we build an application, preparing it for **production** requires several optimizations to ensure **speed, scalability, and a smooth user experience.**

### *Essential Optimizations -*

* **Clean up code** - Remove console.log, debug statements, and unused code.
* **Minify & compress** - Reduce the size of JavaScript, CSS, and HTML.
* **Bundle files** - Combine multiple files into fewer bundles for faster loading.
* **Enable caching** - Use cache headers/CDN so users don’t re-download assets.
* **Optimize assets** - Compress images, fonts, and media for performance.

But when we run npx create-react-app in the terminal, it automatically sets up a React project that’s already optimized for production with all these features included.

Now the question is - what goes into building a tool like create-react-app.

To understand that clearly, let’s try building our own simplified version of create-react-app.

Creating our own create-react-app

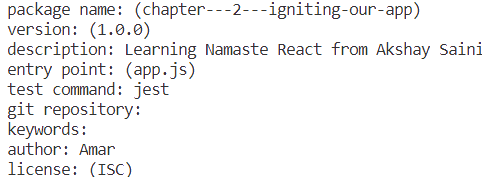
React by itself doesn’t guarantee good speed or high performance for an app.

To build a fast, scalable, and production-ready application, we need to use extra tools and libraries. That’s where NPM comes in - it helps us add and manage these tools in our project.

Many people think NPM stands for *Node Package Manager*, but the official docs say it doesn’t actually stand for anything. It could even mean *"No Problem Man"* - just a fun fact. No matter what the name is, NPM is basically a package manager for Node.js. It gives developers access to a huge collection of packages, libraries, and tools.

To start using NPM in our project, we run the command: npm init

This starts the NPM setup process. It asks some basic questions like:

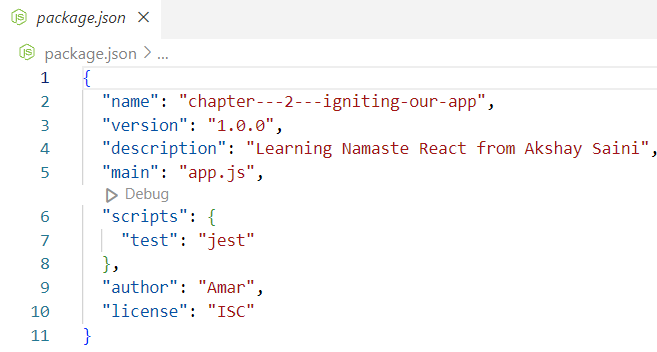


Once we answer those, NPM creates a file called package.json.  
This file stores all the important information about our project - including the list of dependencies we install later.

**Note:** If we want to skip all these questions and use default values, we can simply run: npm init -y

This will instantly create a package.json file with default settings.

At this point, the package.json file will look something like this:



These values are based on the answers we gave during the npm init process.

What is the package.json File?

The package.json file is a special configuration file used by NPM.

It keeps track of all the packages (or libraries) your project needs to work.

When you run a command like: npm install <package-name> , That package gets downloaded into your project, and its name and version are saved under the "dependencies" section in the package.json file.

These are called dependencies because your project depends on them to run properly.

Earlier, we talked about the different things needed to make an app ready for production - like code optimization, image compression, and caching.

All these tasks are usually handled by a tool called a bundler.

### What is a Bundler?

A **bundler** is a tool that **prepares your app for production**.  
It **bundles** (or combines) all your files and assets so the app can be deployed and run smoothly. Before the app is deployed, the **bundler (like Webpack, Vite, Parcel, etc.)** performs several optimizations to improve performance:

* **Code Splitting** - Breaks the code into smaller parts (lazy loading) so only what’s needed is loaded first.
* **Chunking** - Splits large files into smaller chunks for efficient delivery.
* **Image Optimization** - Compresses and optimizes images for faster loading.
* **Code Compression/Minification** - Removes spaces, comments, and shortens variable names → smaller files.
* **Tree Shaking -** Removes unused/dead code from the final bundle.
* **Caching** - Adds cache-busting techniques (like unique hash in filenames) for better browser caching.
* **Other Optimizations -** Preloading, prefetching, gzip/brotli compression, etc.

When you run the create-react-app command, tools like **Webpack** and **Babel** work behind the scenes to do all these tasks automatically.

