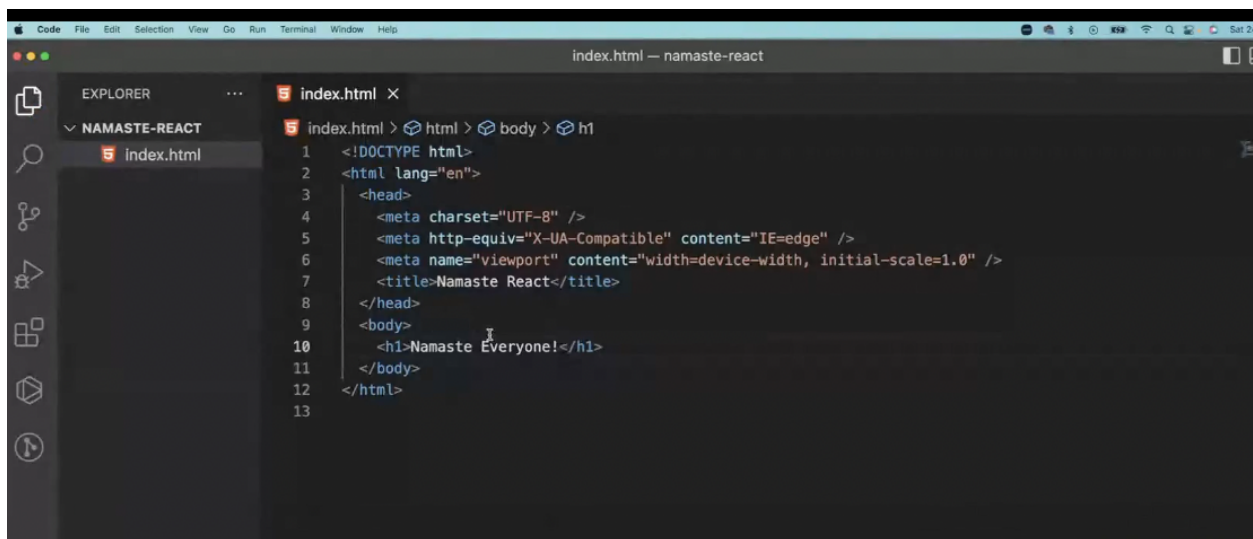


Namaste React

Three ways of writing Hello in Web

1. HTML



```
1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8" />
5     <meta http-equiv="X-UA-Compatible" content="IE=edge" />
6     <meta name="viewport" content="width=device-width, initial-scale=1.0" />
7     <title>Namaste React</title>
8   </head>
9   <body>
10    <h1>Namaste Everyone!</h1>
11  </body>
12 </html>
13
```

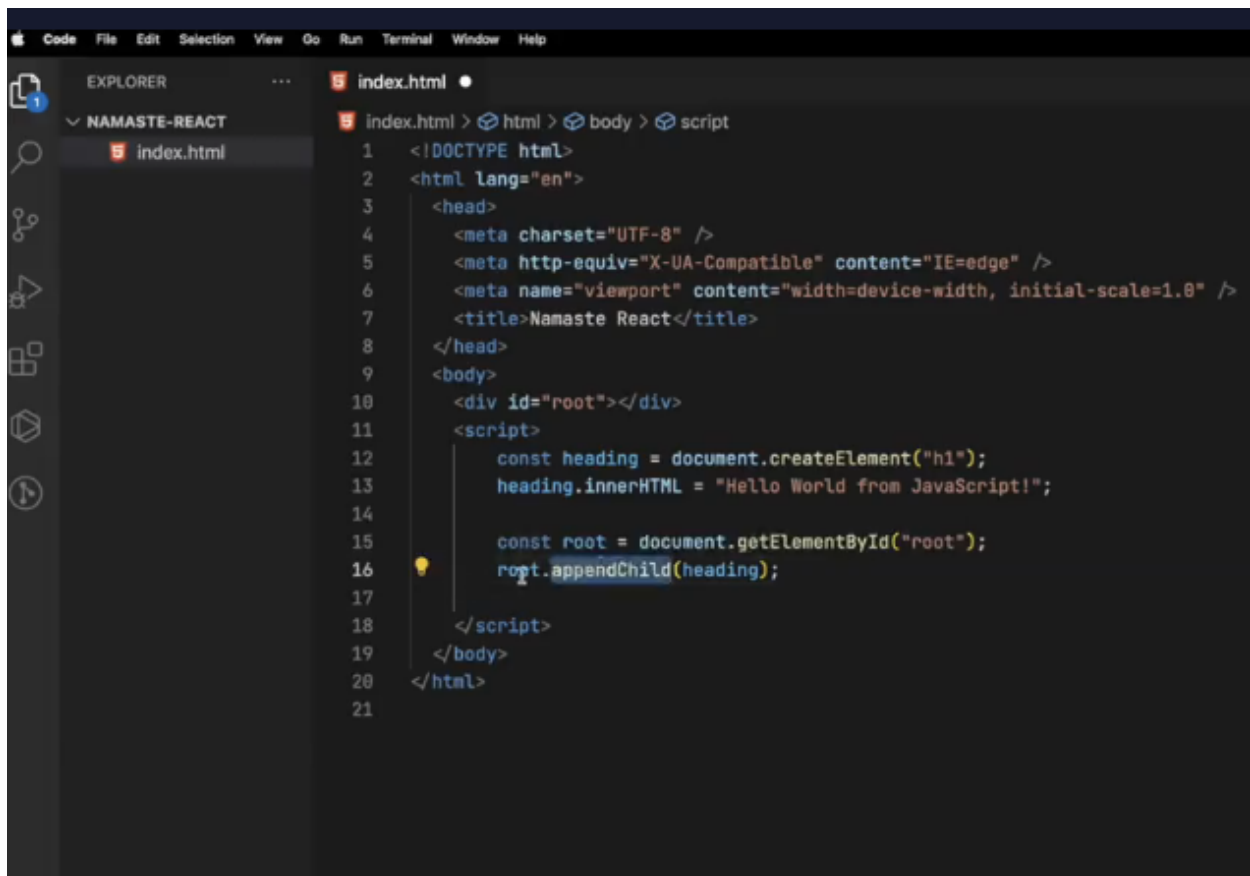
2. Using Javascript

```
index.html > html > script > heading
1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8" />
5     <meta http-equiv="X-UA-Compatible" content="IE=edge" />
6     <meta name="viewport" content="width=device-width, initial-scale=1.0" />
7     <title>Document</title>
8   </head>
9   <body>
10    <div id="root"></div>
11  </body>
12  <script>
13    const heading = document.createElement("h1");
14    heading.innerHTML = "Hello Saurav";
15    document.getElementById("root").append(heading);
16  </script>
17 </html>
18
```

