React elements are plain JavaScript objects.

root.render() takes this **React element object** and **converts it into real DOM nodes** (actual HTML tags).

Before render

React Element = **object**

`{ type: "h1", props: { children: "Hello" } }`

After render

Browser creates:

`<h1>Hello</h1>`

Difference Between Git and GitHub

Git is a local version control tool;

GitHub is an online service to host and share Git repositories.

Git = keeps code history on your laptop

GitHub = keeps that code online so you can share it

⭐ Git → Software on your laptop

- Helps you save versions of your code.
- Like a “history” system for your project.
- Works without internet.

Think of Git like:
➡️ *A save button + history for your code.*

⭐ GitHub → Website

- A place on the internet where you upload your Git projects.
- Helps you share code with others.
- Works only with internet.

Think of GitHub like:
➡️ *Google Drive for Git projects.*

npm is NOT "Node Package Manager"

People think npm means "Node Package Manager,"
but the npm team officially said it doesn't stand for anything now.

What npm actually does (simple)

- ***npm is a tool that manages packages** for Node and JavaScript projects.*
- *It keeps all packages in one place (the **npm registry**).*
- *It downloads, installs, updates, removes packages.*
- *It manages your project dependencies using **package.json**.*

What is package.json?

package.json is a configuration file for your project.
It tells npm what packages your project uses and which versions.

bundler

Your React project has many separate files:

- App.js
- Header.js
- Footer.js
- CSS files

- Images
- JSX
- Imports
- ES6 code

The browser **cannot understand** these separate React files.

So…

Webpack mixes everything into ONE final file

like fruit salad 👉 **“bundle.js”**
which the browser can understand.

Webpack is **one bundler**.

Create React App

When you use **create-react-app**, it automatically uses **Webpack** behind the scenes.

You don’t see it…
but it is doing the mixing (bundling) for you.

Two types of dependencies

Normal dependency (production dependency)
Dev dependency (only needed while developing)

npm install parcel

This installs Parcel as a **normal dependency**.

Meaning:

- It will be included in "dependencies" in package.json
- It is needed in production also
- Example: axios, react, express

But **Parcel is not used in production**… so this is NOT what we want.

npm install -D parcel OR npm install --save-dev parcel

This installs Parcel as a **dev dependency**.

Meaning:

- Added under "devDependencies"
- Only used when developing
- Not needed when the app is running live

This is correct because bundlers (Parcel, Webpack, Vite) are **development tools**.

What's the difference between tilde(~) and caret(^) in package.json?

~ = allow only patch (small) updates

^ = allow minor + patch (bigger) updates

npm updates automatically ONLY within these limits

Why do we need ~ and ^ ?

Because when a **new version** of a package comes,
npm needs to know:

❓ “Should I update automatically... or not?”

Some people want:

- only **small safe updates**
- some want **medium updates**
- some want **all updates**

So **~ and ^** tell npm how much update is allowed.

That's it.
They are **just rules for auto-update**.

Is *integrity* added automatically in package.json

You never write it manually — npm generates it automatically when you install a package.

When npm downloads a package, it also stores a **hash** (like a fingerprint) of the package in package-lock.json.

This fingerprint = integrity value
(example: "sha512-30iZtAPgz...")

