

1_2_3_4_introduction

11 September 2023 08:14 PM

What is Library and framework?

React is a library made by facebook, similar to jQuery

Carousel is a JS library

Framework is complete in itself, it has everything needed to create an app

A framework is a set of pre-written code that provides a structure for developing software applications. A library, on the other hand, is a collection of pre-written code that can be used to perform specific tasks

Both the framework vs library is precoded support programs to develop complex software applications. However, **libraries target a specific functionality, while a framework tries to provide everything required to develop a complete application.**

It takes minimum effort for a library to put in inside our code

What is emmet?

Emmet is a set of plug-ins for text editors that allows for high-speed coding and editing in HTML, XML, XSLT, and other structured code formats via content assist.

How to create a H1 using JS and put it inside a div with id root?

Browser has a JS engine which interprets the code written below and react accordingly

```
<title>Namaste React</title>
</head>
<body>
  <div id="root"></div>
</body>
<script>

  const heading = document.createElement("h1");

  heading.innerHTML = "Namaste Everyone from JavaScript!";

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

  root.appendChild(heading);
```

What is React CDN?

Content delivery/distribution network

A content delivery network (CDN) is a **network of interconnected servers that speeds up webpage loading for data-heavy applications.** CDN can stand for content delivery network or content distribution network.

What is cross origin in Script tag?

```

<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>

```

The crossorigin attribute **sets the mode of the request to an HTTP CORS Request**. Web pages often make requests to load resources on other servers. Here is where CORS comes in. A cross-origin request is a request for a resource (e.g. style sheets, iframes, images, fonts, or scripts) from another domain

Shortest Program of React?

The below is shortest program of react, we have just injected react CDN into our document

```

index.html X
index.html > html > script
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 </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 </html>
21

```

Now if we write React in console we get this

```

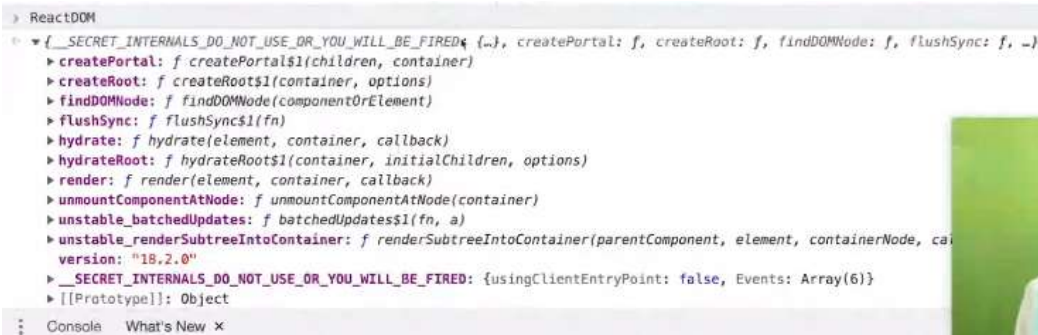
React
  {Children: {...}, Fragment: Symbol(react.fragment), Profiler: Symbol(react.profiler), Component: f, PureComponent: f, ...}

```

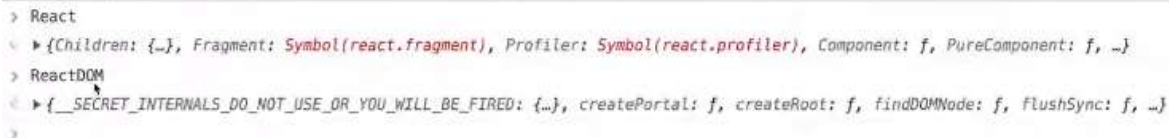
We can also use React.createContext etc etc in console now.

React is a global object and it can be used anywhere using React.something anywhere because we have added react CDN now

We also access to React DOM now



Use of those 2 CDN links were



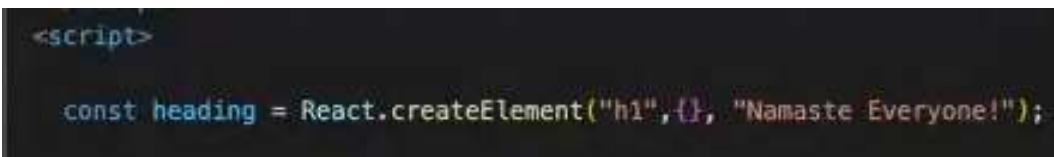
Why there are 2 files for React and ReactDOM?

React is not limited to browsers only there is React native also for mobiles

ReactDOM means web version of React which gives us access to DOM

Let us use React Now, do same H1 thing using React

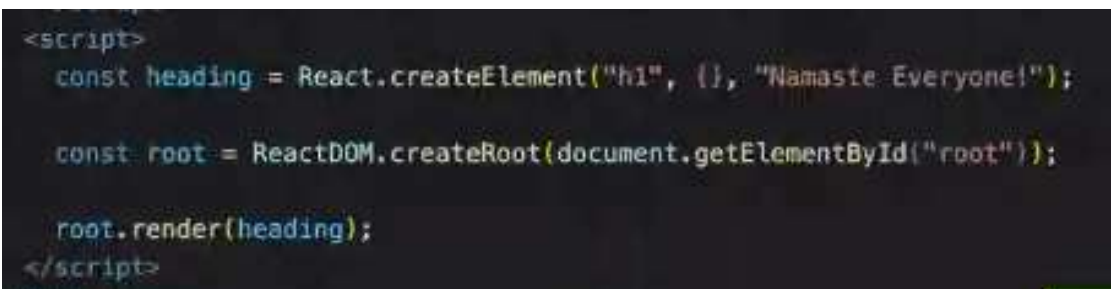
We create h1 now like this:



Now we want to render this heading inside div of id = "root"

We use const root = ReactDOM.createRoot(document.getElementById('root')) to tell react that this is my root

Now we use root.render(heading)



Output is



Namaste Everyone!



If we do console.log(heading)
We see an object of type h1
render() injects element in to DOM



Can we have multiple roots?
Generally in our react app, we have only 1 root and 1 render method

What if we have something on top of div with Id = root?
Let say we make div of id header
And div of Id = root
And div of Id = footer
What will be output?



Everything will run as it is just that React will be rendered inside root only. There can be header or footer also. So we can use react anywhere in our project like search bar, footer etc etc



Header

Namaste Everyone!

Footer

What is { } in React.createElement?

```
script>
const heading = React.createElement("h1", {}, "Namaste Everyone!");
```

Let say we have one more heading with id = title inside div with id = root

```
<body>
  <div id="root">
    <h1 id="title">Hello world</h1>
  </div>
</body>
<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>
<script>
  const heading = React.createElement("h1", {
    id: "title",
  }, "Namaste Everyone!");
```

So to make changes in id = title we can pass these parameters inside { }

```
const heading = React.createElement(
  "h1",
  {
    id: "title",
  },
  "Namaste Everyone!"
);
```

Our DOM now looks like

```
<!DOCTYPE html>
<html lang="en">
  <head>...</head>
  <body>
    <div id="root">
      <h1 id="title">Namaste Everyone!</h1>
    </div>
    <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>
  </body>
</html>
```

Now if we fill root with many h1 headings

Now if we render heading "Namaste Everyone" inside root what will happen?

React will overwrite everything and only Namaste Everyone will be there inside root

```
<body>
  <div id="root">
    <h1 id="title">Hello world</h1>
    <h1 id="title">Hello world</h1>
    <h1 id="title">Hello world</h1>
    <h1 id="title">Hello world</h1>
    <h1 id="title">Hello world</h1>
    <h1 id="title">Hello world</h1>
    <h1 id="title">Hello world</h1>
  </div>
</body>
```

Output

Namaste Everyone!

We generally write "Not Rendered Correctly" inside root div

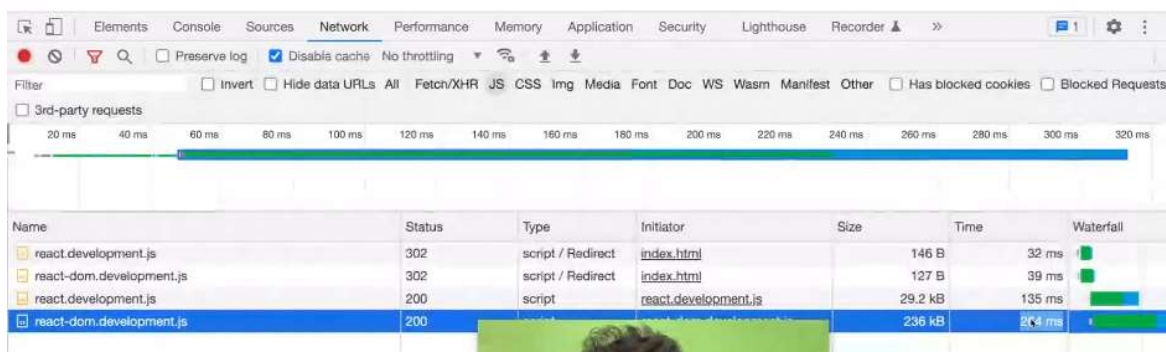
To make sure that if there is some error we see that message and can solve error that why root not rendered correctly

```
<body>
  <div id="root">Not Rendered</div>
</body>
```

It takes some time for react to render in our browser so we always see "Not Rendered Correctly" for sometime on doing refresh.

It takes sometime for scripts to load

Namaste Everyone!



The screenshot shows the Chrome DevTools Network tab with a list of resources. The resource 'react-dom.development.js' is highlighted, showing its status as 200, type as script, and size as 236 kB. The waterfall chart below the table shows the loading timeline of various resources, including 'react.development.js' and 'react-dom.development.js'.

Name	Status	Type	Initiator	Size	Time	Waterfall
react.development.js	302	script / Redirect	index.html	146 B	32 ms	
react-dom.development.js	302	script / Redirect	index.html	127 B	39 ms	
react.development.js	200	script	react.development.js	29.2 kB	135 ms	
react-dom.development.js	200	script	react-dom.development.js	236 kB	234 ms	

What if we put script tag inside body below div id = root?

It will not work.


```

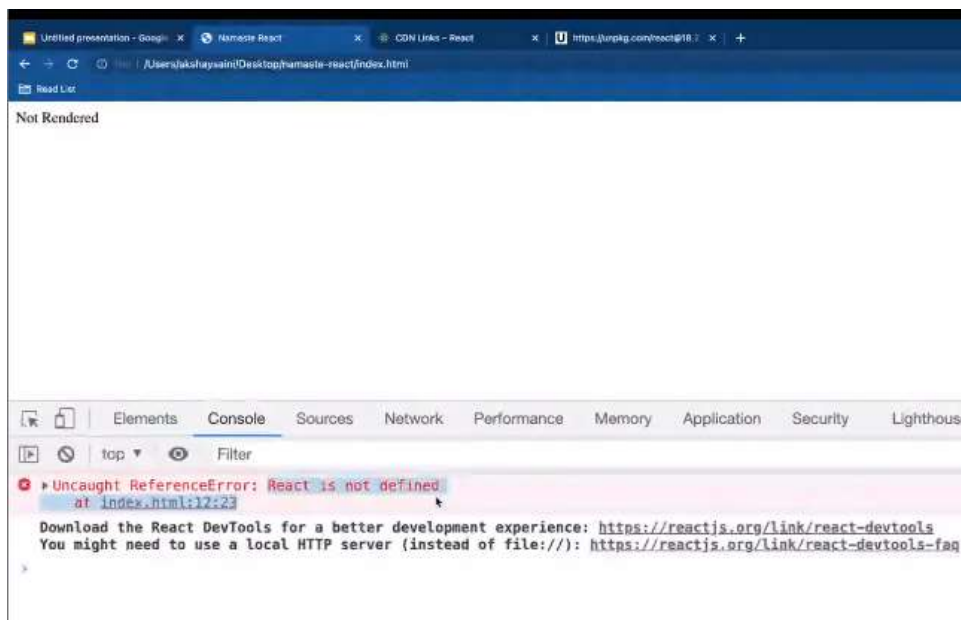
<body>
  <div id="root">Not Rendered</div>
  <script>
    const heading = React.createElement(
      "h1",
      {
        id: "title",
      },
      "Namaste Everyone!"
    );

    console.log(heading);

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

    //passing a react element inside the root
    root.render(heading);
  </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>
</body>

```



What is difference between async/defer?