`parentElement.append(childElement1, childElement2, childElement3)`

`parentElement.appendChild(childElement);`

Unlike `append()`, `appendChild()` only accepts a single node object and cannot directly append text nodes or HTML strings. If you want to append multiple nodes using `appendChild()`, you need to call it for each individual node.

A screenshot of a code editor interface. The top menu bar includes 'Code', 'File', 'Edit', 'Selection', 'View', 'Go', 'Run', 'Terminal', 'Window', and 'Help'. The left sidebar shows the 'EXPLORER' view with a folder named 'NAMASTE-REACT' containing a file 'index.html'. The main editor area displays the content of 'index.html'. The file path 'index.html > html > body > script' is shown at the top of the editor. The code is as follows:

```
1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8" />
5     <meta http-equiv="X-UA-Compatible" content="IE=edge" />
6     <meta name="viewport" content="width=device-width, initial-scale=1.0" />
7     <title>Namaste React</title>
8   </head>
9   <body>
10    <div id="root"></div>
11    <script>
12      const heading = document.createElement("h1");
13      heading.innerHTML = "Hello World from JavaScript!";
14
15      const root = document.getElementById("root");
16      root.appendChild(heading);
17    </script>
18  </body>
19 </html>
```

- React is just like any other library, ex- jQuery

CDN

Let's say you're playing a popular online game with players from all over the world. When you're playing the game, you need to send and receive information between your computer and the game's server. This information can include things like your character's movements, actions, and the location of other players.

If the game's server was located in a faraway place, it could take a long time for your computer to send and receive all of this information. This could cause lag or delays in the game, which could make it harder to play and less fun.

But if the game's developer uses a CDN, they can store copies of the game's data on servers located in different parts of the world. When you play the game, your computer

can connect to the server that's closest to you, which can make the game much faster and more responsive.

So, in this example, the CDN acts like a team of delivery people who are stationed in different parts of the world. Instead of having to send and receive information from a faraway place, your computer can communicate with a server that's nearby, which can help the game run more smoothly.

That's a good question! While storing the same file in multiple locations might seem like it would take up a lot of extra space, CDN providers use clever techniques to minimize the amount of space they need.

For example, instead of storing an exact copy of a file on every server, they might store different versions of the file that have been compressed or optimized for different types of devices or network speeds. That way, when someone requests the file, the CDN can automatically deliver the version that's best suited for their device and network.

Additionally, CDN providers also use a technique called caching, which means that when a file is requested, the CDN will store a copy of the file in a temporary storage location that's closer to the person making the request. This means that if someone else requests the same file later, the CDN can deliver it quickly from the cache without having to retrieve it from the original source.

Overall, while it's true that storing files in multiple locations can result in some duplication of data, CDN providers use techniques like compression and caching to minimize the amount of storage space they need while still delivering content quickly and efficiently.

You're correct that if the CDN has a very large number of servers and many copies of the same file are stored on each server, this can result in a significant amount of duplication. However, CDN providers have ways of managing this duplication to ensure that it doesn't become a problem.

One approach is to use a technique called deduplication, which means that if the same file is uploaded to multiple servers, the CDN will only store one copy of the file and create references to it on the other servers. This way, all of the servers can access the file without actually storing multiple copies of it.

Another approach is to use advanced algorithms to determine which servers should store copies of each file based on factors like the popularity of the file, the geographic

location of users, and the type of device they're using. By optimizing the distribution of files in this way, the CDN can minimize duplication and ensure that files are stored in the most efficient way possible.

Overall, while storing files on a large number of servers can potentially lead to duplication, CDN providers have developed sophisticated techniques to manage this and ensure that content is delivered quickly and efficiently.

Content Delivery Networks (CDNs) employ several techniques and algorithms for optimization and storing multiple copies of files. Here are some commonly used ones:

1. **Caching:** CDNs store copies of frequently accessed files on edge servers located closer to end users. When a user requests a file, the CDN delivers it from the nearest edge server, reducing latency and network congestion.
2. **Content Preloading:** CDNs proactively fetch and cache content that is likely to be requested soon based on user behavior and historical data. By preloading popular files onto edge servers, they can further reduce response times.
3. **Load Balancing:** CDNs use load balancing techniques to distribute traffic across multiple edge servers, ensuring optimal performance and preventing any single server from becoming overwhelmed. This involves routing requests to the least busy server or using algorithms like Round Robin, Weighted Round Robin, or Least Connections.
4. **Anycast Routing:** CDNs leverage anycast routing to provide geographically distributed points of presence (PoPs). Anycast allows multiple servers in different locations to share the same IP address. When a user requests a file, the CDN automatically routes the request to the nearest PoP using the shortest network path.
5. **Dynamic Content Optimization:** CDNs have capabilities to optimize dynamic content generation by offloading certain processing tasks from origin servers to edge servers. This reduces the load on the origin servers and improves overall performance.
6. **Deduplication:** CDNs use deduplication techniques to eliminate redundant data. Instead of storing multiple copies of the same file, CDNs identify duplicate content and store a single copy, saving storage space and improving caching efficiency.

