# Chapter 2 – Igniting our App

When we build an application, making it ready for production means we need to do a lot of optimizations like cleaning up console logs, shrinking and compressing the code, combining files, and setting up caching. If the app uses many images, we also need to optimize those. So overall, getting an app production-ready takes quite a bit of effort.

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.

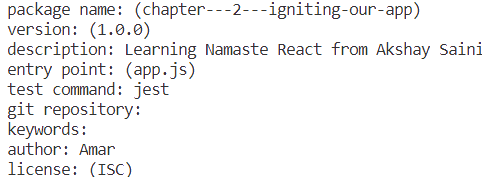
Creating our own create-react-app

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:

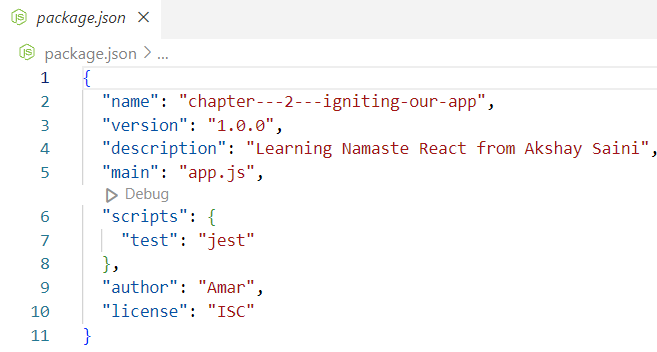


Once we answer those, NPM creates a file called package.json.  
This file stores all the important information about our project — including the list of dependencies we install later.

**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 also does several **optimizations** to improve performance, such as:

* **Code splitting** (breaking the code into smaller parts to load faster)
* **Chunking** (splitting large files into smaller ones)
* **Image optimization**
* **Code compression** (making the code smaller)
* **Tree shaking** (removing unused code)
* **Caching** and more

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:

"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 called **version prefixes** and control **how much your package can be 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 contains all the necessary details to rebuild the node\_modules folder the same way every time. This ensures that the project is consistent and runs smoothly across different developers’ environments.

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:

1. 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, and the package will be considered invalid.
2. Verification: When you install a package, the SHA key is used to verify that the package you're downloading is exactly the same as the one that was originally added to your project, ensuring you get the exact version you need.
3. Security: By using the SHA key, you can trust that the package hasn't been compromised, making the project more secure.

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** specifies the dependencies and their version ranges (e.g., ^, ~, exact), while **package-lock.json** locks the exact versions of those dependencies to ensure everyone gets the same versions when they install the project.

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

The package.json file is created when you run npm init and contains information about the project's dependencies and version ranges. It defines what packages your project needs and allows flexibility in versions.

The package-lock.json file is created or updated whenever packages are installed. It locks the exact versions of the installed dependencies, ensuring consistency across all environments by keeping track of the precise versions used in the project, regardless of the version ranges defined in package.json.

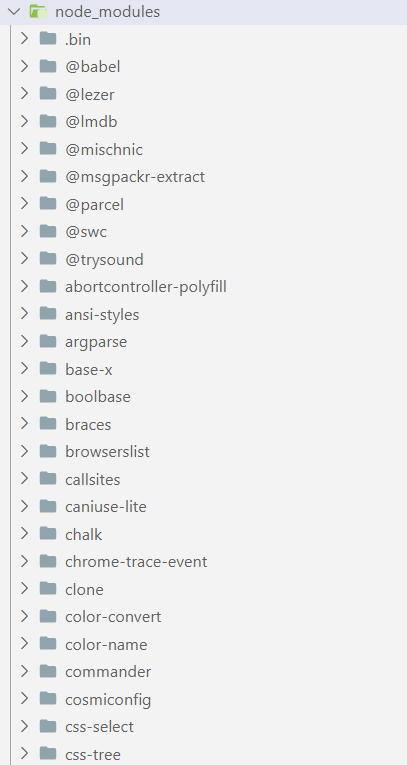
Why should I not modify `package-lock.json`?

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.

