**Ep02: Igniting our app**

**What is React.createElement()**

createElement lets you create a React element. It serves as an alternative to writing JSX.

const element = React.createElement(type, props, ...children)

const element = React.createElement("div", {classname=”divClass”}, [heading1, heading2]);

=>tag Name, attributes, children

Parameters

type: The type argument must be a valid React component type. For example, it could be a tag name string (such as 'div' or 'span'), or a React component (a function, a class, or a special component like Fragment).

* **props**: The props argument must either be an object or null. If you pass null, it will be treated the same as an empty object. React will create an element with props matching the props you have passed. Note that ref and key from your props object are special and will not be available as element.props.ref and element.props.key on the returned element. They will be available as element.ref and element.key.
* **optional ...children:** Zero or more child nodes. They can be any React nodes, including React elements, strings, numbers, portals, empty nodes (null, undefined, true, and false), and arrays of React nodes.

**Returns**

createElement returns a React element object with a few properties:

* type: The type you have passed.
* props: The props you have passed except for ref and key. If the type is a component with legacy type.defaultProps, then any missing or undefined props will get the values from type.defaultProps.
* ref: The ref you have passed. If missing, null.
* key: The key you have passed, coerced to a string. If missing, null.

Usually, you’ll return the element from your component or make it a child of another element. Although you may read the element’s properties, it’s best to treat every element as opaque after it’s created, and only render it.

Caveats

-You must treat React elements and their props as immutable and never change their contents after creation. In development, React will freeze the returned element and its props property shallowly to enforce this.

-When you use JSX, you must start a tag with a capital letter to render your own custom component. In other words, <Something /> is equivalent to createElement(Something), but <something /> (lowercase) is equivalent to createElement('something') (note it’s a string, so it will be treated as a built-in HTML tag).

-You should only pass children as multiple arguments to createElement if they are all statically known, like createElement('h1', {}, child1, child2, child3). If your children are dynamic, pass the entire array as the third argument: createElement('ul', {}, listItems). This ensures that React will warn you about missing keys for any dynamic lists. For static lists this is not necessary because they never reorder.

**Bundlers**

What Are JavaScript Bundlers?

A Bundler is a tool that looks at your entire application, which at scale can be composed of sub-projects and rely on many third-party dependencies, and it will bundle everything into a minimal set of files that can be deployed to production, often a single .js file to optimize performance for the client.

Bundlers also allow for setting up different environments, so you can have config files changing based on whether you’re testing locally or whether the application is in staging/pre-production, or production.

e.g. Webpack, parcel, vite

**What is Parcel and use**

parcel is like an package. For example, package in JS code. whenever we need to install any package to our code then we need a package manager. We will be using package manger named as NPM.

NPM we will use to manage our packages.

**Why do we use npm?**

Because we want lot of packages in our project. Our react app is powered by lot of things and cannot be build by just injecting react into our webpage.

To get the npm in our project we need to use below command which will initialise the npm

npm init -y

here -y will skip lot of options like

-package name

-version

-description

-entry point(of the project)

-git repository

-author name

...etc

After doing this we will get the package.json file. This is the configuration file which npm needs to run itself.

This file will contain all the details we gave in npm init command, in case of -y, npm generated information will be available here.

After initialising the npm to start our app we will use parcel.

To get the parcel by npm use below command.

npm install -D parcel

Here npm install is the command to install any package followed by the package name. and we have used -D as we want install in development machine(-D means dev dependency, it is also same as –save-dev), in case of prod it will be like below.

npm install parcel

After doing this we will get the package.json config file

**Difference between tilde ( ~ ) and caret ( ^ ) in package.json**

~version “Approximately equivalent to version”, will update you to all future patch versions, without incrementing the minor version. ~1.2.3 will use releases from 1.2.3 to <1.3.0.

^version “Compatible with version”, will update you to all future minor/patch versions, without incrementing the major version. ^2.3.4 will use releases from 2.3.4 to <3.0.0.

If we are not giving either ~ or ^ then update won’t happen automatically.

**What is package-lock.json?**

package-lock.json is a lockfile that contains information about the dependencies/packages with their exact version numbers (\\*important) that were installed for a node.js project.

It helps different developers working on the same repo to install the exact package versions installed previously, even if the packages have released new versions. This ensures the same node\_modules tree across different machines/environments.

package-lock.json file is essentially used to lock dependencies to a specific version number.

This file is automatically generated (or re-generated) when there is a change in either the node\_modules tree or package.json file.