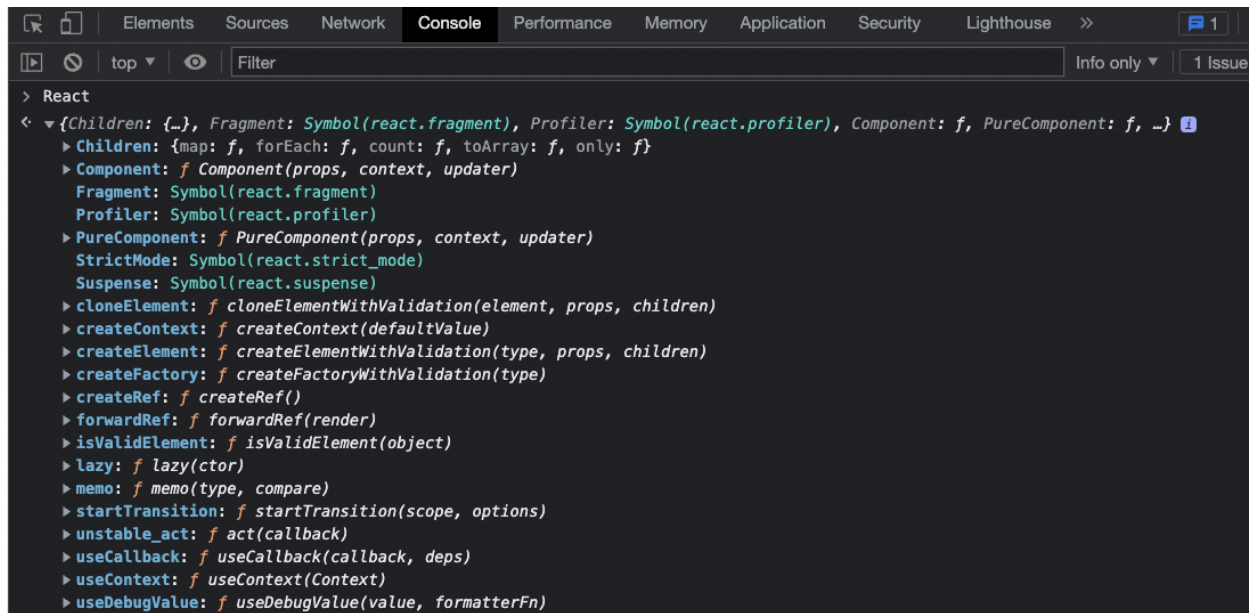
7. **Compression:** CDNs compress files before delivering them to users, reducing the file size and optimizing bandwidth usage. This results in faster downloads and reduced network congestion.
8. **Intelligent Routing:** CDNs continuously monitor the network conditions and performance metrics of their edge servers. They use this information to dynamically route traffic, selecting the most efficient path based on factors like latency, server availability, and network congestion.
9. **SSL/TLS Acceleration:** CDNs optimize the secure delivery of content by offloading SSL/TLS encryption and decryption processes from origin servers. This reduces the computational load on the origin servers and enhances overall performance.

These techniques and algorithms work together to improve content delivery speed, reduce latency, enhance scalability, and ensure a smooth user experience across geographically distributed networks. Different CDNs may employ variations of these techniques based on their specific offerings and optimizations.

3.Using React Library

```
<script
  crossorigin
  src="https://unpkg.com/react@18/umd/react.development.js"
></script>
<script
  crossorigin
  src="https://unpkg.com/react-dom@18/umd/react-dom.development.js"
></script>
```

This is the Shortest code of react. This code we have just injected the code



If u just go to the browser and type react then this will identify react keyword. This way we confirms that react is injected to our code. **React is a global object. Its just a piece of JS Code.**

3.Using React npm

npm install react

Why two React JS Import files ?

```
<script  
  crossorigin  
  src="https://unpkg.com/react@18/umd/react.development.js"  
></script>
```

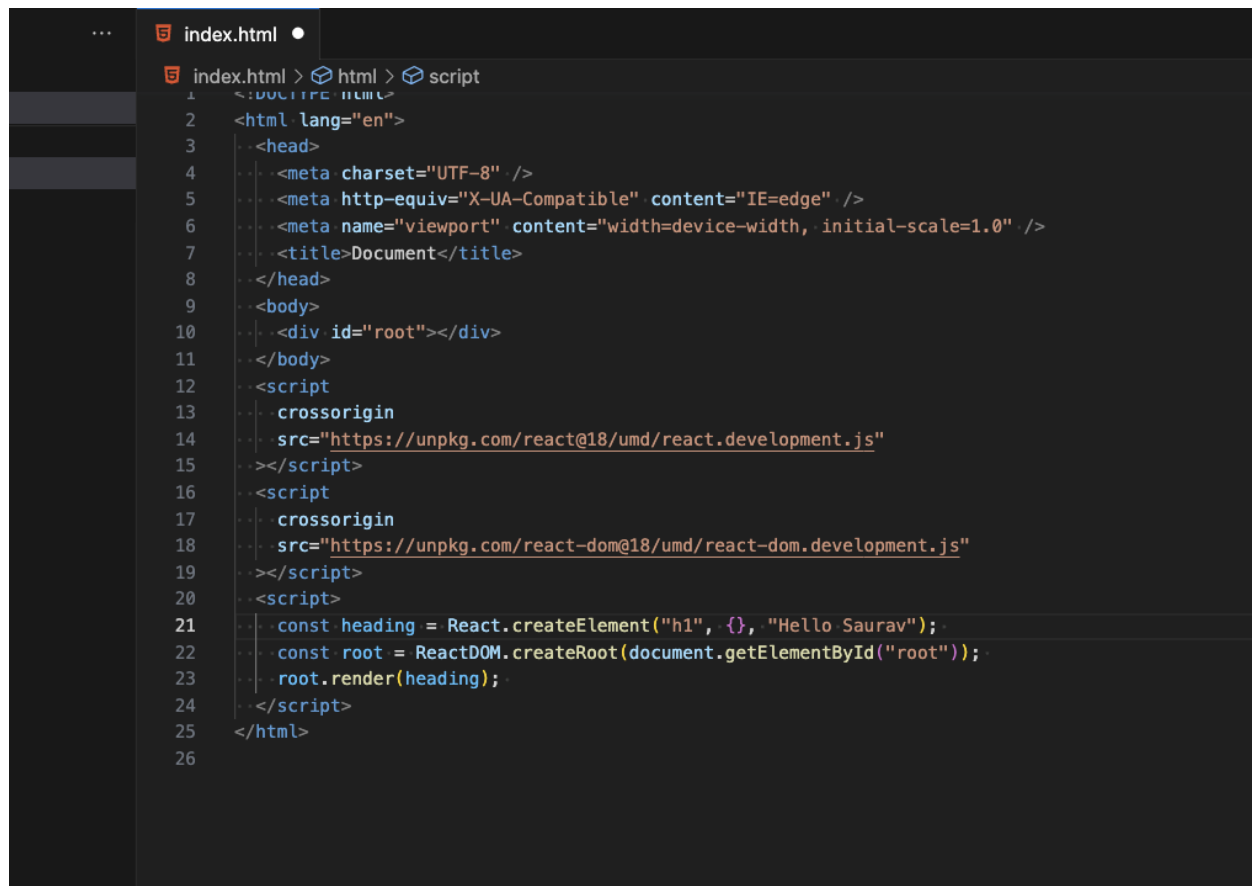
this import consists of basic react 18 version Predefined Functions and other items.