Some popular bundlers are:

* **Parcel**
* **Webpack**
* **Vite**

***Let’s Use Parcel as Our Bundler.***

To install Parcel in our app, run:

npm install -D parcel

Here, -D means you're installing it as a dev dependency.

There are two types of dependencies in any app.

1. Regular dependencies - Needed in production (e.g., React, Axios)
2. Dev dependencies - Needed only during development (e.g., bundlers, linters, testing tools)

Parcel is a development tool, so we add it as a dev dependency.



If we look at the package.json file after installing Parcel, we’ll find it listed under devDependencies with a version like "^2.10.3"

"devDependencies": {

"parcel": "^2.10.3"

}

To understand this versioning better, let’s break it down:

What Are Major, Minor, and Patch Versions?

In a version like **2.10.3**, each number has a meaning:

* **2** - This is the **major version** - Big changes that might break existing code
* **10** - This is the **minor version** - New features added, but still backward compatible
* **3** - This is the **patch version** - Bug fixes or small updates that don’t change how the feature works

What Do the Symbols ^ and ~ Mean?

#### These are **version prefixes** in **npm (Node Package Manager)**.They control **which versions of a dependency can be installed/updated** when you run npm install.

#### **1. Caret (^)**

Example: "parcel": "^2.10.3"

* Allows updates to:
  + **Minor versions** → like 2.11.0, 2.12.5
  + **Patch versions** → like 2.10.4, 2.10.9
* Keeps the **major version** fixed (still 2)

#### **2. Tilde (~)**

Example: "parcel": "~2.10.3"

* Allows updates to:
  + **Only patch versions** → like 2.10.4, 2.10.5
* Keeps both **major** and **minor versions** fixed (still 2.10)

These updates are saved in the package-lock.json file, which tracks exactly which version was installed.

Note - The commands npm install parcel --save-dev & npm install parcel -D do the **same thing**.

Both install Parcel as a **development dependency**, meaning it’s only needed during development, not in production.

What is package.lock.json?

The package-lock.json file tracks the exact versions of all the packages used in a project.

It locks these versions, making sure that everyone working on the project has the same versions of the packages, no matter where they are working, like on their local machine.

In addition to locking versions, the SHA hash (a security code) is used to check the integrity of the packages.

This ensures that the packages haven’t been tampered with or changed in an unexpected way. The hash makes sure that when you install a package, it’s exactly the same as it was when it was first added to the project.

The package-lock.json file stores the exact versions of all installed dependencies (and sub-dependencies).

This ensures the node\_modules folder can be recreated exactly the same on any machine.

Prevents issues where package.json allows flexible versions (^, ~), which could install slightly different versions.

Used for consistency,reliability,andreproducibility across development, testing, and production.

What is the use of the SHA key?

The SHA key (SHA hash) is a security feature used to ensure package integrity. It acts like a digital fingerprint for a package.

Here's how it works:

* Package Integrity: The SHA key ensures that the package you're installing has not been tampered with or altered in any way. If someone tries to modify the package, the hash will no longer match & the package will be considered invalid.
* Verification: When you install a package, it confirms that the package you’re downloading is **exactly the same version** that was originally added to the project.
* Security: Protects against malicious code injection by verifying authenticity.

In short, the SHA key is important because it helps to ensure that the packages used in your project are safe, unmodified, and exactly what you expect them to be.

What is the difference between package.json and package.lock.json?

package.json allows flexible versions, while package-lock.json locks exact versions for consistent installs.

Relation between package.json and package-lock.json -

***package.json -***

* + Created when you run npm init.
  + Defines **what packages** your project needs.
  + Uses **version ranges** (^, ~, exact) → allows flexibility.

***package-lock.json -***

* + Created/updated automatically when dependencies are installed.
  + Records the **exact versions** actually installed (including sub-dependencies).
  + Ensures **consistency across all environments,** regardless of version ranges in package.json.

***package.json = what you need (flexible), package-lock.json = what you got (exact).***

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

You shouldn't modify the **package-lock.json** file because it ensures consistency by locking the exact versions of dependencies used in your project. If you manually change this file, you risk causing version mismatches or errors, especially when working in a team or deploying to production. Deleting or editing it can result in different versions of packages being installed, which can lead to unexpected issues.