Async allows your script to run as soon as it's loaded, without blocking other elements on the page. **Defer** means your script will only execute after the page has finished loading. In most cases, async is the better option — but there are exceptions

Async in script tag is a way to load scripts asynchronously. That means, if a script is async, it will be loaded independently of other scripts on the page, and will not block the page from loading.

If you have a page with several external scripts, loading them all asynchronously can speed up the page load time, because the browser can download and execute them in parallel.

To use async, simply add the async attribute to your script tag:

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

By using the defer attribute in HTML, the browser will load the script only after parsing (loading) the page. This can be helpful if you have a script that is dependent on other scripts, or if you want to improve the loading time of your page by loading scripts after the initial page load.

To use defer, simply add the defer attribute to your script tag:

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

If we want to build the below structure in our HTML using React. How to do it?

```
<body>
  <div id="root">Not Rendered</div>

  <div id="container">
    <h1>Heading 1</h1>
    <h2>Heading 2</h2>
  </div>

  <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>
</body>
```

We do it like:

We will pass heading 1 and heading 2 as an Array

We put heading 1 and heading 2 inside container

Now we render the container inside root


```

<script>
  const heading = React.createElement(
    "h1",
    {
      id: "title",
    },
    "Heading 1"
  );

  const heading2 = React.createElement(
    "h2",
    {
      id: "title",
    },
    "Heading 2"
  );

  const container = React.createElement(
    "div",
    {
      id: "container",
    },
    [heading, heading2]
  );

  console.log(heading);

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

  //passing a react element inside the root

  //async defer
  root.render(container);

```

Our DOM looks like:



If we want to build a big index.html

The above method is not development-friendly

React came with a idea to build HTML using JS

We need not to go to HTML anymore, we can do anything from React using APIs like createElement() etc etc.

To make a complex file, its better to split our components and put all JS in app.js

```

<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>
<script src="App.js"></script>
</body>
</html>

```



Why do we import CSS inside head in HTML?

seeks to reduce the number of times the browser must re-flow the document by ensuring that the CSS styles are all parsed in the head, before any body elements are introduced.

This is based on the best practice for optimizing browser rendering.

What is rel = stylesheet in link tag while linking CSS in HTML?

The REL attribute is used **to define the relationship between the linked file and the HTML document**. REL=StyleSheet specifies a persistent or preferred style while REL="Alternate StyleSheet" defines an alternate style. A persistent style is one that is always applied when style sheets are enabled.

CDN of react.development.js is for development

CDN of react.production.js has same code but it is much more optimised and for production

Session 2

Revising JS

What is function keyword in JS?

It is present inside JS,

Data Structure used for memory in JS is Heap

Function without name is anonymous function which can be assigned to a variable as well

```

function x() {
  const a = 10;
}
var xyz = 30;

x(); //functional execution context is created

var x = function () {
  console.log("I'm an anonymous function");
  I

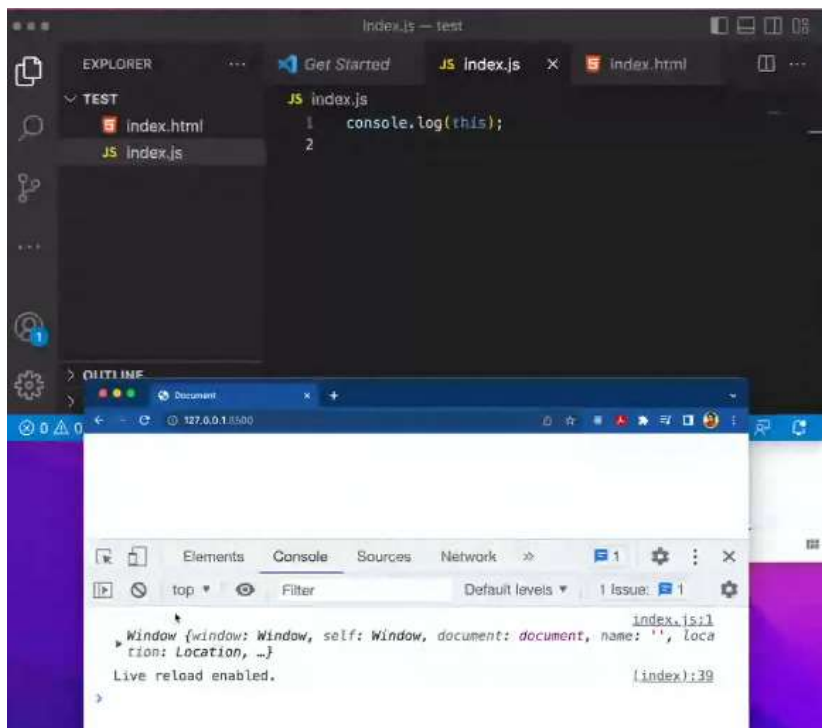
```

Expression is something that executes. Like `console.log` is a expression but `a = 30` is not a expression.

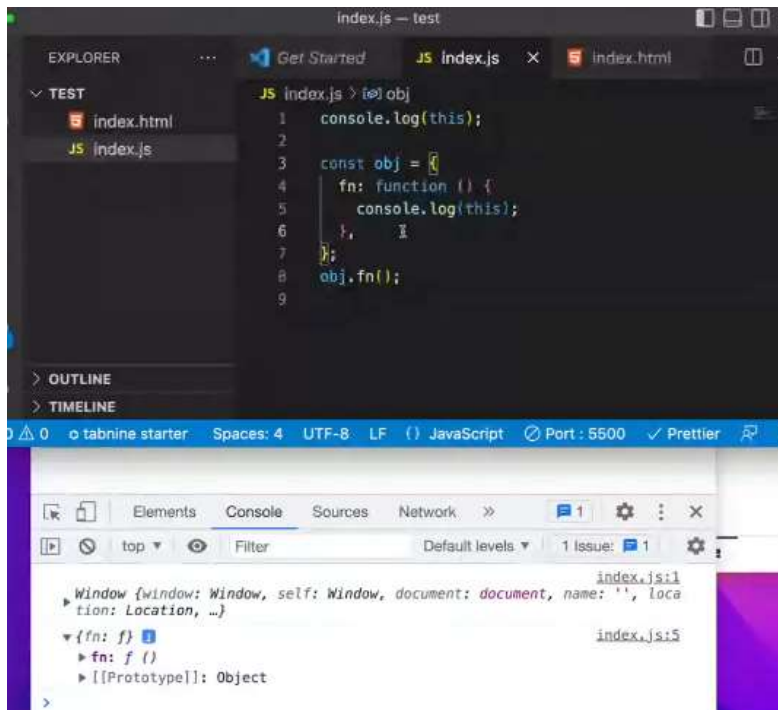
Arrow function was introduced in ES6 with `let`, `const`, promises, spread operator etc etc
`=>` is known as fat-arrow

Only difference between normal function and arrow function is 'this' keyword
 Arrow function, 'this' refers to window

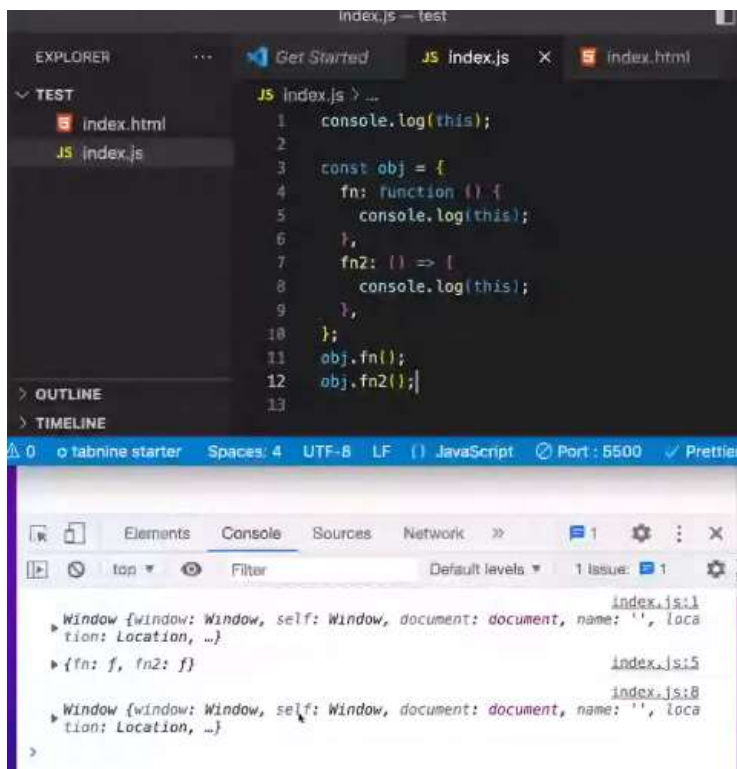
Output of `console.log(this)` is window object



In normal function 'this' refer to parent object

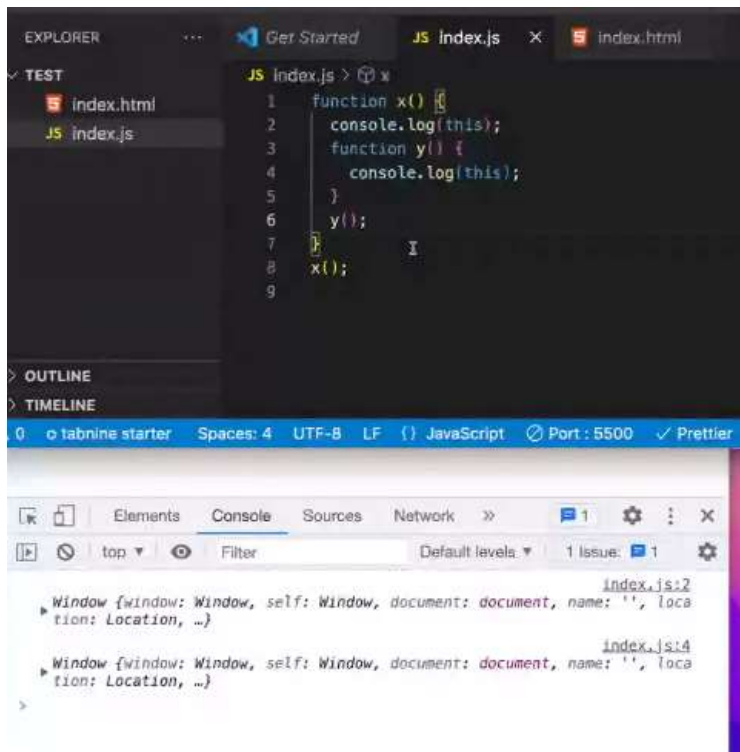


But for arrow function it refers to window object

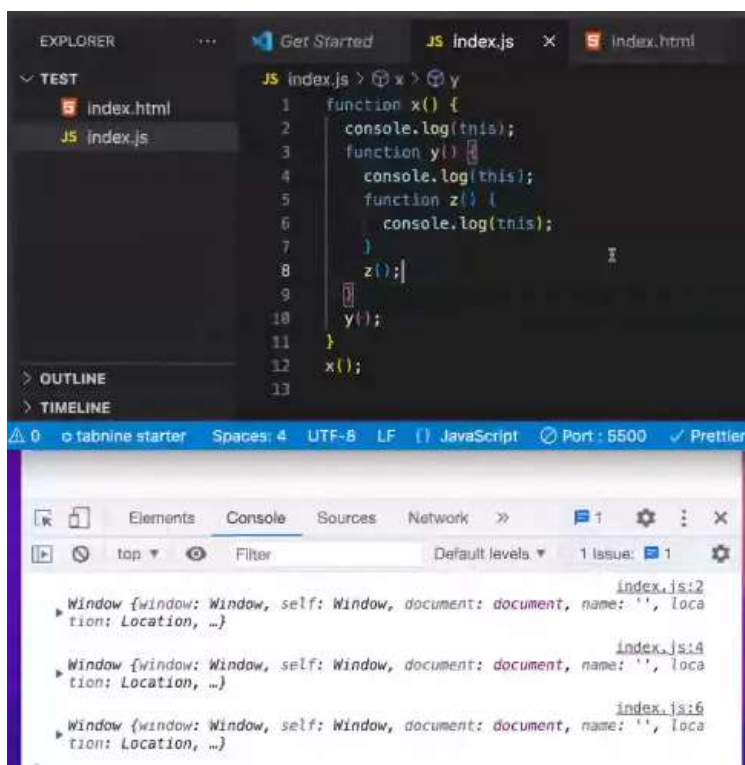


Window is the global object given to us by the browser

What if we have normal function and another normal function inside it.