Beause we also have React Native and React 3d so the second file specifies the react-dom. That means the web version of React.

```
<script
  crossorigin
  src="https://unpkg.com/react-dom@18/umd/react-dom.development.js"
>
</script>
```

The first one import is the core library of react and 2nd one gives web functionalty - react dom

Writing code in React



```
index.html
1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8" />
5     <meta http-equiv="X-UA-Compatible" content="IE=edge" />
6     <meta name="viewport" content="width=device-width, initial-scale=1.0" />
7     <title>Document</title>
8   </head>
9   <body>
10    <div id="root"></div>
11  </body>
12  <script
13    crossorigin
14    src="https://unpkg.com/react@18/umd/react.development.js"
15  ></script>
16  <script
17    crossorigin
18    src="https://unpkg.com/react-dom@18/umd/react-dom.development.js"
19  ></script>
20  <script>
21    const heading = React.createElement("h1", {}, "Hello Saurav");
22    const root = ReactDOM.createRoot(document.getElementById("root"));
23    root.render(heading);
24  </script>
25 </html>
26
```


To create element we write

ReactDOM - Reading file from document

```
const heading = React.createElement("h1", {}, "Hello Saurav"); // This is the code  
Library which will be present in every react APP. This line is just creating h1 tag
```

**the root variable is the place where all the core html with id root will be inside it.
everything will be rendered inside it**

To create element we write

ReactDOM - Reading file from document

```
const root = ReactDOM.createRoot(document.getElementById("root")); // ReactDOM  
tells react we are trying to do something in document and trying to modify in web.  
This will just put my h1 into browser. Therefore its DOM Manipulation and it  
comes from 2nd script tag of react react-dom.development.js
```

```
root.render(heading); // to put my heading into the body with div id root. Passing a  
react element in the root
```

```
</script>
```

```
const heading = React.createElement("h1", {}, "Hello Saurav");
```

```
console.log(heading) ⇒ its a object
```

props are children + attributes

this is heading console

Download the React DevTools for a better development experience: <https://reactjs.org/link/react-devtools>
You might need to use a local HTTP server (instead of file:///): <https://reactjs.org/link/react-devtools>

```
▼ {$$typeof: Symbol(react.element), type: 'h1', key: null, ref: null, props: {...}, ...} ⓘ  
  $$typeof: Symbol(react.element)  
  key: null  
  ▼ props:  
    children: "Hello World from React!"  
    id: "props.children"  
    xyz: "abc"  
    ▶ [[Prototype]]: Object  
    ref: null  
    type: "h1"  
    _owner: null  
    ▶ _store: {validated: false}  
    _self: null  
    _source: null  
    ▶ [[Prototype]]: Object
```

Mostly in a React App we will have only one Root and One Render Method. only one render and root

```
<body>  
  <div id="root">  
    <h1 class="title"></h1>  
  </div>  
</body>  
const heading = React.createElement("h1", { id: "title" }, "Hello Saurav");  
console.log(heading);
```

Here h1 will automatically get id as title...whatever we pass in create element 2nd props. We can name our own attribute name ..need not be necessary id or class

It will be accepted as props

The **react.render()** is basically taking the heading object and converting into html format like h1 tag

React give superpower to manipulate DOM using JS using helper functions. The most costly operation in JS is that whenever there is change in any element then the DOM tree updates (putting some nodes inside DOM and removing some)and the framework or library which updates with least operations is the best . All are trying to optimise it

why react over JS ?

Now example i want to create this nested parent child then i can use below code to implement this

```
{
  /* <div id="parent">
    <div id="child">
      <h1>I am child</h1>
    </div>
  </div>; */
}

const parent = React.createElement(
  "div",
  { id: "parent" },
  React.createElement(
    "div",
    { id: "child" },
    React.createElement("h1", { id: "heading" }, "i am heading")
  )
); // for printing i am a child
const root = ReactDOM.createRoot(document.getElementById("root"));

root.render(parent);
```

2nd example when i want multiple heading within same child

```
{
  /* <div id="parent">
    <div id="child">
      <h1>I am heading 1</h1>
      <h2>I am heading 2</h2>
    </div>
  </div>; */
}

const parent = React.createElement(
  "div",
  { id: "parent" },
  React.createElement("div", { id: "child" }, [
    React.createElement("h1", {}, "I am heading 1 "),
    React.createElement("h2", {}, "I am heading 2"),
  ])
);
const root = ReactDOM.createRoot(document.getElementById("root"));

root.render(parent);
```