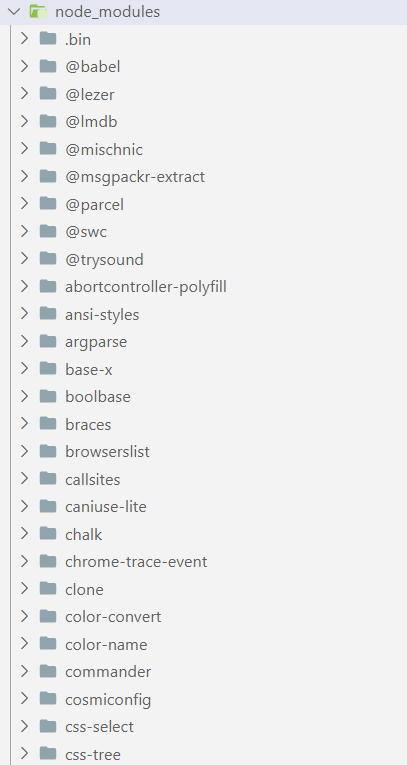
It’s best to let **npm** manage the **package-lock.json** file automatically.

What are node modules? What is Transitive depenancy?

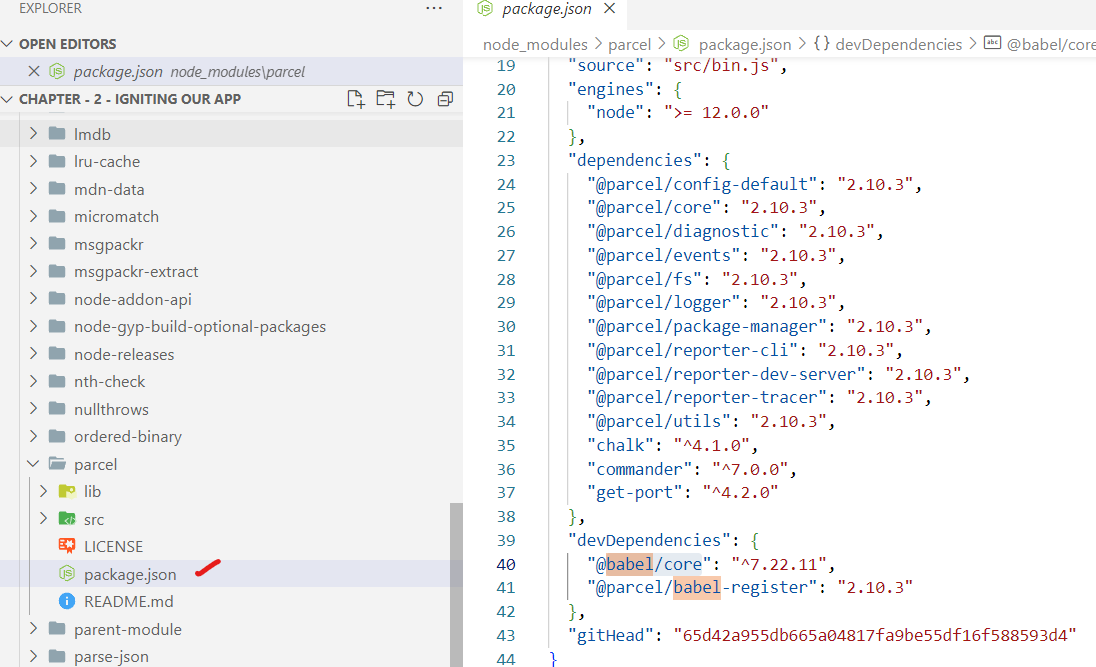
When you run npm install <packageName>, npm downloads the specified package from the internet and installs it in the node\_modules folder. This folder stores all the packages (dependencies) required for your project to work.

Since the node\_modules folder can become very large, it is important to add it inside .gitignore file to prevent it from being tracked in version control.

Inside the **node\_modules** folder, you may also find other packages that were automatically installed alongside the one you specified during npm install. This occurs because many packages depend on additional packages to function properly. These additional dependencies, on which the current package depends, are called **transitive dependencies.** They form a chain of dependencies that are necessary for the original package to work.