What if we have one more function inside it Z ()



If we make a function and an object.
We call that object using the function. Let's see what happens

```
1  const person = {
2    name: "Akshay",
3  };
4  const person2 = {
5    name: "Simran",
6  };
7
8  function x() {
9    console.log(this);
10 }
11 x.call(person);
12
```

Console output: {name: 'Akshay'}

```
1  const person = {
2    name: "Akshay",
3  };
4  const person2 = {
5    name: "Simran",
6  };
7
8  function x() {
9    console.log(this);
10 }
11 x.call(this);
12 x.call(person);
13 x.call(person2);
14
```

Console output: Window {window: Window, self: Window, document: document, name: '', location: Location, ...}, {name: 'Akshay'}, {name: 'Simran'}

What if we put that function inside object


```
1 const person = {
2   name: "Akshay",
3   print: function () {
4     console.log(this);
5   },
6 };
7 const person2 = {
8   name: "Simran",
9 };
10
11 person.print();
12
```

Ln 12, Col 1 Spaces: 4 UTF-8 LF JavaScript

▶ {name: 'Akshay', print: f}

What if we use .call()
call takes window object and then this refers to window object

```
1 const person = {
2   name: "Akshay",
3   print: function () {
4     console.log(this);
5   },
6 };
7 const person2 = {
8   name: "Simran",
9 };
10
11 person.print();
12 person.print.call();
13 person.print.call(this);
14
```

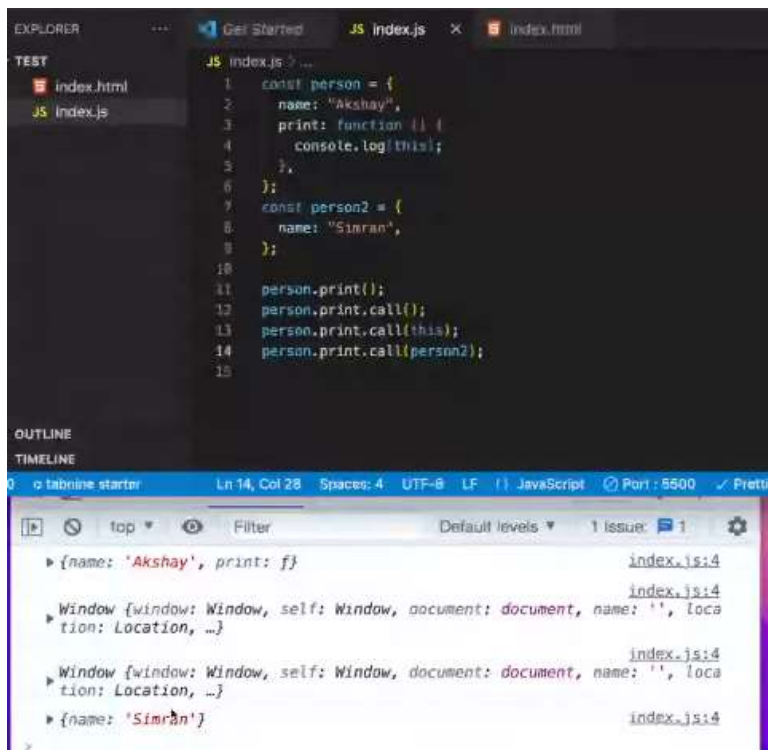
Ln 13, Col 26 Spaces: 4 UTF-8 LF JavaScript Port: 5500

▶ {name: 'Akshay', print: f} index.js:4

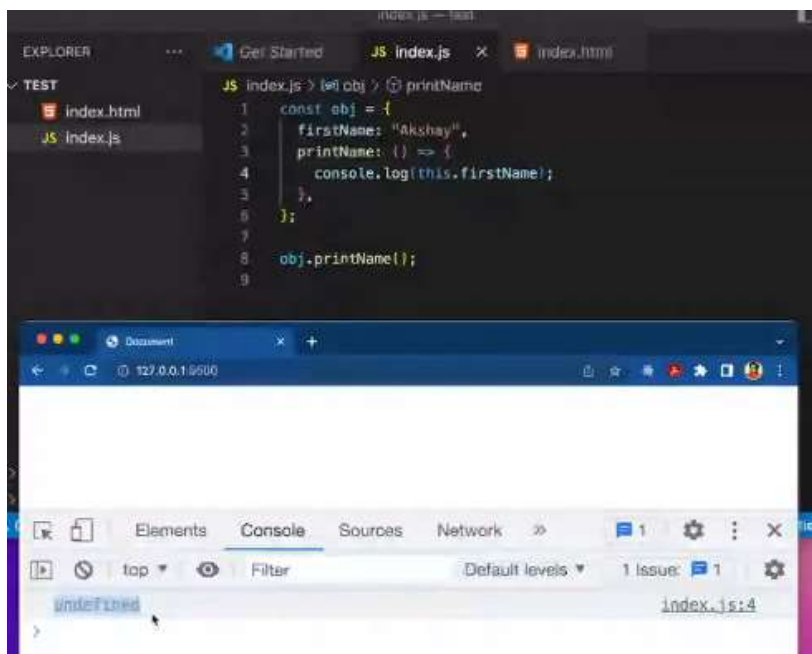
▶ Window {window: Window, self: Window, document: document, name: '', location: Location, ...} index.js:4

▶ Window {window: Window, self: Window, document: document, name: '', location: Location, ...} index.js:4

What if we use .call(object)



This refers to window in arrow function that is why we are getting undefined



Interview Tips

Luck is very important and we cannot control it.
 Many Companies don't train their interviewers well
 A person can be a good software engineer but bad interviewer

What we can control is:

1. Our preparation

Technical preparation

Communication skills: Lot of people fail due to communication skills ,We spend so much time for tech skills but comm skills are equally important.

Learn to speak while you write, speak your thoughts.

Practice to speak even when you are coding alone.

Mock Interviews

2. Talk while you are coding so that interviewer can also see what you are doing.

If you cannot explain, interviewer thinks you also don't know or you have crammed things.

3. In a company you don't work alone so comm skills should be good. Does not matter how good Software engineer you are, you should be able to communicate your thoughts and ideas to others.

4. Spoken english is very important.

5. Preparation on the interview day (not technical preparation)

You should not be in panic state before/during interview

Keep your pen and paper handy

Keep you water, next to you.

Keep your laptop charged, keep your charger handy.

Keep power point near to you.

These small things mess your interview.

Keep your camera always open during interview.

Keep your phone on silent.

Have a power backup, mini UPS (it does not cost much).

6. Confidence comes from preparation (not just your technical prep) (it includes non tech prep also which are mentioned above).

Session 3

How to make our app deployable?

How to launch our app

How to build our own create-react-app

Let say with ID we also pass something which is not an attribute, will it also get passed in container?

```
const heading2 = React.createElement(
  "h2",
  {
    id: "title",
  },
  "Heading 2"
);

const container = React.createElement(
  "div",
  {
    id: "container",
    hellow: "world",
  },
  [heading, heading2]
);
```

Yes, we can put in anything over { } while using createElement



They are not called as attributes

They are called as props just like properties or attributes

Is this a production ready app?

What it needs to be production ready?

Bundle Everything, remove console, server, optimisation, Caching, minify many things

We need to use something known as Bundlers

Same as web-pack, it is a type of bundler

Vite is also a bundler, parcel is also a bundler

We will be using Parcel, in original create-react-app, web-pack is used as Bundler and it also uses Babel.

Module Bundling, on a high level, is a process of integrating together a group of modules in a single file so that multiple modules can be sent to the browser in a single bundle.

When we write our code in a modular pattern:

- We keep our javascript in separate files and folders based on functionality.
- Add the script tag for each file that we are using in correct order of dependency.

For each script tag, browser will send the request to the server which will have bad effect on the performance of our application. In order to overcome this we usually create a single bundled file which will integrate all the other files and that bundled file is sent to browser.

Module bundling can also include a minification step i.e. all the unnecessary characters like space, comma, comments etc. are removed from the file and its minified version is created

and whenever a request comes from the browser that minified version is sent back. Less data means less browser processing time.

Webpack vs Rollup vs Parcel :

All these differ in following things, although there is very little difference.

They almost do the same job

Configuration, Entry points, Transformations, Tree Shaking, Dev Server , Hot Module Replacement, Code Splitting

All above optimisations are done by bundlers only.

What is Parcel?

Parcel is a package, means a module of JS files

Some piece of code so we need a package manager also

That is why we do "npm init"

What can we do other than "npm"?

We can also use "yarn"

What is npm?

It does not stand for node package manager

If we go to official doc of npm we see

♥ Nerdiest Precious Modules

♥ New Priority Mail

♥ Nanometers Per Millisecond

♥ Nitrogen Poisonous Monoxide

But now-where is written node package manager.

There is a repo of npm where they take any name for npm.

So there is no official name for npm

npm is the world's largest **Software Registry**.

The registry contains over 800,000 **code packages**.

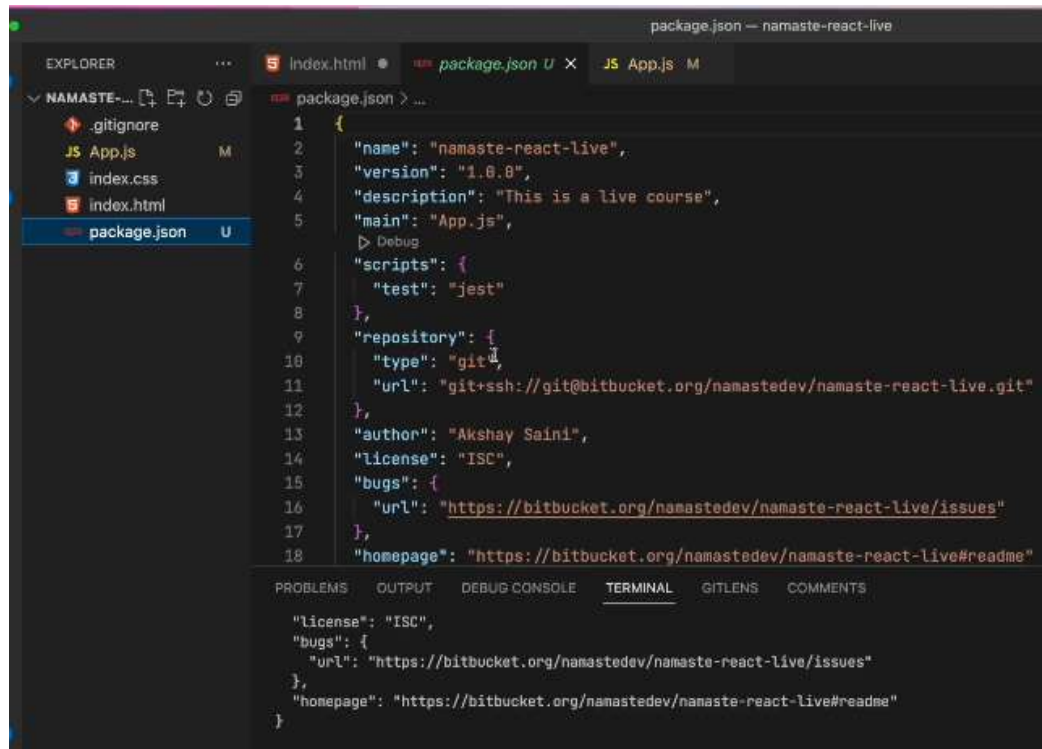
How to install npm?

We write "npm init"

```
{
"name" : "foo",
"version" : "1.2.3",
"description" : "A package for fooing things",
"main" : "foo.js",
"keywords" : ["foo", "fool", "foolish"],
"test command": jest,
"author" : "John Doe",
```

```
"licence": "ISC"
}
```

Now we get package.json in our file and folder structure



The screenshot shows a code editor with the following content:

```
package.json — namaste-react-live
EXPLORER
  NAMASTE-...
    .gitignore
    JS App.js
    index.css
    index.html
    package.json
  package.json
1 {
2   "name": "namaste-react-live",
3   "version": "1.0.0",
4   "description": "This is a live course",
5   "main": "App.js",
6   "scripts": {
7     "test": "jest"
8   },
9   "repository": {
10    "type": "git",
11    "url": "git+ssh://git@bitbucket.org/namastedev/namaste-react-live.git"
12  },
13  "author": "Akshay Saini",
14  "license": "ISC",
15  "bugs": {
16    "url": "https://bitbucket.org/namastedev/namaste-react-live/issues"
17  },
18  "homepage": "https://bitbucket.org/namastedev/namaste-react-live#readme"
}
```

Below the code editor, the TERMINAL tab is active, showing the following output:

```
"license": "ISC",
"bugs": {
  "url": "https://bitbucket.org/namastedev/namaste-react-live/issues"
},
"homepage": "https://bitbucket.org/namastedev/namaste-react-live#readme"
}
```

Why do we use npm?