`console.log(parent)`

```

▼ {$$typeof: Symbol(react.element), type: 'div', key: null, ref: null, props: {...}, ...} ⓘ
  $$typeof: Symbol(react.element)
  key: null
  ▼ props:
    ▼ children:
      $$typeof: Symbol(react.element)
      key: null
      ▼ props:
        ▼ children: Array(2)
          ► 0: {$$typeof: Symbol(react.element), type: 'h1', key: null, ref: null, props: {...}, ...}
          ► 1: {$$typeof: Symbol(react.element), type: 'h2', key: null, ref: null, props: {...}, ...}
          length: 2
          ► [[Prototype]]: Array(0)
          id: "child"
          ► [[Prototype]]: Object
          ref: null
          type: "div"
          _owner: null
          ► _store: {validated: true}
          _self: null
          _source: null
          ► [[Prototype]]: Object
          id: "parent"
          ► [[Prototype]]: Object
          ref: null
          type: "div"
          _owner: null

```

Now imagine if i have more complex structure something like this. This code is core REACT

```

{
  /* <div id="parent">
    <div id="child">
      <h1>I am heading 1</h1>
      <h2>I am heading 2</h2>
    </div>
    <div id="child2">
      <h1>I am heading 1</h1>
      <h2>I am heading 2</h2>
    </div>
  </div>; */
}

```

```

const parent = React.createElement("div", { id: "parent" }, [
  React.createElement("div", { id: "child" }, [
    React.createElement("h1", {}, "I am heading 1 "),
    React.createElement("h2", {}, "I am heading 2"),
  ]),
  ,
  React.createElement("div", { id: "child2" }, [
    React.createElement("h1", {}, "I am heading 1 "),
    React.createElement("h2", {}, "I am heading 2"),
  ]),
]); // for printing i am a child

console.log(parent);
const root = ReactDOM.createRoot(document.getElementById("root"));

root.render(parent);

```

here i have more than one child and the code looks even more complex to read

Hence here comes JSX

we think that using JSX only we can run react but its not true. Now to make rendering creating object more easy and optimise way we use JSX

```

<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
  </head>
  <body>
    <div id="root"></div>
  </body>

  <script src="./script.js"></script>

  <script
    crossorigin
    src="https://unpkg.com/react@18/umd/react.development.js"
  ></script>
  <script

```

```
crossorigin
src="https://unpkg.com/react-dom@18/umd/react-dom.development.js"
></script>

</html>
```

now in this example we defined script.js even before declaring react src. Hence it will throw reference error as its not able to find react code. So Sequence is very important and its from top to bottom

if i write this and then render root.render(parent) then this hello saurav will be replaced by parent element content. For millisec hello saurav will be rendering but once it finds React.render it will replace its content

```
<div id="root">
  <h1>hello saurav</h1>
</div>
</body>
```

React is a library because it can work with the specific section of the code or small portion of our app. where we want it to implement. ex- here we have implemented only in div with id root. so it will be used only for root div not for others. Its not a full fledged **framework**. React is bare minimal JS Library whereas in framework everything will be loaded or will be worked inside it only. Framework will require to create full app using it. But **react can be used with the existing app as well.**

Where as one app build in one framework cannot be applied to another, but react can .React is just giving us helper methods to develop application at end of day its just JS

Ep -2 Igniting our App

npm doesn't stand for node package manager. **NPM doesn't have full form. In background npm acts as node package manager but it doesn't stand for node package manager**

NPM is a standard repository for all the packages. It contains all library and code needed to run

package.json is a configuration for npm. npm is package manager which will manage and install all library and dependencies

BUNDLER - it bundles /minifies/cleans all our html cs js code into single bundle before it goes to production. ex- webpack, parcel, vite (these are bundlers)

When we use create-react-app, the behind the code it install a bundler for us.

For our project we will use parcel library, which will ignite our app as a parcel, now parcel will give muscle to our app. Parcel comes as node package

npm install -D parcel

There are two types of dependencies a app can have

1. **Dev Dependencies** - Dev dependencies are used only when we are developing the app and normal dependencies are used which we are using in production as well
2. **Normal Dependencies**

npm install -D parcel “-D” because we don't need parcel in production as we need to create build from dev dependency only, no need in prod.

After this npm command i got dev dependency of "parcel": "^2.9.3"parcel


```
"devDependencies": {
  "parcel": "^2.9.3"
}
```

Tilde(~) notation	Caret(^) notation
Used for Approximately equivalent to version.	Used for Compatible with version.
It 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.	It 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
It gives you bug fix releases.	It gives you backwards-compatible new functionality as well.
It will update in decimals.	It will update to its latest version in numbers.
Not a default notation used by NPM.	Used by NPM as default notation.
Example: ~1.0.2	Example: ^1.0.2

Tilde (~) Notation:

- Imagine you're buying a specific brand of chocolate, let's say "Chocolate Brand 1.2.3."
- Using Tilde (~) is like saying you want any version of that chocolate that's in the same box (1.2.x).
- So, you're fine with getting versions like 1.2.3, 1.2.4, 1.2.5, but not 1.3.0 or 2.0.0. The major version (the first number) stays the same.

Example: "~1.2.3" means any version from 1.2.3 to 1.2.x but not versions like 1.3.0 or 2.0.0.

Caret (^) Notation:

- Now, think of buying the same chocolate with Caret (^).
- It's like saying you want any version of that chocolate as long as it's still from the same store aisle (1.x.x).

- So, you're fine with 1.2.3, 1.3.0, or even 2.0.0, as long as it's still in the same store aisle (the same major version).