Each package folder inside the node\_modules directory contains its own **package.json** file. This file lists both the regular dependencies (needed for the package to run) and the development dependencies (needed only during development). Essentially, a package’s **package.json** provides information about its transitive dependencies - the dependencies that the package itself relies on.



### What is .gitignore? What should we add and not add to it?

* The. gitignore file is a simple text file that tells Git **which files or folders to ignore** when committing to a repository.
* Purpose - Prevents **unnecessary or sensitive files** from being tracked in version control.

### *Commonly ignored files:*

* **Autogenerated files -** build outputs, logs, temporary files.
* **Dependencies -** e.g., node\_modules/.
* **Sensitive information -** API keys, security keys, environment files (.env).

***⚡*** One-liner (Interview Ready) -. gitignore keeps unwanted or sensitive files out of Git, ensuring cleaner and more secure repos.

# Common Conventions in. gitignore

### 1. \* → Wildcard (any characters, any length)

Matches **anything** in filenames or folder names.

\*.txt # Ignore all .txt files anywhere

temp\* # Ignore all files/folders starting with "temp"

### 2. / → Root path restriction

A leading / matches **only at the repo root.**

/temp\* # Ignore root-level temp files/folders only

/node\_modules # Ignore node\_modules only at root

node\_modules # Ignore node\_modules everywhere

### 3. # → Comments

Just for adding explanations inside. gitignore.

# Ignore all log files

### What **not** to add to. gitignore?

package.json & package-lock.json files should not be added to .gitignore because these files are crucial for defining project dependencies and ensuring that the same versions of packages are installed on every machine.

Running the app with Parcel

After installing Parcel, we can start our app by running: npx parcel index.html

### When we run this command, **Parcel** uses the entry point (index.html) to create a development build of our app. It then serves the app on a local server at <http://localhost:1234/>, making it accessible in the browser.

### Difference Between npm and npx -

***npm (Node Package Manager)***

Used to **install** packages (locally or globally) into your project.

Example: npm install parcel. This installs Parcel inside node\_modules.

***npx (Node Package Execute)***

Used to **execute a package** without needing a global install.

If the package exists locally → npx runs it.

If not installed → npx downloads it temporarily and runs it.

Example: npx parcel index.html

This command runs Parcel and uses index.html as the entry point.

### One Liner -

* ✅ Use **npm** → to install
* ✅ Use **npx** → to run

Why CDN Links Are Not Recommended to Include React in a Project?

Using CDNs (Content Delivery Networks) to bring React and ReactDOM into your project is not ideal for a few reasons:

### *1. Extra Network Calls*

* Using <script> with a CDN (like unpkg) means the browser must fetch React/ReactDOM from the internet.
* This adds **extra network requests** → slower start up in dev and prod.

### *2. Local Copies Already Exist*

* If React is installed via **npm**, it’s already in your node\_modules.
* No need to fetch again from the web.
* Local copies are faster and more reliable.

#### **3. Version Management Is Harder**

With CDNs, if you want to update React to a newer version, you need to **manually update the URL**.  
But if you're using npm, you can just run:

npm install react@latest

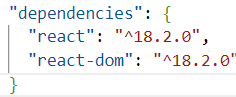
and everything is managed automatically in package.json.