Whenever we clone a repo and run npm i on a new machine, npm will first look to see if a package-lock.json file is present. If yes, it will proceed by installing the packages given in that file. Otherwise, it will look into the package.json file and start installing the required dependency packages.

package-lock.json is very important file, it locks the version, you never have to keep in .gitignore. if you keep in .gitignore you will face the problem as its working on my machine but not on the server 😊.

The package-lock.json file ensures that the same node\_modules tree is generated every time npm install is run.

One more thing happens after we do install parcel, node\_modules folder will be created. This folder will have all the necessary packages and dependencies that parcel required.

We never have to put or upload the node modules to git or other Version controlling system.

**to ignite or to start the App we need to use below command**

npx parcel index.html

here npx --> Execute using npm and index.html is the starting point.

We need to import the React and the ReactDOM as below.

import React from "react";

import ReactDOM from "react-dom/client";

Whenever we are using the script tag in our html to mention the App.js file we need to mention the type= ”Module” as App.js is not the normal JavaScript file. And if we are not doing so our bundler parcel will give an error at the import statement of App.js.

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

Tasks and duties done by parcel

**-Hot Module Reload**

Hot Module Reloading (HMR) improves the development experience by automatically updating modules in the browser at runtime without needing a whole page refresh. This means that application state can be retained as you change small things. Parcel's HMR implementation supports both JavaScript and CSS assets. As of version 1.12.0, the default implementation has changed to fully refresh the page when files change. You can opt-in to enable true HMR by adding the following in your app. This will only apply in development; HMR is automatically disabled when bundling in production mode.

if (module.hot) {

module.hot.accept()

}

As we save files, Parcel rebuilds what changed and sends an update to any running clients containing the new code. The new code then replaces the old version, and is re-evaluated along with all parents.

For HMR parcel uses something knows as file watcher algorithm, who is continuously checking the file changes after we save the file.

Whenever we use the command npx parcel index.html parcel will take the build and it will host it on our server.

When we want the production build we will execute the below command

npx parcel build index.html

And all the production ready build files will be saved inside the dist folder.

While executing the command we need to be careful about the entry point that we are giving in command and we need to remove the starting point from the package.json file

“main”: “App.js” -> remove this from package.json

-In dist folder we can see only 3 files, index.css, index.js, index.html etc because parcel minifies all the files and bundle together.

-parcel cleans the code by removing the console.log from the code.

-parcel also optimizes the images in the project to run the project fast.

-parcel also do something knows as caching while development

Once we take build, folder named as .parcel-cache will be created and will be use in further build to reduce the build time.

-parcel compresses our code while building, by renaming the functions and variables etc.

-parcel takes care of the older versions of the browsers. compatibility with older browsers.

-we should put .parcel-cache in .gitignore.

**Why? ->** because anything which is autogenerated should be put in .gitignore, anything we can generate on server shall be put in .gitignore.

**How we can get the below folder?**

1. node\_modules –> by using npm install

2. dist –> by taking build (npx parcel stating\_point\_filename)

-Parcel uses something knows as consistent hashing algorithm to hash things up for bundling

-Parcel is zero configuration bundler that need no configuration to work. You just give the entry point it will take care of all the things.

**Difference between devDependencies and normal dependencies**

1. **dependencies**

Dependencies that your project needs to run, like a library that provides functions that you call from your code.

They are installed transitively (if A depends on B depends on C, npm install on A will install B and C).

Example: lodash: your project calls some lodash functions.

1. **devDependencies**

Dependencies you only need during development or releasing, like compilers that take your code and compile it into JavaScript, test frameworks or documentation generators.

They are not installed transitively (if A depends on B dev-depends on C, npm install on A will install B only).

Example: grunt: your project uses grunt to build itself.

1. **Transitivity/Transitive dependency(mentioned above):**

dependencies that are installed transitively: if A requires B, and B requires C, then C gets installed, otherwise, B could not work, and neither would A.

How to make our app compatible with older versions of browser?

To make this happen we have to use something known as browserList.

**How to use browserlist?**

Once we add the browserlist package we can mention about the browsers to be supported, to do that we have to add “browserlist” in package.json file as below.

"browserslist": [

"defaults and supports es6-module",

"maintained node versions"

]

Here in browserlist we are passing an array for the configuration.

**Summary of Parcel**

• HMR: Hot Modul Reloading

• File watcher algorithm (C++)

• Bundling

• Minification

• Cleaning our code

• Dev and Prod build

• Super-fast build algorithm

• Image optimization

• Caching while development

• Compression

• Compatible with older versions of browser

• HTTPS on dev

• Port Number

• Consistent hashing algorithm

• Zero configuration

• Transitive dependencies

Long story short Parcel alone not doing all the stuff rather it architects all the things together by using different packages or node modules required.

\\*Read parcel documentation for more details.