Example: "^1.2.3" means any version from 1.2.3 to 2.0.x but not versions like 3.0.0.

package-lock.json - it keeps track what exact version of npm is install in our system

So basically **package.json** keeps track of approx version of the dependencises and **package.lock.json** keeps track of exact current version installed

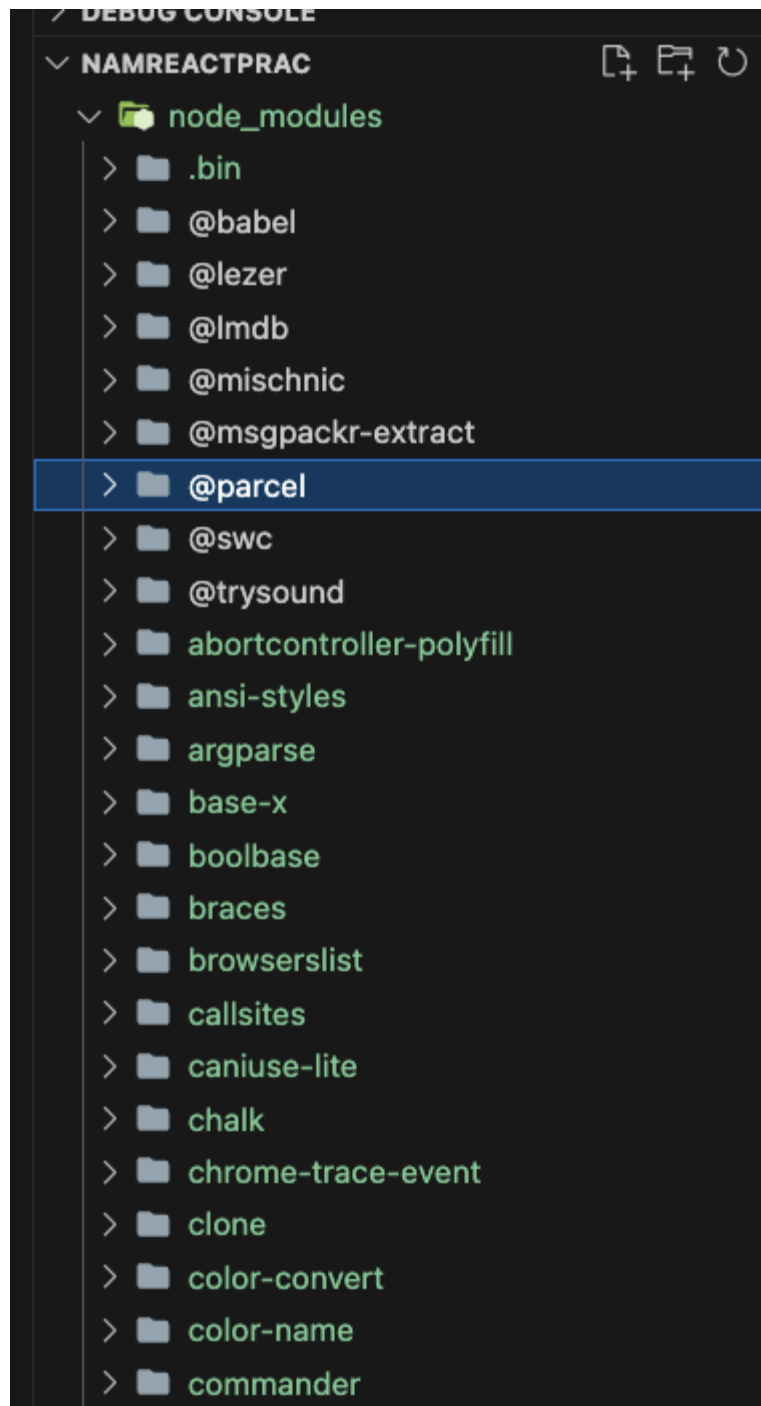
we have something called integrity in lock.json

```
},
"@parcel/diagnostic": {
  "version": "2.9.3",
  "resolved": "https://registry.npmjs.org/@parcel/diagnostic/-/diagnostic-2.9.3.tgz",
  "integrity": "sha512-6jxBdyB3D7gP4iE66ghUGntWt2v64E6EbD4AetZk+hNJpgud00PsKTovcMi/i7I4V0qD7WXSf4tvkZUoac0jwA==",
  "dev": true,
  "requires": {
    "@mischnic/json-sourcemap": "^0.1.0",
    "nullthrows": "^1.1.1"
  }
}
```

This integrity is basically a hash to verify that whatever there is in my local machine its there in production. Because something code break in production and works fine in local . So to avoid this integrity is added

when we did npm install parcel npm was fetching all the code and putting into code modules

node_modules is a kind of database of all models which is installed. So whatever dependencies are there in our packet.json it fetches all the code of it. Like in this case parcel was installed. SO all the algo and codes that are projects need..and are there inside it