Because we use so many packages so we need someone to manage these many packages. Our app does not run only on react, we need certain packages to build it. These packages come inside npm.

What is inside package.json?

All information we filled.

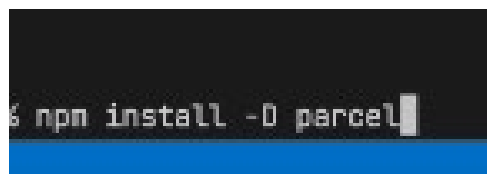
How to ignite our app?

We use parcel (Zero configuration build tool for everything)

npm install to install a package + package name

We do not want parcel in production we want it for development so we use -D

npm install -D parcel where -D means devDependency



```
% npm install -D parcel
```

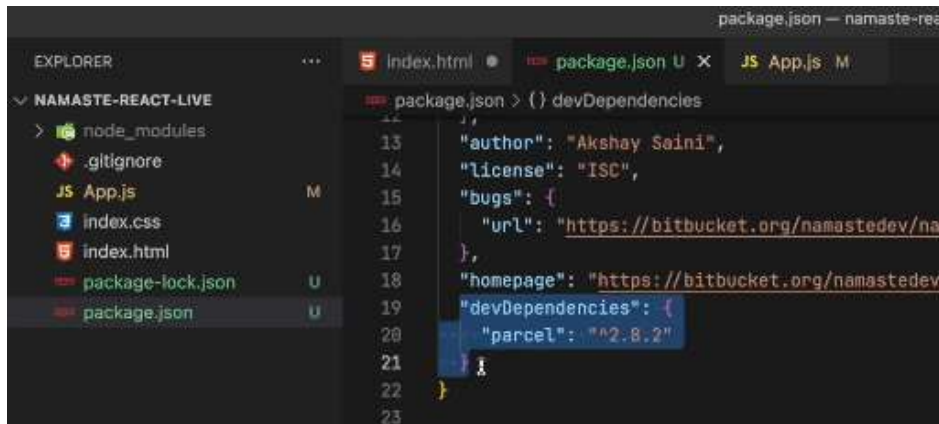
Some people also do --save-dev also instead of -D, it is same

Dependency means all packages that my project needs, my project is dependent on it.

Parcel is one of the dependency which our react project need as bundler.

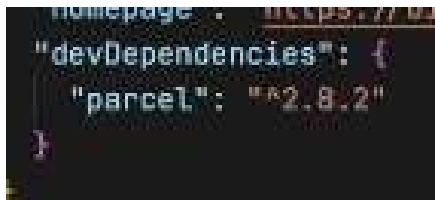
Now we get package.lock.json and in package.json we have devDependencies {

Parcel : "version" }



Package-lock.json is an important file.

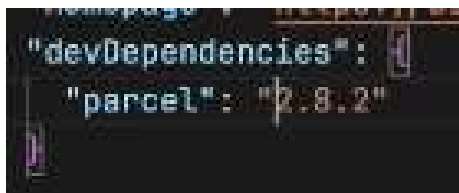
^ is known as caret



What's the 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. Our package will automatically update to new version for major changes.
- ^version **“Compatible with version”**, will update you to all future minor/patch versions, without incrementing the major version. ^1.2.3 will use releases from 1.2.3 to <2.0.0. Our package will automatically update to new version for minor changes.

If we do not use anything means I just want this version only, do not update



What is package-lock.json?

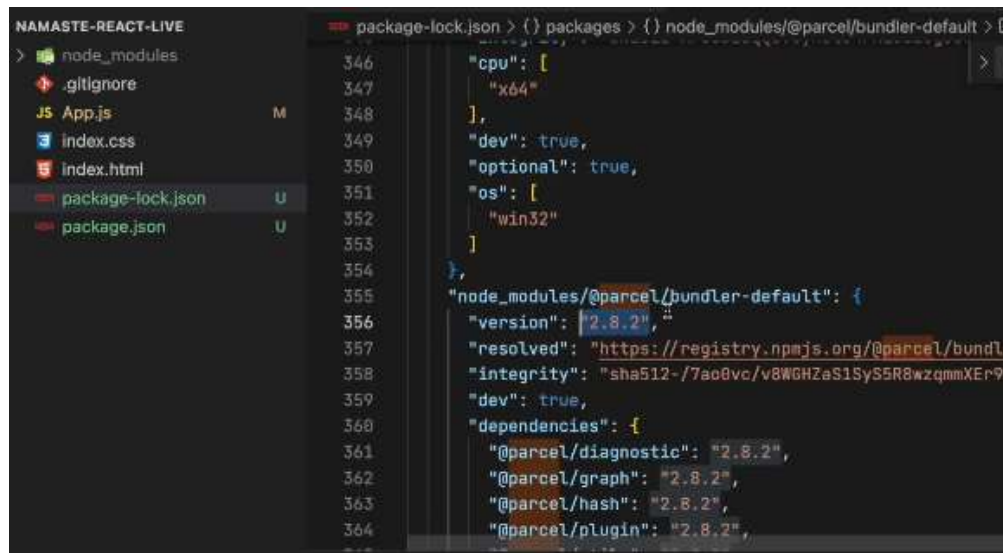
package.json

It records important metadata about the project.

package.lock.json

It allows future devs to install the same dependencies in the project.
It records the exact version of every installed dependency, including its sub-dependencies and their versions.

We see our package-lock.json and we get exact version of our Parcel.



```
346     "cpu": [  
347       "x64"  
348     ],  
349     "dev": true,  
350     "optional": true,  
351     "os": [  
352       "win32"  
353     ]  
354   },  
355   "node_modules/@parcel/bundler-default": {  
356     "version": "2.8.2",  
357     "resolved": "https://registry.npmjs.org/@parcel/bundl  
358     "integrity": "sha512-7ao8vc/v8WGHZaS1SyS5R8wzqmmXEr9  
359     "dev": true,  
360     "dependencies": {  
361       "@parcel/diagnostic": "2.8.2",  
362       "@parcel/graph": "2.8.2",  
363       "@parcel/hash": "2.8.2",  
364       "@parcel/plugin": "2.8.2",
```

It is working on my local but breaking in production?

Let say we use ^ in package.json

So package updates itself automatically and it breaks in production so package-lock.json has exact version, it locks the version.

It takes snapshot of exact version we have in our project.

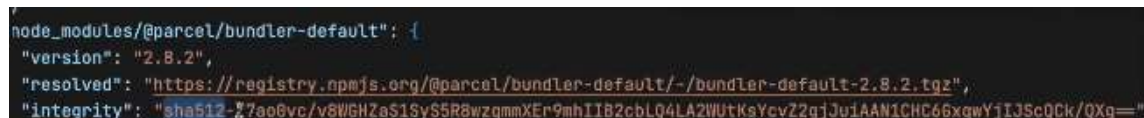
Important things about package-lock

Never put package-lock.json in git ignore

Always put it in your git with your project.

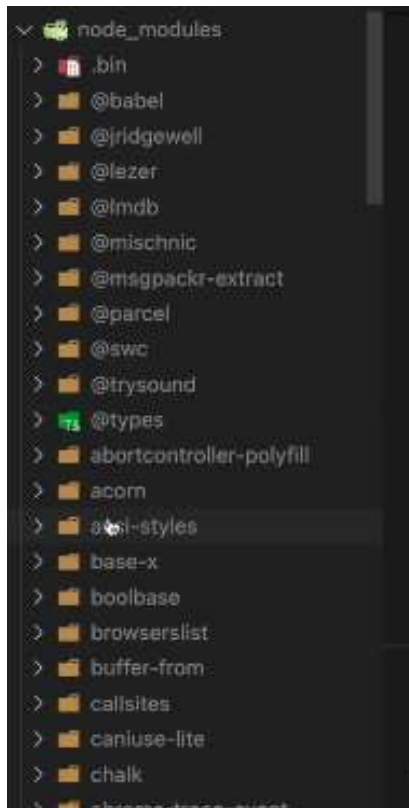
We need to push our package-lock to server and we cannot do it directly so we push it in git and server fetch it from git so we need to push package-lock in git.

It maintains a hash of version of package also, keeps track that hash is same in local and production. It maintains the integrity.



```
node_modules/@parcel/bundler-default": {  
  "version": "2.8.2",  
  "resolved": "https://registry.npmjs.org/@parcel/bundler-default/-/bundler-default-2.8.2.tgz",  
  "integrity": "sha512-7ao8vc/v8WGHZaS1SyS5R8wzqmmXEr9mhIIB2cbLQ4LA2WUtKsYcvZ2gJJu1AAN1CHC66xqwYjI3ScQCk/QXg="
```

When we installed parcel, node modules also got created.



Whatever we install, it gets installed in node modules.

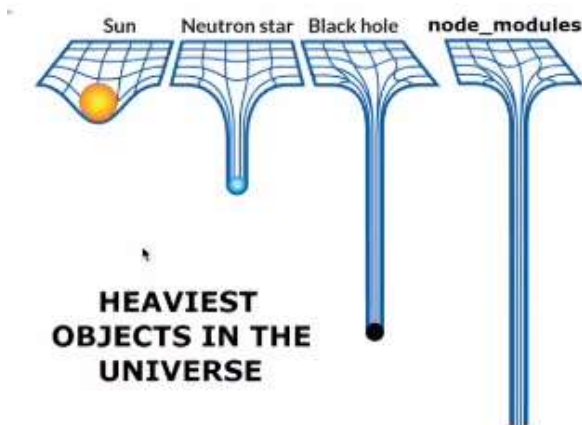
So node modules is like database of npm.

Parcel is there in node modules.

Parcel uses lot of things to optimise our app.

Parcel also has many dependencies to do this optimisation, they are also inside node modules so our node modules becomes huge.

We have something like browserslist in node modules which help our app work nicely in older version of browsers, sameway there are many packages to optimise our app.



Should we add our node_modules to our git??

It is foolish to put node modules in git repo, it is the heaviest thing in your project. It 1GB large.

Our package-lock.json has sufficient material to make another node modules

We can generate another node modules using package-lock.json so no need to push node_modules.

Just push package-lock.json

We will generate our node modules in server using package-lock.json and it will make sure our app does not break due to node modules not being pushed inside git.

We are using CDN to get react in our project

This is not the good way.

Currently we are using react 18, what if it gets updated to react 19?

CDN is on different server

What if we create our own server and fetch react from it?

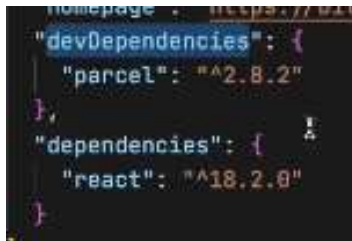
That is why we do not use CDN and create our own react app

How to install react in our project?



```
% npm install react
```

Now we do not use -D as we need react globally and we get react in our dependencies in package.json

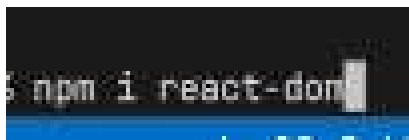


```
homepage: https://...  
"devDependencies": {  
  "parcel": "^2.8.2"  
},  
"dependencies": {  
  "react": "^18.2.0"  
}
```

This is the right way to install react in our project.