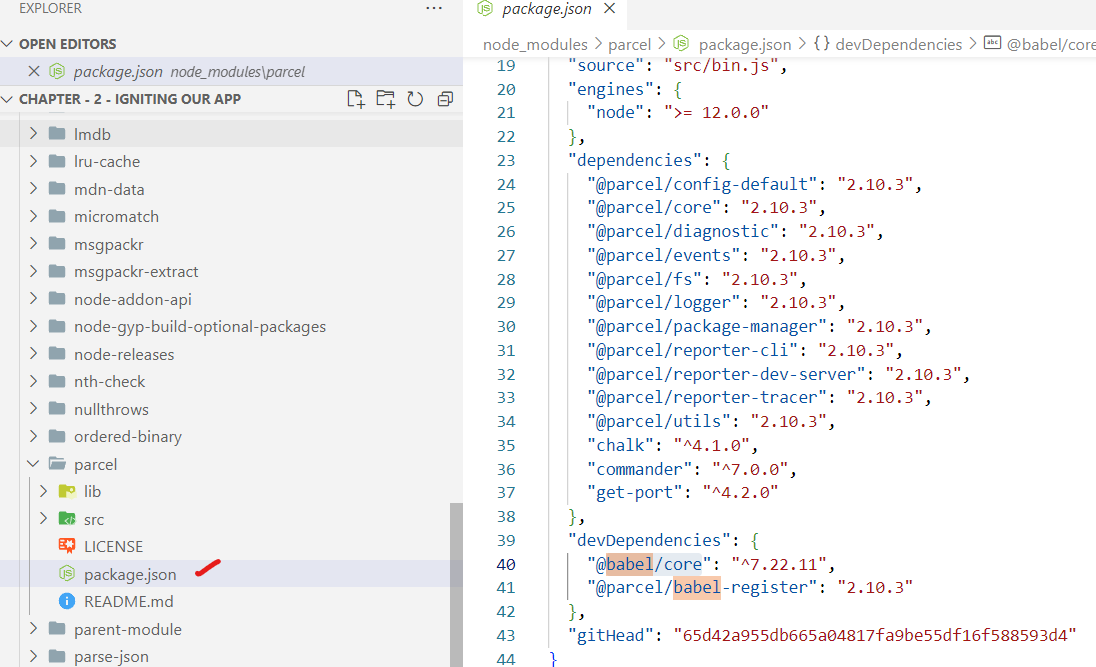
What are node modules? What is Transitive depenancy?

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 to the .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. 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 **development dependencies** (needed only during development) for that package. Essentially, a package's **package.json** provides information about its **transitive dependencies**, which are the dependencies that the package relies on.



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

The **.gitignore** file is a text file that instructs Git to ignore certain files or directories when committing to a repository. It's used to ensure that unnecessary or sensitive files are not tracked by version control. Typically, autogenerated files, such as build outputs and dependencies, should be added to the .gitignore file. Sensitive information like security keys and API keys should also be ignored for security reasons.

Here are some common conventions used in the .gitignore file:

* \* is a wildcard that matches any characters.
* / is used to ignore pathnames relative to the .gitignore file.
* # is used to add comments.

### Example - .gitignore

# Ignore node\_modules folder

/node\_modules

# Ignore all text files

\*.txt

# Ignore files related to API keys

.env

# Ignore SASS config files

.sass-cache

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

package.json & package-lock.json files should not be added to .gitignore. 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 will access the entry point (index.html) and create a development build of our app. It will then serve the app on a local server at [**http://localhost:1234/**](http://localhost:1234/), making it accessible in the browser.

### 🔁 Difference Between npm and npx

**npm (Node Package Manager)**

Used to **install** packages into your project.  
👉 Example: npm install parcel

**npx (Node Package Execute)**

Used to **execute** a package **without installing** it globally.  
👉 Example: npx parcel index.html

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

📝 in short:

* Use npm to **install**
* Use npx to **run**

Why CDN Links Are Not Recommended for React and ReactDOM?

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

#### 1. **Extra Network Calls**

When the browser sees a script tag with a CDN link like:

https://unpkg.com/react@18/umd/react.development.js

https://unpkg.com/react-dom@18/umd/react-dom.development.js

…it has to make a **network request** to fetch these files. This can slow down your app during development or in production.

#### 2. **You Already Have Local Copies**

If you've installed React and ReactDOM using npm, they already exist in your node\_modules folder. There's **no need to fetch them from the internet** — your project can use the local versions instead.

#### 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

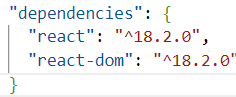
…and everything is managed automatically in package.json.