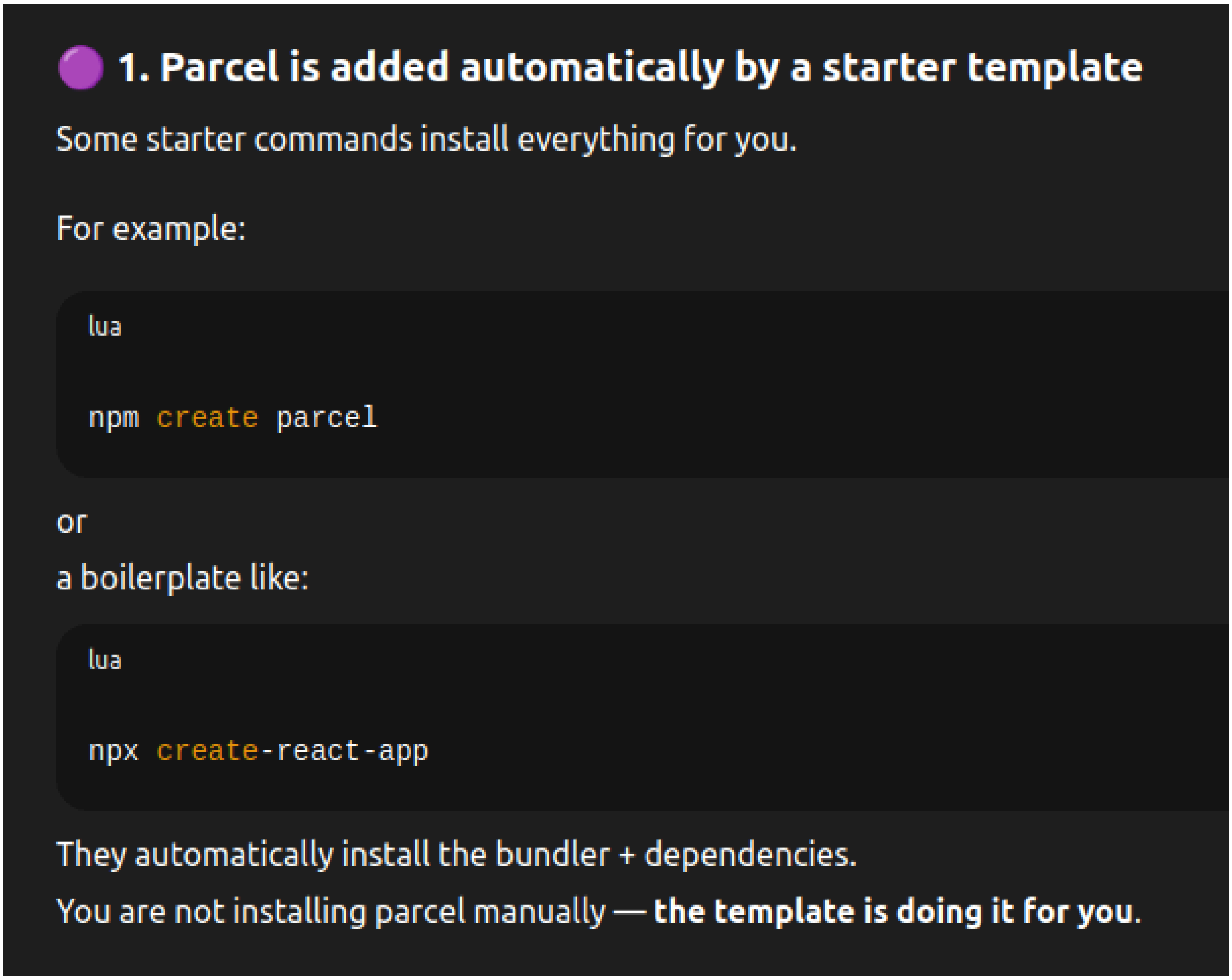
Later, when installing again:

- npm checks the package
- compares with integrity value
- if anything is changed → npm will **reject** it

So no one can inject bad code into your package.

Parcel

Parcel is in node_modules because npm installs both Parcel and all the packages Parcel depends on. Starter templates may install it automatically even if you didn't manually install it



🟣 **1. Parcel is added automatically by a starter template**

Some starter commands install everything for you.

For example:

```lua
npm create parcel
```

or
a boilerplate like:

```lua
npx create-react-app
```

They automatically install the bundler + dependencies.
You are not installing parcel manually — **the template is doing it for you.**

How does npm know that Parcel has its own dependencies?

Parcel tells npm what packages it needs through its own package.json. npm reads that file and installs everything automatically.

You say:
👉 **“npm, bring Parcel.”**

Parcel says:
👉 **“Okay, but I need 50 friends to work.”**

npm says:
👉 **“Fine, I will bring your 50 friends also.”**

So npm installs:

- **Parcel**
- **Parcel's friends (its dependencies)**
- **Those friends' friends (more dependencies)**

Everything goes inside node_modules.

whole project, how many package.json and package-lock.json exist?

Your project has:

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

node_modules has:

- **hundreds of package.json files** (one for each dependency)

Should you put node_modules on Git or Production?

node_modules should NEVER go to Git. It should be in .gitignore because npm can recreate it anytime using package.json and package-lock.json.

npm vs npx

npm = download the app

npx = open the app immediately without downloading

React comes from node_modules

When you write:

```
import React from "react"
```

React is loaded from:

```
your-project/node_modules/react/
```

Why <script src="App.js"> gives import/export error ?

❌ Browsers cannot understand import/export in normal scripts.

```
import React from "react";
```

Browser will say:

“Unexpected token import”

Why?
Because browser treats it as **old JavaScript**, not module JavaScript.

```
<script type="module" src="App.js"></script>
```

This tells the browser:

“This JS file uses modern features like import/export.”

Now the browser loads it correctly.

Why webpage updates automatically when we save the file?

Because **Parcel uses HMR (Hot Module Replacement)**.

✓ What HMR does?

- Watches your files (file-watching algorithm in C++)
- Detects changes instantly
- Sends updates to the browser without full reload
- Fast development experience

● 1. File Watching (C++ algorithm)

Detects whenever you save a file → triggers rebuild.

● 2. HMR (Hot Module Replacement)

Updates the browser without full reload.

● 3. Caching

- Stores previous build results
- Makes rebuilds extremely fast

● 4. Image Optimization

- Compresses & optimizes images
- Because **loading images is the most expensive work for browsers**

● 5. Minification

- Removes spaces, comments
- Makes JS/CSS smaller

● 6. Bundling

- Combines many JS files into fewer files

● 7. Compression

- Gzip / Brotli → smaller files for network

● 8. Local Dev Server

- Runs your app at localhost
- Auto refreshes with HMR

● 9. Consistent Hashing

- Adds hash in file names for caching:

Browserslist in package.json

Browserslist tells bundlers how old or modern the target browsers are, so they can convert your code to work everywhere.

```
"browserslist": [
  "last 2 versions"
]
```