Now we install ReactDOM

"npm i" is same as "npm install"



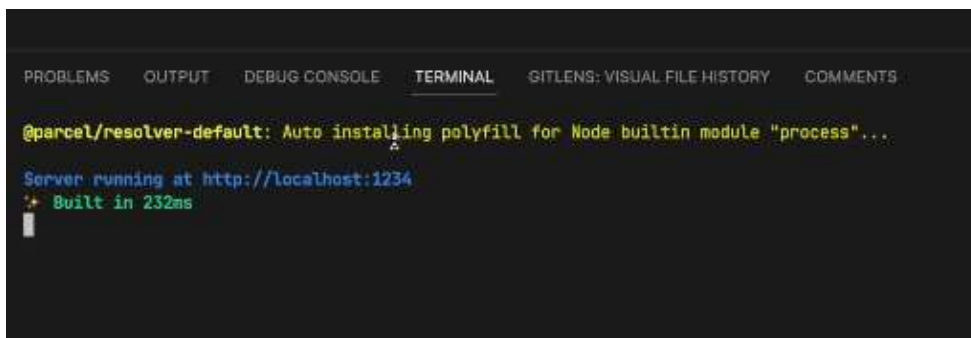
```
$ npm i react-dom
```

npx means execute using npm

Now we use "npx parcel index.html"

Where index.html is our entry-point

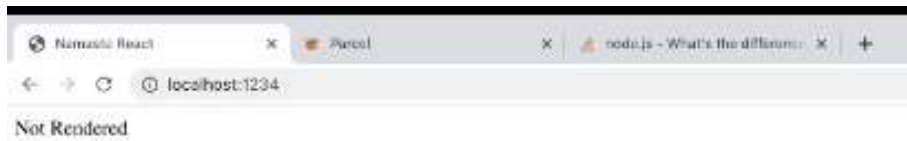
It starts a mini-server for us.



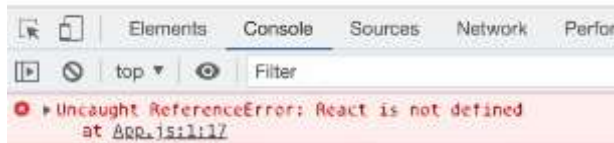
```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL GITLENS: VISUAL FILE HISTORY COMMENTS  
@parcel/resolver-default: Auto installing polyfill for Node builtin module "process"...  
Server running at http://localhost:1234  
+ Built in 232ms
```

Parcel gave a mini-server to us.

Now our app runs on localhost:1234



Now our console gives this error



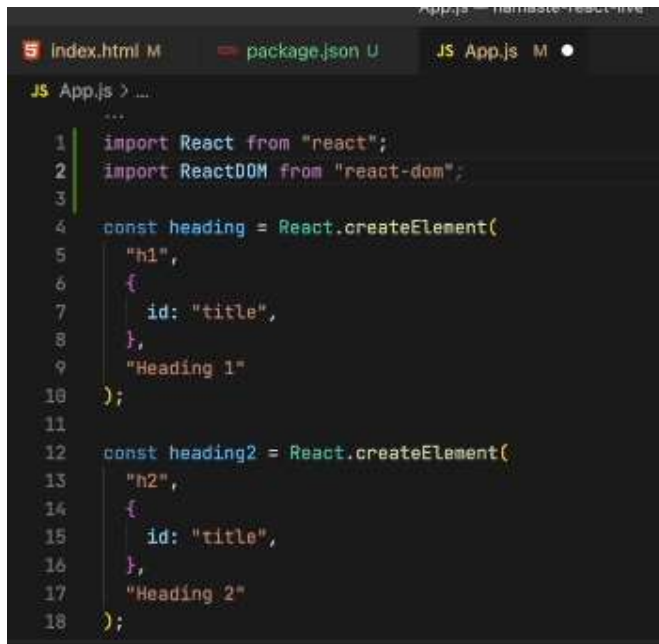
Because we have use `React.createElement()` etc in our `app.js` but we have not imported React in our js file

How to import React??

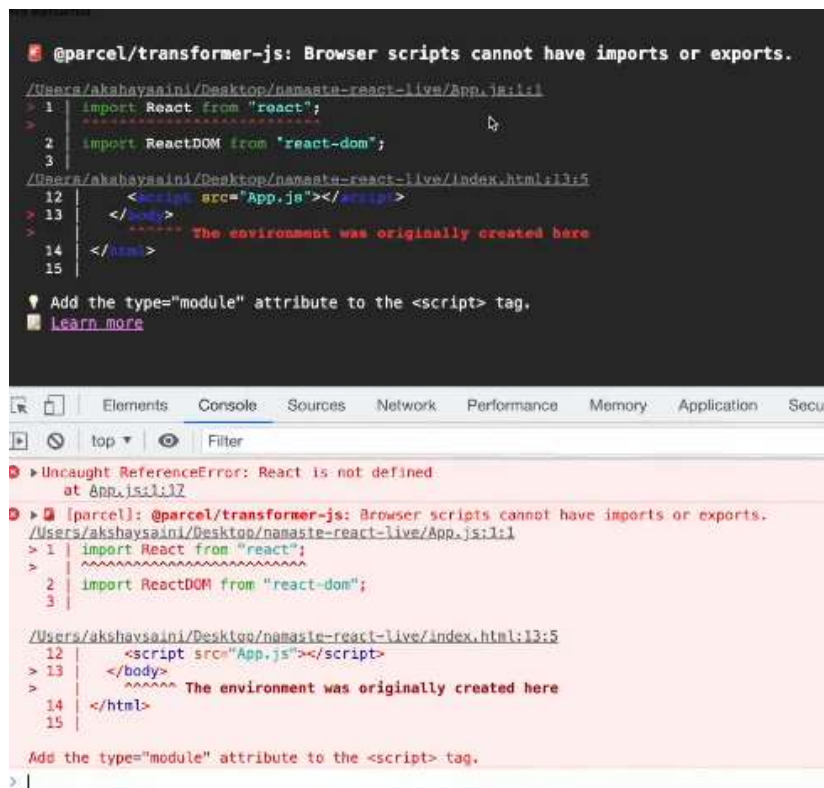
We use a keyword "import"

We did not get this error before because we were using a CDN earlier.

We want to use react and reactDOM from our node modules now so we use "import"



Now our code gives below error



Never touch your node modules

Never update your package.json or node modules.

The error is coming because we have use `<script src= "app.js">` in index.html

And we use import in app.js

Browser does not understands import

So we need to tell browser that this is not a normal JS file, it's a module so what we do is Specify it's a type = "module"



Earlier it was like

import ReactDOM from "react-dom" but now they have updated it to



There is something known as **Hot Module Replacement (HMR)**

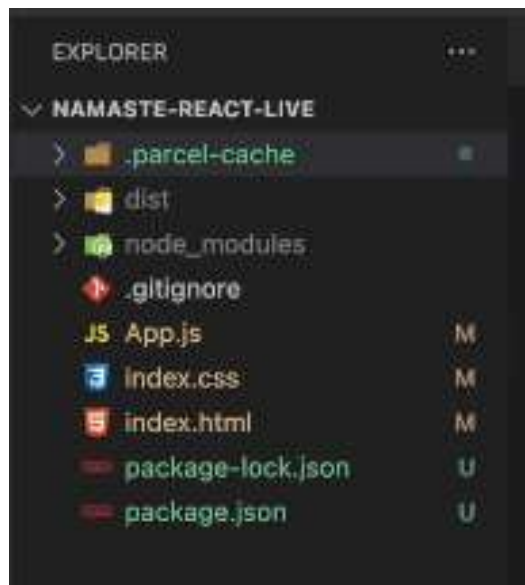
Means keep on updating the page on saving any changes

Parcel does this HMR, and saves all the changes in our HTML,CSS or JS.

How does Parcel does HMR?

There is something known as **file watcher algorithm** and parcel uses this algo which is written in **C++ internally**.

What is parcel_cache and dist in file and folder structure?



Parcel need some space to do this HMR and other things

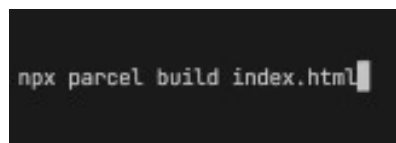
So it creates parcel_cache and it contains all files which does all optimisation things for us in parcel

dist folder keeps the file minified for us

npx parcel index.html creates a development build and host it on our server.

How to tell parcel to make a production build?

We use "build" command and it minifies all our files, get it ready for production and push them inside dist folder.



We use "main: App.js" in our package.json which tells entry point of our app as App.js
Which is not needed if we are using parcel so we delete this command

Parcel minified everything for us.
It bundled everything for us.
Removed all console logs and cleaned the code

What takes lot of time to load in browser?

No, HTML

No, CSS

No, JS

node modules is in server so it is not the answer.

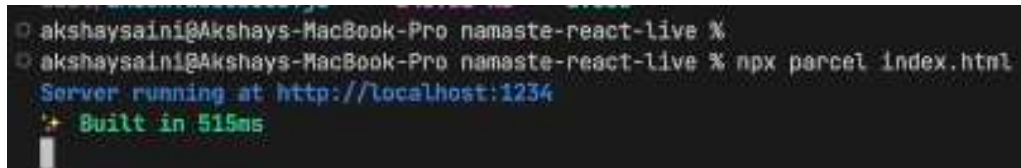
Images, media takes most time to load in browser

Parcel manages dev and production build and is superfast

Parcel does image optimisation also, it minifies the images also if they are in our project.

Parcel also does **Caching while development for us**, What is caching?

We see our build took 515ms initially for dev build.

A terminal window on a dark background. The prompt is 'akshaysaini@Akshays-MacBook-Pro: ~/namaste-react-live'. The command 'npx parcel index.html' has been executed. The output shows 'Server running at http://localhost:1234' and 'Built in 515ms' with a green star icon.

Now if we refresh our page we see time is 5ms, 4ms, 9ms

The time is reducing means parcel is using something known as caching.

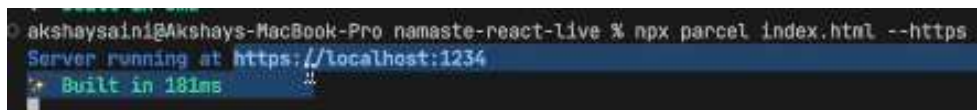
Parcel also compresses our files, renames our variables. This process also called compression.

Parcel checks your project's compatibility with older version of browsers.

Sometimes we need our build to test in https also. Parcel gives us that functionality to enable https in our localserver

A terminal window showing the command 'npx parcel index.html --https' being entered at the prompt.

It took lot of build time this time as we are doing this change for first time, if we do it again It takes lesser time due to caching

A terminal window showing the output of 'npx parcel index.html --https'. It displays 'Server running at https://localhost:1234' and 'Built in 181ms' with a green star icon.

If we run 2 project at a time, Port number of server will also change automatically, it is also managed by Parcel.

Should we push parcel-cache in git?

We should not push our parcel-cache in git. And we should put it in our gitignore



Because anything which can be auto-generated on server will be put inside git-ignore. We do not push it to git.

So we do not push parcel-cache and dist in git as we can generate them later

Parcel also uses something known as consistent hashing algorithms to cache things up and bundling up.

Parcel is a zero-config bundler.

If React is Narendra Modi then Parcel is Amit shah



If BJP wins, its not only because of Namo and amit shah

There are some other ministers which are more or less important.

React (modi) need Parcel (amit shah) and parcel need many dependencies (other ministers) .