✅ Better Way: Install React Locally

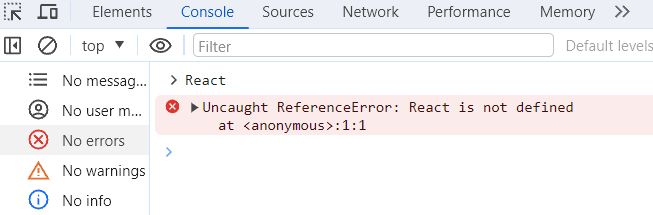
npm install react

npm install react-dom

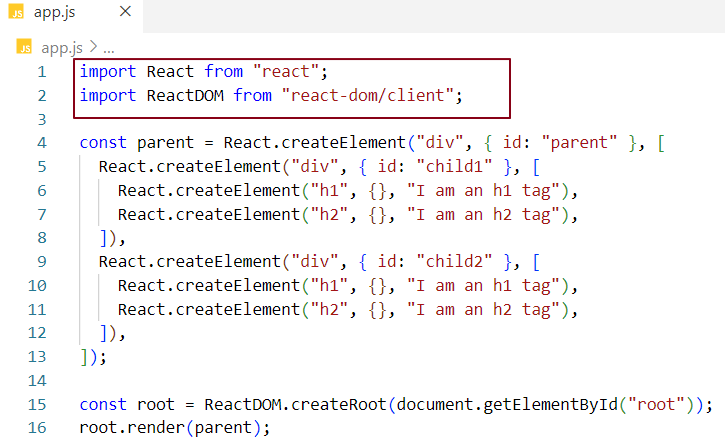
These are **normal (runtime) dependencies**, not dev dependencies — because they are needed when your app runs in production too.



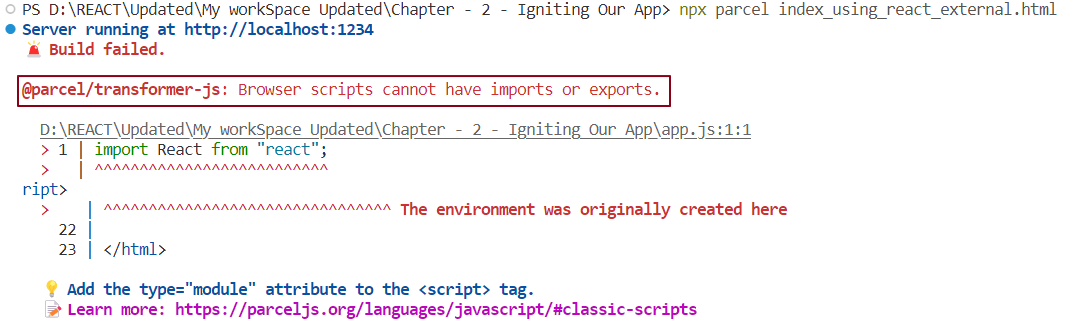
When we run the application, we encounter an error that says 'React is not defined.



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 (found in the node\_modules folder). Similarly, we should import ReactDOM from react-dom/client to fully use 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. So, 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 lets the browser 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 the page upon saving, showing the updated content in the browser. This happens thanks to Parcel, which takes care of Hot Module Replacement (HMR) behind the scenes.

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

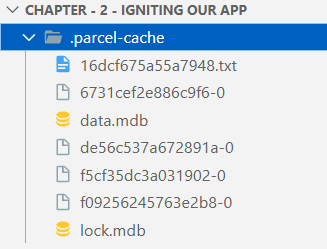
**Hot Module Replacement (HMR)** is a feature that allows you to update your application in real-time as you make changes to the code — without needing to refresh the entire browser page.

How it works:

* **HMR Meaning** - HMR enables 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 alerts the Parcel development server.
* **Parcel’s Role**-Parcel receives the update and injects the modified module directly into the running app. This avoids a full page reload and helps retain application state — making development faster and smoother.

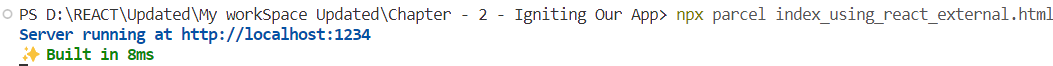
What is Caching? Where parcel stores caches?

**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 and an enhanced developer experience.









Images can be quite large when loaded into the DOM, but Parcel also performs image optimization to reduce their size. In addition to this, Parcel handles tasks like code compression, minification, and bundling to improve overall performance.

While React plays a key role in boosting application performance, other tools like bundlers and Babel are also crucial for achieving optimal speed. Parcel enhances application performance by employing techniques such as consistent hashing, code splitting, differential bundling (for supporting older browsers), diagnostics for displaying errors, and tree shaking.

Furthermore, Parcel provides HTTPS support for securely testing the application and automatically manages the port number for convenience.

### **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 you only use a few functions from it. Without tree shaking, even the functions you don’t use will still be included in your project’s final code bundle.

Tree shaking solves this problem. It analyses the code and only includes the functions or pieces of code that are being used, leaving out the rest.

### **How It Helps:**

* **Smaller Bundle Size:** By removing unused code, tree shaking reduces the size of the final JavaScript file, which means faster loading times for the website.
* **Better Performance:** With less code to load and run, the app performs faster, improving the user experience.
* **Cleaner Code:** Tree shaking helps keep your codebase clean and free from unnecessary pieces of code.