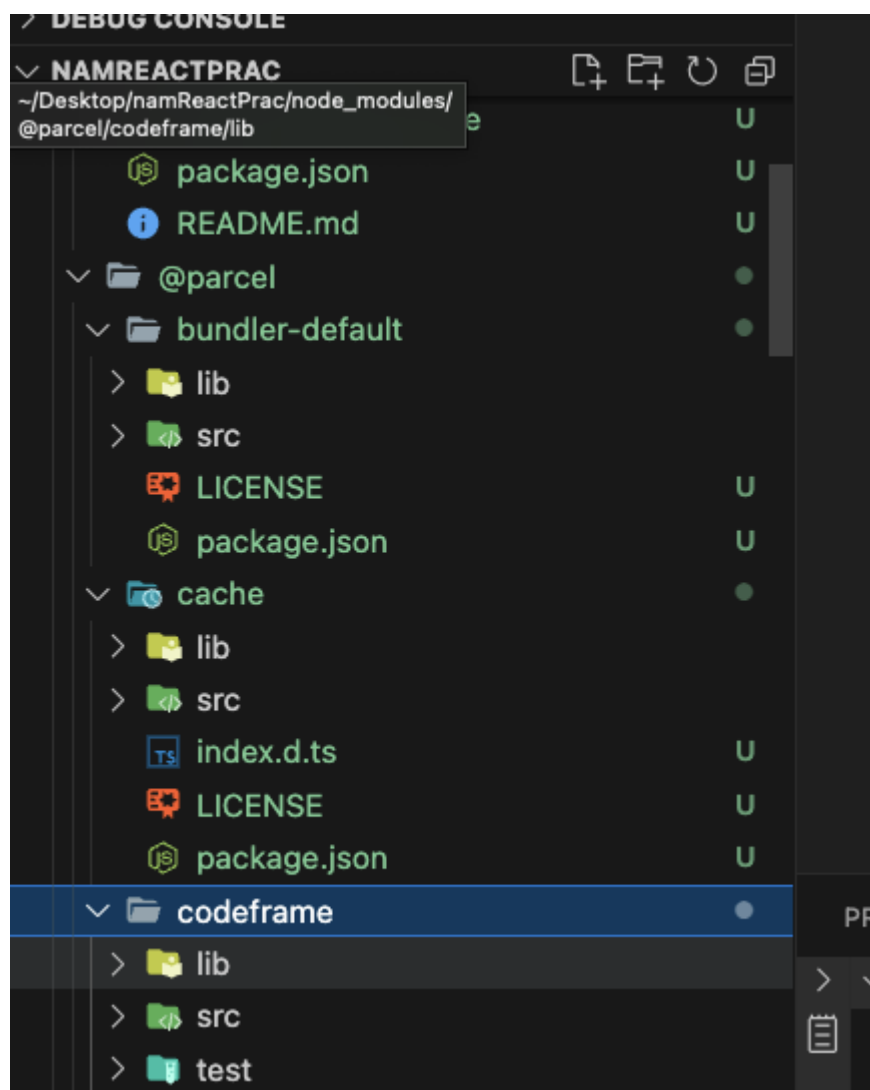


But we only installed parcel then why other files got installed ?

This is because parcel has their own internal dependencies and that dependencies has their own. So this way all other files got generated and this is called **Transitive dependencies**.

Parcel needs help of all other packages thats why it install other libraries.

Also each dependencies has their own package.json file and that json file give list os all dependency of the current library file and it goes on like dependency tree and is cal **Transitive dependencies**.



even if i delete my node module i can regenerate it using package json and lock json..because it contains exact versions to be installed

To install a package we use - npm

To execute a package or library we use -npx

ex- npx parcel index.html

In this command parcel goes to index.html host the development build to localhost:1234

Why to install react using npm than direct cdn ?

```
<script
  crossorigin
  src="https://unpkg.com/react@18/umd/react.development.js"
></script>
```

This is because now we already have the node manager so it will be easy to manage if we just npm it. Also if we use cdn and some new version of react comes in then we have to update it manually.

Now i will install REACT using npm

npm install react

I didnt use -D because i want my react dependency to be in production

now instal REACT - DOM

npm install react-dom to replace this cdn

```
<script
  crossorigin
  src="https://unpkg.com/react-dom@18/umd/react-dom.development.js"
></script>
```

now we dont need cdn links

```
import React from "react";
import ReactDOM from "react-dom";

const parent = React.createElement("div", { id: "parent" }, [
  React.createElement("div", { id: "child" }, [
    React.createElement("h1", {}, "I am heading 1 "),
    React.createElement("h2", {}, "I am heading 2"),
  ]),
]); // for printing i am a child

const root = ReactDOM.createRoot(document.getElementById("root"));

root.render(parent);
```

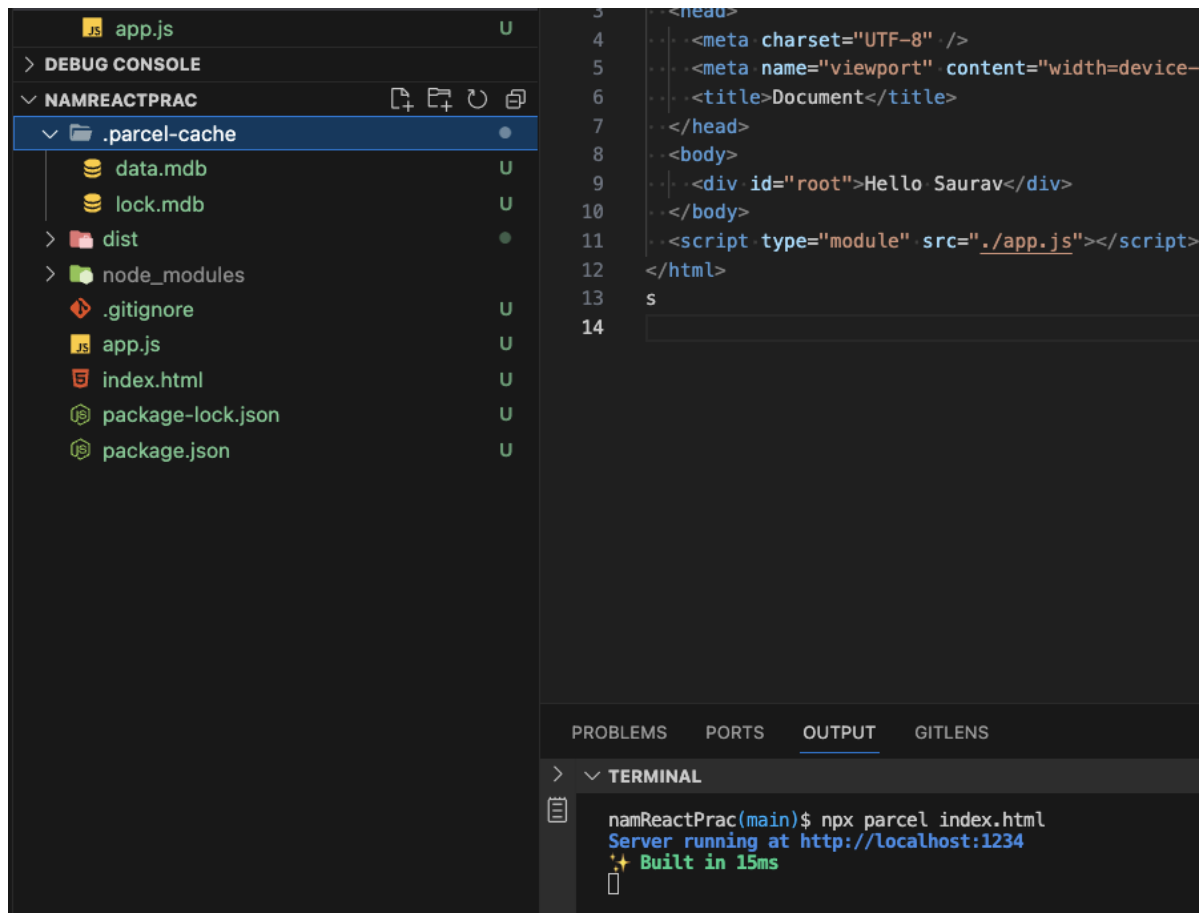
```
<script type="module" src="./app.js"></script>
```