What are benefits of using Parcel?

```

*
* HMR - Hot Module Reloading
* File Watcher algorithm - C++
* BUNDLING
* MINIFY
* Cleaning our Code
* Dev and Production Build
* Super Fast build algorithm
* Image Optimization
* Caching while development
* Compression
* Compatible with older version of browser
* HTTPS on dev
* port Number
* Consistent Hashing Algorithm
* Zero Config
*
*

```

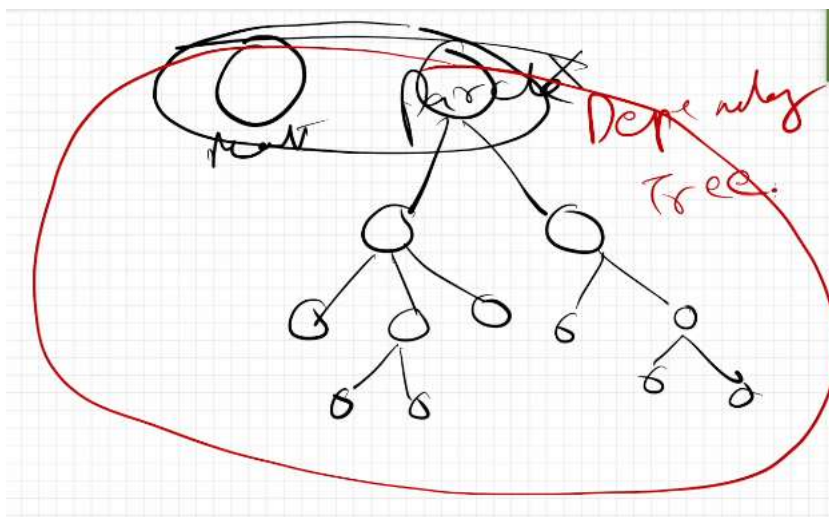
Parcel uses many dependencies in itself and all dependencies can have there dependencies so this is infinite loop and this process is called **Transitive Dependencies** (dependencies ki dependencies)

```

//
"node_modules/@parcel/bundler-default": {
  "version": "2.8.2",
  "resolved": "https://registry.npmjs.org/@parcel/bundler-
  "integrity": "sha512-/7ao8vc/v8W6HZaS1Sy5SR8wzqmmXEr9n
  "dev": true,
  "dependencies": {
    "@parcel/diagnostic": "2.8.2",
    "@parcel/graph": "2.8.2",
    "@parcel/hash": "2.8.2",
    "@parcel/plugin": "2.8.2",
    "@parcel/utis": "2.8.2",
    "nullthrows": "^1.1.1"
  },
  "engines": {
    "node": "≥ 12.0.0",
    "parcel": "^2.8.2"
  },
  "funding": {
    "type": "opencollective",
    "url": "https://opencollective.com/parcel"
  }
}

```

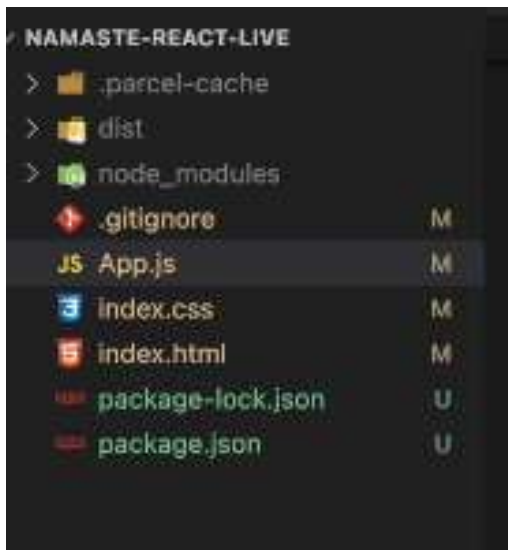
This structure of Transitive Dependencies is called **Dependency Tree**



Read the docs (Be Curious)



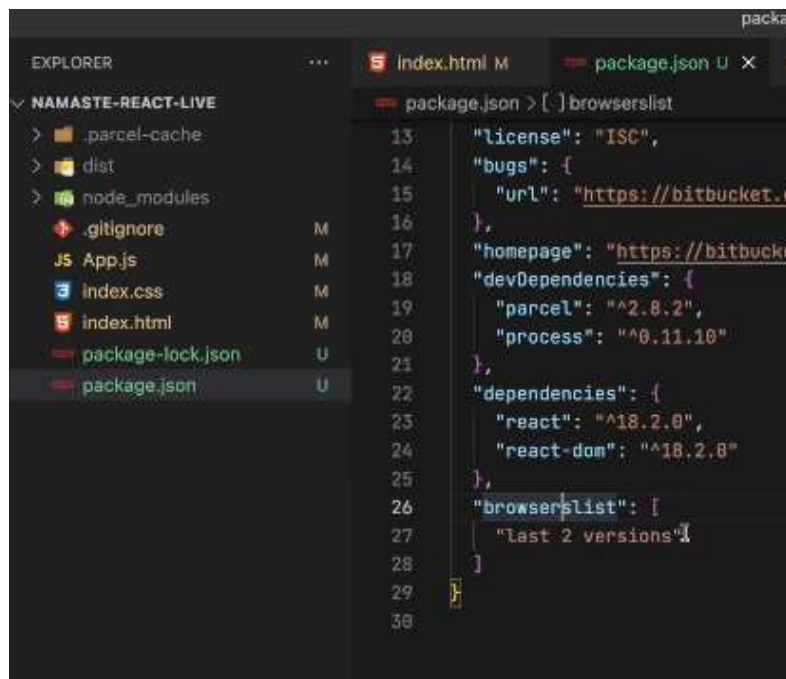
This is our create react app without using create-react-app



How to make your app, compatible with older version of browsers?
We use **browserList** which is already used by parcel.

```
"browserslist": [  
  "defaults and supports es6-module",  
  "maintained node versions"  
]
```

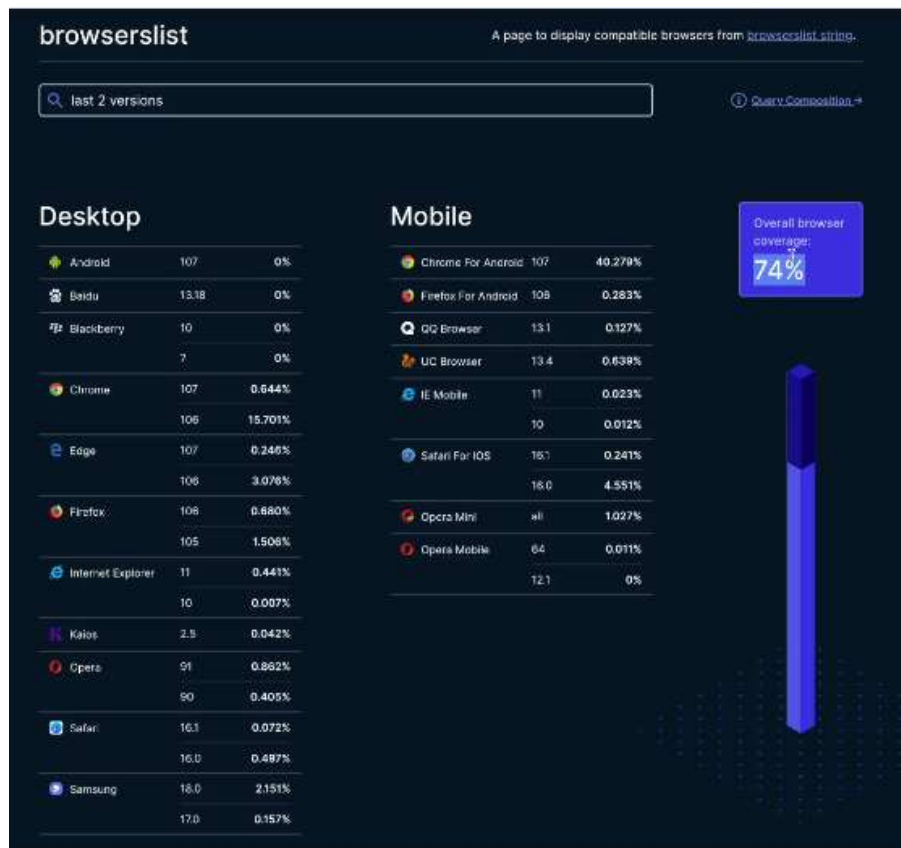
We put this code in our package.json
browserList is feeded with a Array which has some configurations.
If we write "last 2 versions" our app will run in last 2 versions of all browsers available



If we want support for only last 2 chrome versions, we write



We can use browserList.dev website to see



Full List

You can specify the browser and Node.js versions by queries (case insensitive):

- **defaults**: Browserslist's default browsers (`> 0.5%`, `last 2 versions`, `Firefox ESR`, `not dead`).
- **By usage statistics**:
 - `> 5%`: browsers versions selected by global usage statistics. `>=`, `<` and `<=` work too.
 - `> 5% in US`: uses USA usage statistics. It accepts [two-letter country code](#).
 - `> 5% in alt-AS`: uses Asia region usage statistics. List of all region codes can be found at [caniuse-lite/data/regions](#).
 - `> 5% in my stats`: uses [custom usage data](#).
 - `> 5% in browserslist-config-mycompany stats`: uses [custom usage data](#) from `browserslist-config-mycompany/browserslist-stats.json`.
 - `cover 99.5%`: most popular browsers that provide coverage.
 - `cover 99.5% in US`: same as above, with [two-letter country code](#).
 - `cover 99.5% in my stats`: uses [custom usage data](#).
- **Last versions**:
 - `last 2 versions`: the last 2 versions for each browser.
 - `last 2 Chrome versions`: the last 2 versions of Chrome browser.
 - `last 2 major versions` or `last 2 iOS major versions`: all minor/patch releases of last 2 major versions.
- **dead**: browsers without official support or updates for 24 months. Right now it is `IE 11`, `IE_Mob 11`, `BlackBerry 10`, `BlackBerry 7`, `Samsung 4`, `OperaMobile 12.1` and all versions of `Baidu`.
- **Node.js versions**:
 - `node 10` and `node 10.4`: selects latest Node.js `10.x.x` or `10.4.x` release.
 - `last 2 node versions`: select 2 latest Node.js releases.
 - `last 2 node major versions`: select 2 latest major-version Node.js releases.

Different Type of script tags in HTML

The **HTML <script> type Attribute** is used to specify the MIME **(MIME (Multipurpose Internet Mail Extensions) is an extension of the original Simple Mail Transport Protocol (SMTP) email protocol. It lets users exchange different kinds of data files, including audio, video, images and application programs, over email.)** type of script and identify the content of the Tag. It has a Default value which is **"text/javascript"**.

Syntax:

`<script type="media_type">`

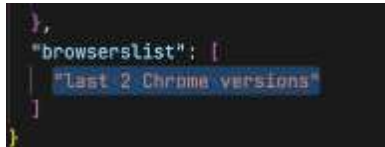
Attribute Values: It contains a single value i.e media_type which specifies the MIME type of script.

Common “media_type” values are:

- text/javascript (this is default)
- text/ecmascript
- application/ecmascript
- application/javascript

Session 4

Let's talk about JSX, Babel



It does not mean our app will not work with other browsers, it means that It will definitely work on chrome + it will work on other browsers also.

When browsers update themselves, there are some changes

Some older browsers still not support ES6 so we need to convert our code to older browser compatible means we create a polyfill for newer things which our browser does not support currently.

Polyfill = If our browser does not know about map() or promise() then polyfill means we can create our own map() which does exact same work as map() does.

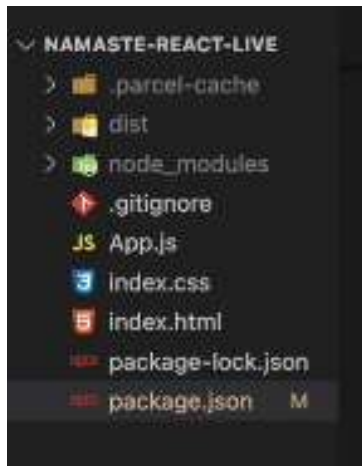
What converts our newer code to older code for older browsers?
Babel does that

If there is something new in the market, there must be some older replacement of it which is compatible in older versions of browsers so babel just convert that newer thing to that old thing so that our browser supports it.

Babel has something like browserList for browser version.
We need not to write polyfill, babel does it for us.

Babel is just a piece of JS code which takes some modern code and spit older code

How did gitignore come from in our folder sturcture?
We did "git init" to initialise a empty repo, it made gitignore automatically.



What is Tree Shaking?

Parcel does something known as Tree Shaking which means removing unwanted code.

How to know which code is unwanted?

Suppose we are using a library in our project which offers 20 functions. We install that library and whole piece of code comes in our project.

Now, say we just want to use only 2 functions out of that 20

Then those 18 are unwanted functions and that code for these 18 functions is just unwanted.

Parcel detects this un-used piece of code and ignores all of it, this optimises our app and this is called Tree Shaking.

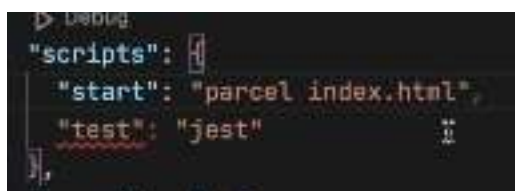
Does web-pack does not have these functions?

All bundlers have almost all these things but they are different in small small ways. Mota mota sab bundlers same hi hain bas sabme choti choti difference hain.

create-react-app uses web-pack + babel

Generally we build a script inside package.json to simplify our work and help us write less code to start our app like currently we write "npx parcel index.html"

So we make change of this script in our package.json



Now our command changes to: `npm run start`



We have a build command also: `npx parcel build index.html`

We change it to something like: **"build" : "parcel build index.html"** now we just write **npm run build**

```

"scripts": {
  "start": "parcel index.html",
  "build": "parcel build index.html",
  "test": "jest"
},

```

In commands npm is used when we want to install something inside our project whereas npx is used when we want execute something in our project.