Recommended way to install React Locally -

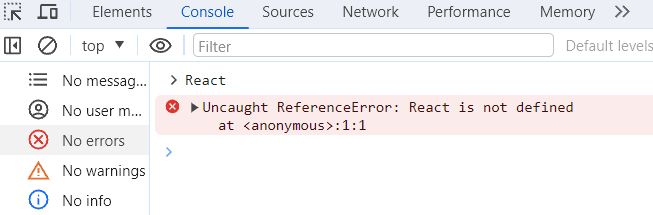
npm install react

npm install react-dom

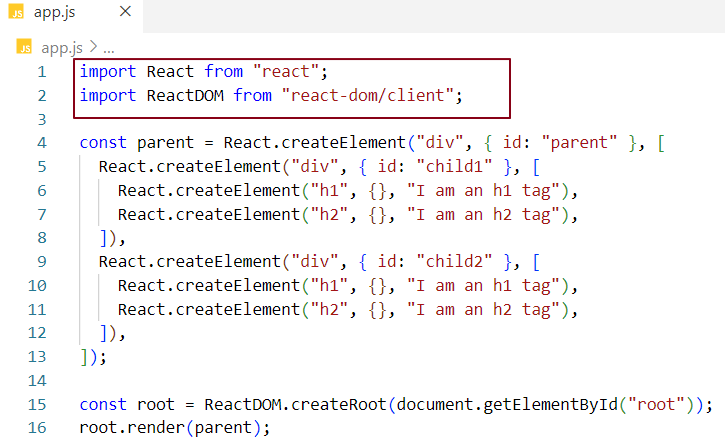
These are normal (runtime) dependencies, not dev dependencies - because they are needed when your app runs in production too.



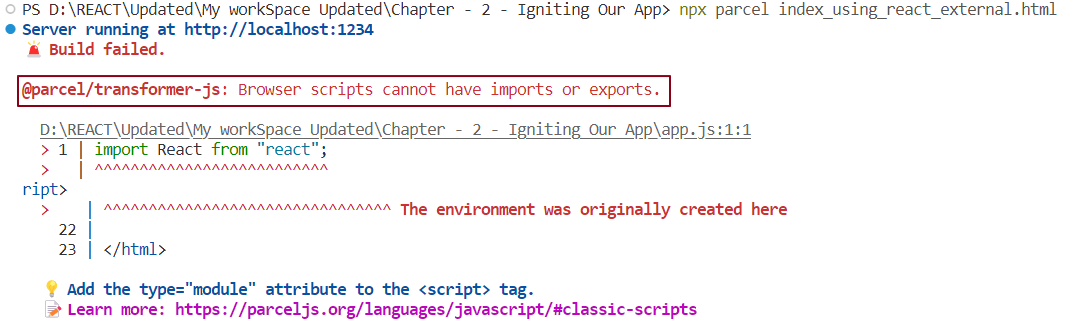
When we run the application, we encounter an error that says 'React is not defined.



The error is valid because our code doesn’t know where React is coming from. Although we’ve installed the React package, we haven’t imported it into our project yet. To fix this, we need to use the import keyword to bring in the React object from the react package (located in the node\_modules folder). Similarly, we should import **ReactDOM** from react-dom/client to fully access ReactDOM’s features.



We're not done yet — there's one more issue we'll face when running the app with Parcel.



This error occurs because the browser doesn’t natively understand import and export statements — it only understands standard JavaScript. When the browser tries to read app.js and encounters these module-related statements, it throws an error. To fix this, we need to tell the browser that app.js is a JavaScript module by adding type="module" to the <script> tag. This allows the browser to properly interpret the code and resolves the error.



Now that React and ReactDOM are properly set up, any changes we make to the code will automatically refresh in the browser upon saving, displaying the updated content. This works because **Parcel** handles **Hot Module Replacement (HMR)** behind the scenes.

### What is HMR (Hot Module Replacement)?

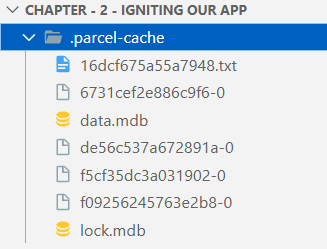
**Hot Module Replacement (HMR)** is a feature that allows you to update your application in real-time as you make changes to the code - without needing to refresh the entire browser page.

How does it work?

* **HMR Meaning -** HMR allows tools like Parcel to track changes in your files as you work on them.
* **File Watcher Algorithm -** Parcel uses a file-watcher algorithm, written in C++, to monitor your files in real time. Whenever a change is detected, the algorithm notifies the Parcel development server.
* **Role of Parcel -** Parcel receives the update and injects the modified module directly into the running app. This avoids a full page reload and preserves the application state — making development faster and smoother.

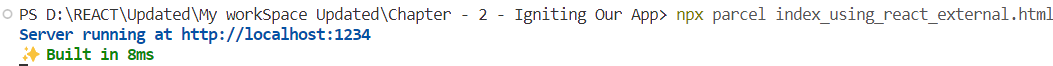
What is Caching? Where parcel stores caches?