NOTE - we are now installing all manually instead of create-react-app which does all the 3 npm installation at once behind the screen

Function of PARCEL (bundlers) <https://parceljs.org/>

- Dev Build (like production build)
- Setup out local server (ex- localhost:3000)
- doing HMR in all file = hot module replacement (auto refresh the content if changed)
- **HMR** uses file watching algorithm which is written in c++
- it keeps an eye on file change and it build once again

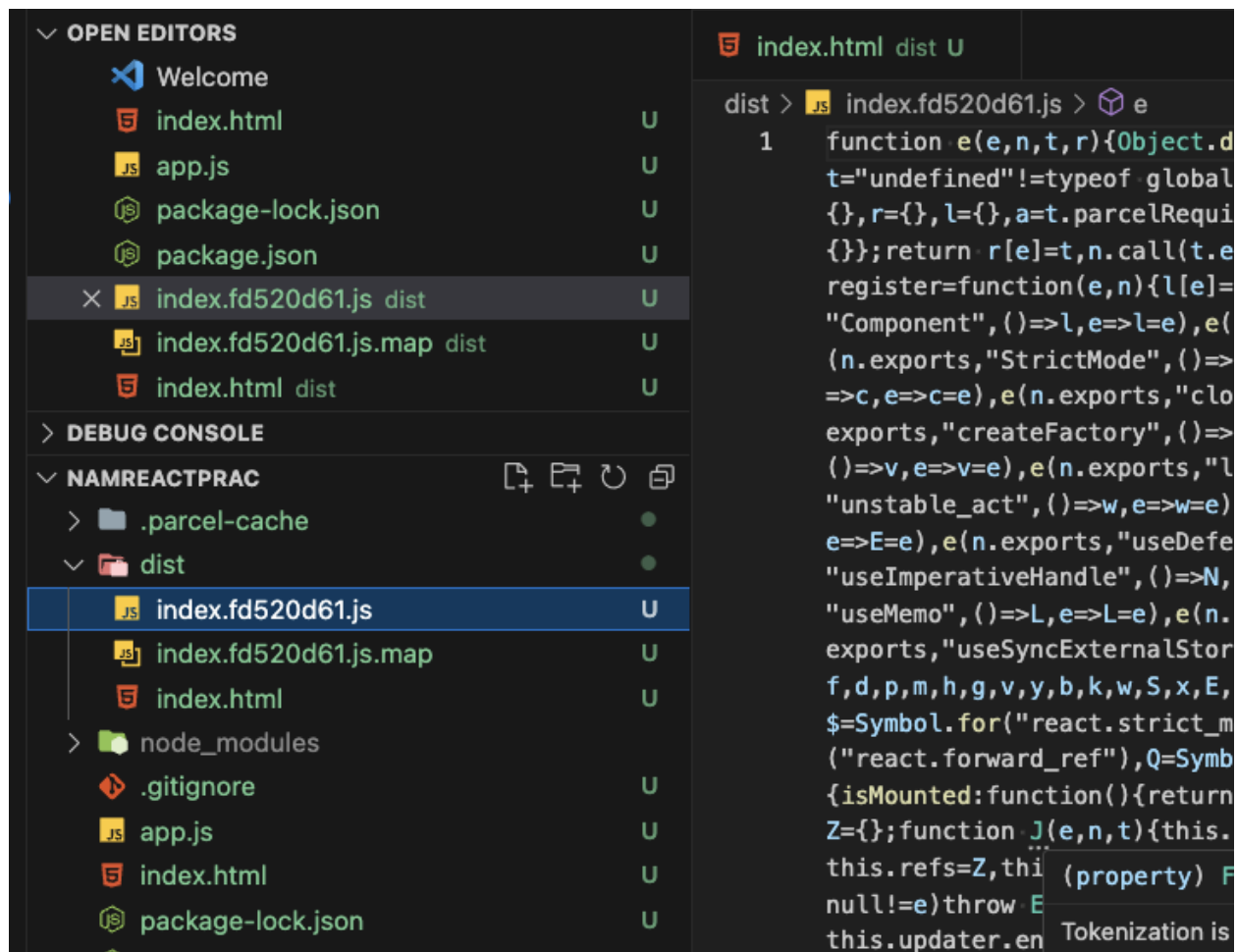
- parcel also cache the files when we run build again and again , also it save it in .parcel-cache
- image optimisation
- When we make a staging or production build it do minification of file as well
- Will do bundle the things
- It will also compress all the files, reduces space and minified it.
- It uses consistent hashing
- it will do code splitting
- It does differential bundling (ex- it gives supports to all types of browsers based on it)
- Parcel is a kind of manager for other library. It doesn't do everything by its own it is also taking help of its other sub dependency library
- It gives beautiful and to the point error message in the browser instead of just showing in console (error handling, Diagnostic)
- It gives ssl support as well (to run app in https)
- Tree shaking (for ex if we use any code or function which is not getting used then during build it will ignore and reduce it)



when we run `npx bundle index.html` then all the `app.js` and `index.html` code **local changes** went to `dist` which is local build file.

Making production build

`npx bundle build index.html`



this is create production build of code in minified version where all code will be shorten and minified. This is highly efficient build and optimised it.

since `/dist` and `.parcel-cache` can be generated again we need to add this in `gitignore`

To make browser specific support / older browser support you can add `browserlist`

```
    "react-dom": "^18.2.0"  
  },  
  "browserslist": [  
    "last 2 versions"  
  ]  
}
```

Ep -3 Layering the foundation