NPM is a package manager used to install, delete, and update Javascript packages on your machine. NPX is a package executor, and it is used to execute javascript packages directly, without installing them.

npx = npm run

That is why we write npm run start which is = npx parcel index.html

Instead of npm run start we can also write npm start. Its same

Parcel even removes console.logs while doing production build.
But we see in our build.js that it did not

The screenshot shows a VS Code editor with a file named `index.54fa30f1.js` open. The file contains a large, minified JavaScript function. In the console output window on the right, the following log is visible:

```

var console: Console
Tokenization is skipped for long lines for performance reasons. The
configured via editor.maxTokenizationLineLength.
[object Object]

```

Actually what happens is, it does not remove console logs automatically, we need to configure our project in such a way that it removes console.logs

There is a package which does this

Babel plugin transform remove console

This removes our console logs

BABEL Docs Setup Try it out Videos Blog Search

babel-plugin-transform-remove-console

Example

In

```
JavaScript
console.log("foo");
console.error("bar");
```

Out

```
JavaScript
```

Installation

```
Shell
npm install babel-plugin-transform-remove-console --save-dev
```

Now we will install this plugin in our project, we can also go to npm website

Readme Code Beta 0 Dependencies

babel-plugin-transform-remove-console

This plugin removes all `console.*` calls.

Example

In

```
console.log("foo");
console.error("bar");
```

Out

Installation

```
npm install babel-plugin-transform-remove-console --save-dev
```

Parcel does not remove console logs for us, we need plugin to do that
We do `--save-dev` if we need it as our dev-dependency
Its version information goes inside `package.json` and its code goes inside node modules.

It does not work now.
We need to first configure it also
Go to that plugin website and see its usage

Installation

```
npm install babel-plugin-transform-remove-console --save-dev
```

Usage

Via `.babelrc` (Recommended)

`.babelrc`

```
// without options
{
  'plugins': ['transform-remove-console']
}
```

```
// with options
{
  'plugins': [{ 'transform-remove-console', { 'exclude': { 'error', 'warn' } } }]
}
```

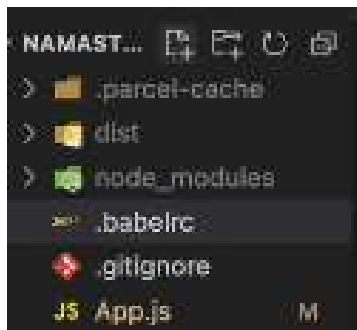
Via CLI

```
babel --plugins transform-remove-console script.js
```

Via Node API

```
require('@babel/core').transform('code', {
  plugins: ['transform-remove-console']
});
```

`.babelrc` is a configuration of babel. We create a `babel.rc`



We pick usage code from documentation and paste it inside `babel.rc`.

Exclude error and warn means exclude error and warning message from console of our production app.



Now we make **npm run build** and we do not get any console in our js file in dist.

Let's code

Suppose we have multiple siblings in a element,

We are passing an array as our children,

```
const container = React.createElement(
  "div",
  {
    id: "container",
    hello: "world",
  },
  heading, heading2
);
```

We get a warning in console for 'key'

We need to give keys as props to our children inside { } of our createElement()

Key can be anything which uniquely identifies the element.

```
const heading = React.createElement(
  "h1",
  {
    id: "title",
    key: "h1",
  },
  "Heading 1 for parcel"
);

const heading2 = React.createElement(
  "h2",
  {
    id: "title",
    key: "h2",
  },
  "Heading 2"
);
```

Why do we need a key?

Suppose we have the following structure,

When comparing two React DOM elements of the same type, React looks at the attributes of both, keeps the same underlying DOM node, and only updates the changed attributes. For example:

```
<div className="before" title="stuff" />

<div className="after" title="stuff" />
```

By comparing these two elements, React knows to only modify the className on the underlying DOM node.

When updating style, React also knows to update only the properties that changed. For example:

```
<div style={{color: 'red', fontWeight: 'bold'}} />

<div style={{color: 'green', fontWeight: 'bold'}} />
```

When converting between these two elements, React knows to only modify the color style, not the fontWeight.

After handling the DOM node, React then recurses on the children

Recurring On Children

By default, when recursing on the children of a DOM node, React just iterates over both lists of children at the same time and generates a mutation whenever there's a difference.

For example, when adding an element at the end of the children, converting between these two trees works well:

```
<ul>
  <li>first</li>
  <li>second</li>
</ul>

<ul>
  <li>first</li>
  <li>second</li>
  <li>third</li>
</ul>
```

React will match the two `first` trees, match the two `second` trees, and then insert the `third` tree.

If you implement it naively, inserting an element at the beginning has worse performance. For example, converting between these two trees works poorly:

```
<ul>
  <li>Duke</li>
  <li>Villanova</li>
</ul>

<ul>
  <li>Connecticut</li>
  <li>Duke</li>
  <li>Villanova</li>
</ul>
```

React will mutate every child instead of realizing it can keep the `Duke` and `Villanova` subtrees intact. This inefficiency can be a problem

Matlab new element ko add krne ke liye react ab poore older DOM ko destroy krega and new DOM banayega with the structure connectuct -> Duke -> Villanova
Which can reduce the performance for us and React has to do lot of work in this. So something introduced called as "keys"

Keys

In order to solve this issue, React supports a key attribute. When children have keys, React uses the key to match children in the original tree with children in the subsequent tree. For

example, adding a key to our inefficient example above can make the tree conversion efficient:

```
<ul>
  <li key="2015">Duke</li>
  <li key="2016">Villanova</li>
</ul>

<ul>
  <li key="2014">Connecticut</li>
  <li key="2015">Duke</li>
  <li key="2016">Villanova</li>
</ul>
```

Now React knows that the element with key '2014' is the new one, and the elements with the keys '2015' and '2016' have just moved.

In practice, finding a key is usually not hard. The element you are going to display may already have a unique ID, so the key can just come from your data:

```
<li key={item.id}>{item.name}</li>
```

As a last resort, you can pass an item's index in the array as a key. This can work well if the items are never reordered, but reorders will be slow.

We should never have same keys for 2 children.

How does React.createElement() works?

createElement is there in code of react which is there in node modules.

React.createElement() gives us objects.

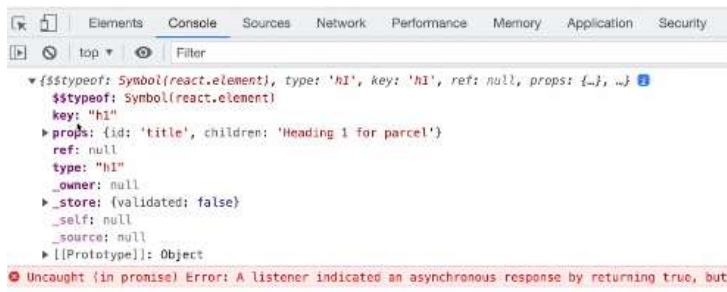
If we console log our heading

```
const heading = React.createElement(
  "h1",
  {
    id: "title",
    key: "h1",
  },
  "Heading 1 for parcel"
);

console.log(heading);

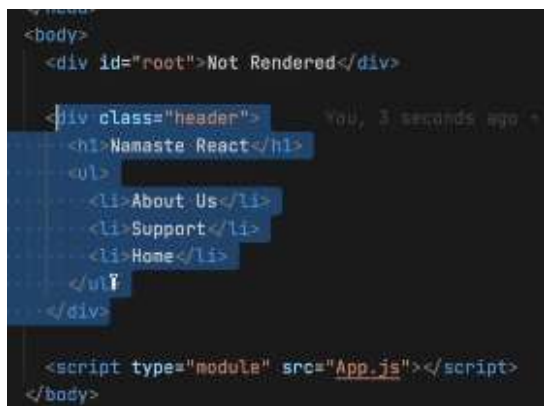
const heading2 = React.createElement(
  "h2",
  {
    id: "title",
    key: "h2",
  },
  "Heading 2"
);
```

We see an object with props as id and key



createElement gives us an object which is then converted into HTML code and put up in the DOM.

If we have to insert the below structure inside our DOM



How to do this using React?

We will create a div first and div should have 2 children h1 and ul

Instead of heading 1, our h1 will come



Instead of heading 2, ul will come and inside ul, li will come with About us, support, home
So we put all these inside array for ul

```

    },
    React.createElement("ul", {}, [React.createElement(
      "li",
      {},
      "About Us"
    ), React.createElement("ul", {}, [React.createElement(
      "li",
      {},
      "About Us"
    ), React.createElement("ul", {}, [React.createElement(
      "li",
      {},
      "About Us"
    ),
  ])],
)]),
]

```

This will become a huge mess if we do it this way.
 React.createElement() for big structure becomes huge mess
 So there should definitely be a good and clean way to do this.

So generally instead of React.createElement()
 We use something known as JSX (Javascript XML)

While writing React, the basic motive was to write HTML using JS in a efficient way
 So they first made React.createElement() API, But it made the code messy like we saw
 above.

Instead of writing React.createElement() everywhere, we import {createElement} from
 react and now we do not need to write React.createElement()
 Just writing createElement() does the job for us.

```

import { createElement } from "react";
import ReactDOM from "react-dom/client";

// React.createElement => Object => HTML(DOM)

const heading = createElement(
  "div",
  {
    id: "title",
    key: "h2",
  }
)

```

We can also use "ce" instead of createElement() by doing

```

import { createElement as ce } from "react";
import ReactDOM from "react-dom/client";

```

But it did not do any big change, code still looks messy
 So **JSX was introduced**

How to create h1 tag using JSX?

```
const heading = <h1>Namaste React</h1>;

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

//passing a react element inside the root

//async defer
root.render(heading);
```

We can give id and key like:

```
const heading2 = (
  <h1 id="title" key="h2">
    Namaste React
  </h1>
);
```

If we are writing it in multiple line, we need to use **parenthesis ()**

```
const heading2 = (
  <h1 id="title" key="h2">
    Namaste React
  </h1>
);
```

Is JSX, HTML inside JS?

JSX allows us to write HTML elements in JavaScript and place them in the DOM without any createElement() and/or appendChild() methods. JSX converts HTML tags into react elements. **You are not required to use JSX, but JSX makes it easier to write React applications.**

JSX is not HTML, it is HTML like syntax, fancy way to write HTML inside JS but it is not HTML inside JS.

Now our final code, looks like:

```

import { createElement as ce } from "react";
import ReactDOM from "react-dom/client";

// React.createElement ⇒ Object ⇒ HTML(DOM)

const heading = ce(
  "h1",
  {
    id: "title",
    key: "h2",
  },
  "Namaste React"
);

// JSX ??

const heading2 = (
  <h1 id="title" key="h2">      You, 3 minutes ago • Uncommitted change
    Namaste React
  </h1>
);

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

//passing a react element inside the root

//async defer
root.render(heading);

```

There are certain major differences between HTML and JSX

In JSX, we write in camel-case => tab-index in HTML becomes tabIndex in JSX
class in HTML becomes className in JSX

In HTML almost all tags have an opening and a closing tag except probably a few like `
`

In JSX, however, any element can be written as a self-closing tag, for example: `<div/>`

Example:

```
const string = <img
src={user.avatarUrl} />;
```

HTML elements have attributes.
e.g., `maxlength` in `<input maxlength="16" />`

JSX elements have props.
e.g., `maxLength` in `<input maxLength="16" />`

In HTML, multiple elements can be returned.
For example:

```

<ul>
  <li>unordered list
  <ol>
    <li>ordered list</li>
    <li>ordered list</li>
    <li>ordered list</li>
  </ol>
</li>
<li>unordered list</li>
<li>unordered list</li>

```

Nested JSX must return one element, which we'll call a parent element that wraps all other levels of nested elements:

```

<div>
  <p>pink</p>
  <p>yellow</p>
  <p>green</p>
</div>

```

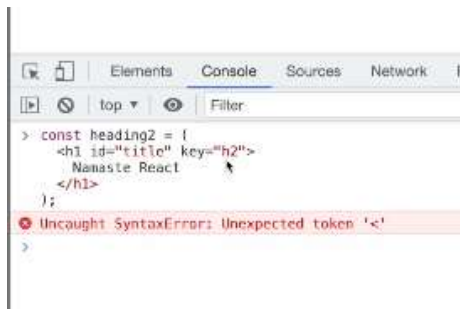
Without the wrapper element, JSX won't transpile. In React, we can render JSX directly into HTML DOM

using React rendering API, aka *ReactDOM*. The formula for rendering React elements seems like this:
ReactDOM.render(componentToRender, targetNode)
ReactDOM.render() must be called after the JSX elements declarations.