**Parcel continuously caches code.** When we run the application for the first time, an initial build is created, which takes only a few milliseconds to complete. If we then make any code changes and save the application, another build is triggered, often taking less time than the previous one. This reduction in build time is due to Parcel's caching mechanism. During the first build, Parcel creates a folder called parcel-cache, where it stores cached data in binary format. In subsequent builds, Parcel immediately loads the code from the cache, resulting in faster builds with an enhanced developer experience.



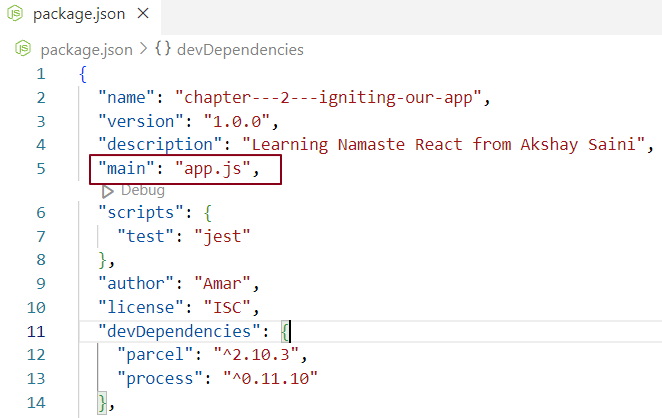






How to create a production build?

For a production build, use the command npx parcel build index.html, as it is more optimized than the development build. Since we have specified index.html as the entry point, it does not match the entry point defined in the package.json file (which is app.js). Therefore, remove the relevant configuration from the package.json file before proceeding with the build.



What is the dist folder?

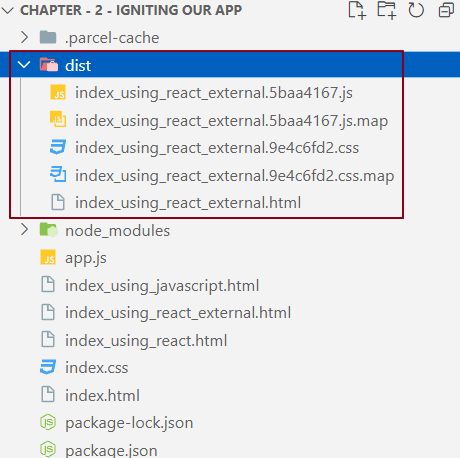
The dist **(distribution) folder** is the output directory created when the bundler (like Parcel, Webpack, or Vite) builds your application. It contains the **production-ready code**, which means all files are **minified, optimized, and bundled** together for better performance.

The JavaScript, CSS, and HTML files inside dist are compressed, hashed for cache-busting, and stripped of unnecessary code to make the app lightweight.

In addition, the folder also contains **compiled modules** - these are transformed versions of your source files (like ES6+ code transpiled to ES5, or JSX converted to regular JavaScript) so that they run smoothly across different browsers.

The dist folder is the one you deploy to production servers (e.g., Netlify, Vercel, AWS), not your raw source code.

✅ **In short:** The dist folder is the final, optimized package of your app — ready for production deployment.



**Note:** Always include the parcel-cache and dist folders in the .gitignore file, as they contain auto-generated code.

What is browserslist?

**Browserslist** is a tool used to define which browsers your frontend application should support.  
It lets you specify target browsers using queries in a configuration file.

Tools like **Babel** (for JavaScript transpilation) and **Autoprefixer** (for CSS prefixes) use these queries to automatically adjust the code for compatibility.

***Example Browserslist Config***

{

"browserslist": "> 1%, last 2 versions, not dead, Firefox ESR"

}

### *What it means -*

* > 1% → Support all browsers with more than 1% global market share.
* last 2 versions → Support the two most recent versions of each browser.
* not dead → Exclude browsers that are no longer maintained or updated.
* Firefox ESR → Explicitly include the Firefox Extended Support Release.

### *Misconception -*

This does **not** mean your app only supports the latest version of Firefox.  
It means your app will **definitely support Firefox ESR** along with the last 2 versions of all major browsers that meet the criteria.

# Difference Between **dist** &. cache Folder

***dist folder (Build Output) -***

* + Contains the **final bundled, minified, and optimized files** (JS, CSS, HTML, assets) ready for **production deployment.**
  + These are generated from your source code whenever you run a build.
  + Should **not** be committed to version control — always add to **.gitignore**, since it can be recreated from source.