**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**:

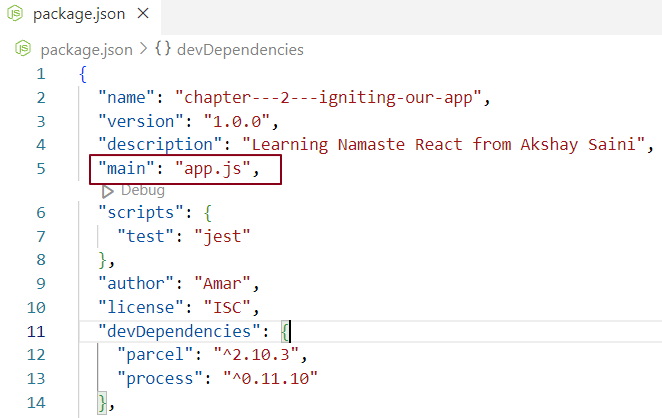
* **Step 1**: You import a library (like utils.js).
* **Step 2:** Tree shaking looks at your code and sees that only the add function is used.
* **Step 3:** Tree shaking removes the subtract, multiply, and divide functions.
* **Step 4:** The final bundle is smaller and more efficient because it only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

Parcel features

1. **HMR (Hot Module Replacement)**: Allows updates to modules in a running application without a full page reload, preserving the application state.
2. **File Watcher Algorithm**: Implemented in C++, this algorithm tracks file changes and notifies the server for reloads.
3. **Minification**: Reduces code size by removing unnecessary characters, making it more efficient for production.
4. **Code Clean-up**: Ensures the codebase remains organized and free of unused code.
5. **Development and Production Builds**: Supports different configurations for development and production environments to optimize performance.
6. **Super-Fast Build Algorithm**: Optimizes the build process for quicker build times.
7. **Image Optimization**: Automatically optimizes images in production mode to reduce file size without losing quality.
8. **Caching During Development**: Uses cached files to speed up subsequent builds, improving the developer experience.
9. **File Compression**: Shortens variable names and performs other optimizations to minimize file size.
10. **Compatibility with Older Browsers**: Ensures applications can run on legacy browser versions.
11. **HTTPS Support in Development**: Allows testing with HTTPS locally, enabling certain features that are unavailable over HTTP.
12. **Automatic Port Number Handling**: Automatically assigns a new port number if the default one is in use.
13. **Consistent Hashing Algorithm**: Utilizes a consistent hashing algorithm for efficient and reliable bundling.
14. **Zero Configuration Requirement**: No need for additional setup or third-party tools to get started with Parcel.
15. **Automatic Code Splitting**: Dynamically separates code into bundles for improved loading times.
16. **Built-in Live Server**: Provides a server with live reloading, enhancing the development workflow.
17. **Tree Shaking**: Eliminates unused code during bundling to reduce file size.

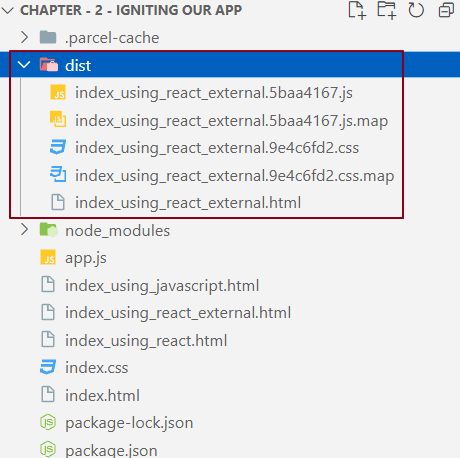
How to create a production build?

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?

When the bundler builds the application, the output is placed in a folder called dist. The /dist folder contains the minified and optimized version of the source code, which is used for production web applications. In addition to the minified code, the /dist folder also includes all the compiled modules, which may or may not be used with other systems.



**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 allows you to specify target browsers using queries in a configuration file, which tools like **Babel** and **Autoprefixer** use to adjust code compatibility.

{

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

}

This query means:

* Support browsers with more than 1% market share.
* Support the last 2 versions of all browsers.
* Exclude unsupported (dead) browsers.
* Include Firefox Extended Support Release (ESR).

This doesn't mean that our app will only support the latest version of Firefox. Instead, it means that our app will work seamlessly with the latest versions of Firefox

Difference between dist and cache Folder?

* The dist folder is where Parcel outputs the build artifacts, such as bundled and minified files for your application. You typically do not want to include the dist folder in version control because it contains generated output that can be recreated from your source files. Instead, add this folder to the .gitignore file.
* Parcel uses the **.**cache folder to store cached data, which speeds up subsequent builds. This cache is specific to local development and can be regenerated, so it should also be added to the .gitignore file.

We have created our own create-react-app.