***.cache folder (Build Cache) -***

* + Stores **cached data** created by Parcel to make **subsequent builds faster.**
  + Used only during **local development** to improve developer experience.
  + Should also be added to **.gitignore**, because it is temporary and can be safely regenerated.

***one-liner -***

* dist - Production-ready build output 🚀
* .cache - Development-time speed booster ⚡

Parcel features -

1. **HMR (Hot Module Replacement):** Updates modules instantly without a full reload, keeping the app state intact
2. **File Watcher Algorithm:** A C++ algorithm that tracks file changes and alerts the server for reloads.
3. **Minification:** Removes unnecessary characters from code to reduce size and make it more efficient for production.
4. **Code Clean-up:** Removes unused code and keeps the codebase organized for better efficiency.
5. **Development and Production Builds:** Provides separate configurations for dev and prod to ensure faster builds during development and optimized performance in production.
6. **Super-Fast Build Algorithm:** Optimizes the build process for quicker build times.
7. **Image Optimization:** Images can be quite large when loaded into the DOM, Parcel performs image optimization to reduce their size. Automatically optimizes images in production mode to reduce file size without losing quality.
8. **Caching During Development:** Stores build output in cache to make future builds faster and smoother for developers.
9. **File Compression:** Optimizes files by shortening variable names and removing redundancies to reduce overall bundle size.
10. **Compatibility with Older Browsers:** Ensures the application runs smoothly on legacy browsers by providing backward support.
11. **HTTPS Support in Development:** Enables local testing over HTTPS, unlocking features that don’t work with HTTP.
12. **Automatic Port Number Handling:** Assigns a new port automatically if the default one is already occupied.
13. **Consistent Hashing Algorithm:** Ensures efficient and reliable bundling by generating consistent file hashes for caching.
14. **Zero Configuration Requirement:** Parcel works out-of-the-box without extra setup or third-party tools.
15. **Automatic Code Splitting:** Splits code into smaller bundles automatically for faster loading times.
16. **Built-in Live Server:** Comes with a live-reloading server to instantly reflect code changes during development.
17. **Tree Shaking:** Removes unused or dead code during bundling to reduce final bundle size.

### **What is Tree Shaking?**

**Tree Shaking** is a process that removes code from your project that you aren't using. It's like cleaning up your house by getting rid of things you don't need. In JavaScript development, tree shaking helps remove unused functions and variables from the final code bundle, making the application faster and smaller.

### **How Does Tree Shaking Work?**

Imagine you import a library into your project but only use a few functions from it.  
Without tree shaking, the final code bundle would still contain all the unused functions, making it unnecessarily large.

Tree shaking solves this by analysing the code and keeping only the parts that are actually used, while discarding the rest.

Benefits of Tree Shaking -

* **Smaller Bundle Size** - Removes unused code, making the final JavaScript file lighter and reducing load times.
* **Better Performance -** Less code to parse and execute, so the app runs faster and feels smoother.
* **Cleaner Codebase** - Keeps the bundle free from unnecessary functions or logic, making the project easier to maintain.

**Example -** Let’s say you have a utility file with several functions, but you only use one of them in your project.

#### **Code Before Tree Shaking** -

// utils.js

export function add(a, b) { return a + b; }

export function subtract(a, b) { return a - b; }

export function multiply(a, b) { return a \* b; }

export function divide(a, b) { return a / b; }

// app.js

import { add } from './utils'; // Only using `add`

console.log(add(5, 3)); // Output: 8

In this case, you’re **only using the** add **function,** but **all** four functions from utils.js will be included in your final code bundle without tree shaking.

#### **Code After Tree Shaking**:

With tree shaking, only the add function would be kept in the final bundle, and the unused subtract, multiply, and divide functions would be removed, resulting in a **smaller file.**

### **How It Works in Practice**:

* **Step 1**: You import a library (like utils.js).
* **Step 2:** Tree shaking looks at your code and sees that only the add function is used.
* **Step 3:** Tree shaking removes the subtract, multiply, and divide functions.
* **Step 4:** The final bundle is smaller and more efficient because it only includes the code you actually need.

We have created our own create-react-app.