How is below code executed by JSX?

```
// JSX ??  
  
const heading2 = (  
  <h1 id="title" key="h2">  
    Namaste React  
  </h1>  
)
```

Our browser does not understand this code



But Babel understands it, Babel is a JS compiler. Its another JS library which takes some piece of code and returns some code.

Babel reads our code line by line and make an **Abstract Syntax tree (AST)**

What does JSX do is, JSX uses `React.createElement()` behind the scenes and it gets converted to object which is then fitted to DOM.

Who is converting JSX to `React.createElement()` ?
Babel, does that

Advantages of JSX

1. JSX in `React.js` **makes code easy to read and write**
2. One of the advantages of JSX is that **React creates a virtual DOM (a virtual representation of the page) to track changes and updates**. Instead of rewriting the entire HTML, React modifies the DOM of the page whenever the information is updated. This is one of the main issues React was created to solve.

Write code such that it is easily readable by humans.

Babel comes along with parcel

We have 2 package-lock.json

1 is we have created and other is inside node_modules which keeps track of all packages and libraries inside node modules (Transitive Dependencies).

JSX is not imported as its not a package. JSX is a syntax

Below code is known as JSX expression

```
const heading2 = <h1 id="title" key="h2">
  Namaste React
</h1>
);
```

React Element

h1 tag is called as element inside React

From now on, we will not use React.createElement()

We will use JSX

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

// JSX ⇒ React.createElement ⇒ Object ⇒ HTML(DOM)

const heading2 = (
  <h1 id="title" key="h2">
    Namaste React
  </h1>
);

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

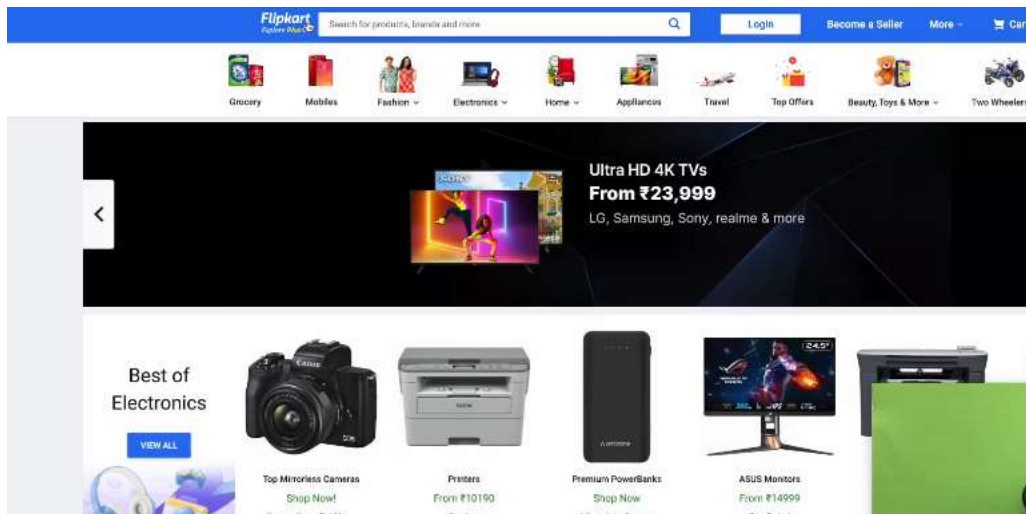
//passing a react element inside the root

//async defer
root.render(heading2);
```

What are components?

Everything is component in react.

Inside flipkart.com



Navbar carousel, products block everything is a component.

There are **2 types** of components in react

1. **Functional component (New way of writing code)**
2. **Class Component (OLD way of writing code)**

We will mostly use Functional Component

Functional Components

They are nothing but a function and is hoisted same as a normal function in JS.

```
// React Component
// - Functional - NEW - I'll use this most of the time

const HeaderComponent = () => {
```

It is JS function which returns some piece react element or a component or JSX.

```
// React Component
// - Functional - NEW - I'll use this most of the time

const HeaderComponent = () => {
  return <h1>Namaste React functional component</h1>;
};
```

Note:

Functional component or for any component, Name starts with Capital Letter. It is not Mandatory but it is a normal convention, good practice to use this.

Functional component is a normal function which returns some element or a components or some piece of JSX.

Suppose we want to build a big HTML structure

```
const HeaderComponent = () => {  
  return <div><h1>Namaste React functional component</h1><h2>This is a h2 tage</h2></div>  
};
```

When we write this complicated structure and our code goes in multiple lines, we have wrap it inside ()

```
const HeaderComponent = () => {  
  return (  
    <div>  
      <h1>Namaste React functional component</h1>  
      <h2>This is a h2 tage</h2>  
    </div>  
  );  
};
```

Some people skip writing "return" to look cool

```
const HeaderComponent = () => {  
  <div>  
    <h1>Namaste React functional component</h1>  
    <h2>This is a h2 tage</h2>  
  </div>  
};
```

Above code works already fine
Both component are same

```
const HeaderComponent = () => {  
  return (  
    <div>  
      <h1>Namaste React functional component</h1>  
      <h2>This is a h2 tage</h2>  
    </div>  
  );  
};  
  
const HeaderComponent2 = () => (  
  <div>  
    <h1>Namaste React functional component</h1>  
    <h2>This is a h2 tage</h2>  
  </div>  
);
```

We can also write normal function instead of Arrow function

```
const HeaderComponent = function () {
  return (
    <div>
      <h1>Namaste React functional component</h1>
      <h2>This is a h2 tage</h2>
    </div>
  );
};
```

When we have to render our React element we write like this

```
const heading = (
  <h1 id="title" key="h2">
    Namaste React
  </h1>
);
```

For above element, we write like

```
//async defer
root.render(heading);
```

For Component how do we render it?

```
const HeaderComponent = () => {
  return (
    <div>
      <h1>Namaste React functional component</h1>
      <h2>This is a h2 tage</h2>
    </div>
  );
};

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

//passing a react element inside the root

//async defer
root.render(<HeaderComponent />);
```

Only difference between React element and React component is, react component is an Arrow function which return some JSX. whereas element is an simple JSX (not a function).

A React element is an object representation of a DOM node. A component encapsulates a DOM tree. Elements are immutable i.e once created cannot be changed. The state in a component is mutable.

Element

An element is always gets returned by a component.

The element does not have any methods.

A React element is an object

Component

A component can be functional or a class that optionally takes input and returns an element.

Each component has its life cycle methods.

A component encapsulates a DOM tree.

representation of a DOM node.

Elements are immutable i.e once created cannot be changed.

An element can be created using `React.createElement()` with type property.

We cannot use React Hooks with elements as elements are immutable.

Elements are light, stateless and hence it is faster.

The state in a component is mutable.

A component can be declared in different ways like it can be an element class with `render()` method or can be defined as a function.

React hooks can be used with only functional components

It is comparatively slower than elements.

How to use React element inside React Component?

Let say we want our heading element to be used inside HeaderComponent

```
const heading = (<h1 id="title" key="h2">  
  Namaste React  
</h1>  
);  
  
const HeaderComponent = () => {  
  return (<div>  
    <h2>Namaste React functional component</h2>  
    <h2>This is a h2 tage</h2>  
  </div>  
  );  
};
```

We do something like: `{heading}`

```
const HeaderComponent = () => {  
  return (<div>  
    {heading}  
    <h2>Namaste React functional component</h2>  
    <h2>This is a h2 tage</h2>  
  </div>  
  );  
};
```

Output looks like:



If our heading is a Component then we use it like this inside our component

```
const Title = () => {
  <h1 id="title" key="h2">
    Namaste React
  </h1>
};

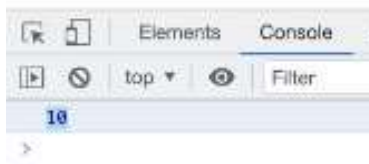
const HeaderComponent = () => {
  return [
    <div>
      <Title />
      <h2>Namaste React functional component</h2>
      <h2>This is a h2 tage</h2>
    </div>
  ];
};
```

There is another way of writing this:
Just call it like a normal JS function

```
const HeaderComponent = () => {
  return [
    <div>
      {Title()}
      <h2>Namaste React functional component</h2>
      <h2>This is a h2 tage</h2>
    </div>
  ];
};
```

Inside JSX we can write anything in JS inside { } and it will render

```
var xyz = 10;
const HeaderComponent = () => {
  return [
    <div>
      {
        console.log(xyz);
      }
      <h2>Namaste React functional component</h2>
      <h2>This is a h2 tage</h2>
    </div>
  ];
};
```

We can also do calculation there

```
var xyz = 10;
const HeaderComponent = () => {
  return (
    <div>
      {1 + 2}
      <h2>Namaste React functional component</h2>
      <h2>This is a h2 tage</h2>
    </div>
  );
};
```

So while writing JS in JSX we need to write it inside { }

That is why we call element like {title} inside Component as title is just a normal JS variable.

Let say we have an API call and we get some data

Suppose our API got hacked and our API gets some malicious code in our code

And we render that data in component.

This attack is known as **Cross Site Scripting (XSS)**

```
const data = api.getData();

const HeaderComponent = () => {
  return (
    <div>
      {data}
      <h2>Namaste React functional component</h2>
      <h2>This is a h2 tage</h2>
    </div>
  );
};
```

Still JSX takes care of this, It makes sure our app is safe.

It **sanitization**

JSX expressions { } automatically take care of encoding HTML before rendering, which means even if u don't sanitise your input your webpage is XSS safe.

All I can find on Google around sanitizing HTML are packages that sanitize Markdown. So my intermediate solution is to convert my API-returned HTML to Markdown to-markdown, and the convert *that* back to HTML using marked, which includes a sanitizer.

We need to use 2 libraries to first mark-down code and then convert it back to HTML using marked which automatically sanitises it and keep us away from attacks.

```
import toMarkdown from 'to-markdown'
import marked from 'marked'

var sampleApiResponse = '<h1>I am a heading</h1><p>I am a paragraph</p>';

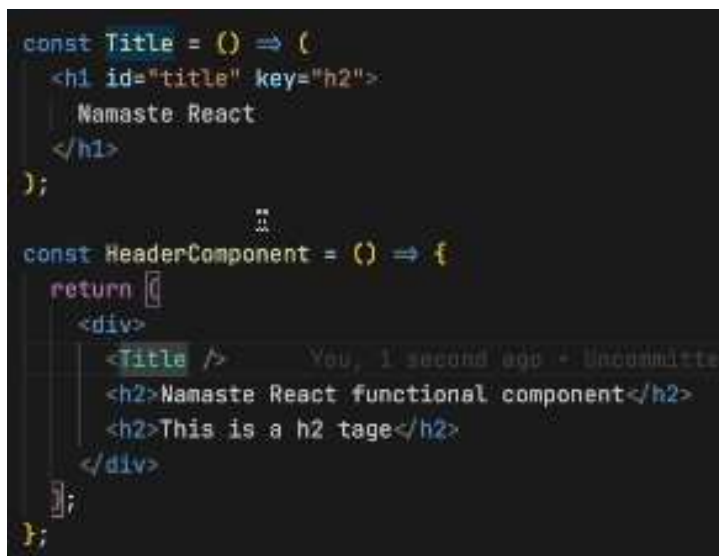
var MyComponent = React.createClass({
  render: function(){
    var markup = this.props.htmlToRender
    var sanitize = function(htmlString){
      return marked(toMarkdown(html), {sanitize: true})
    }

    return (
      <div dangerouslySetInnerHTML={sanitize(markup)}></div>
    );
  }
})

ReactDOM.render(
  <MyComponent htmlToRender={sampleApiResponse} />,
  document.getElementById('react-wrapper')
)
```

Component Composition

In React, we can make components more generic by accepting props, which are to React components what parameters are to functions. Component composition is **the name for passing components as props to other components, thus creating new components with other components**

A screenshot of a code editor with a dark background. It shows two JavaScript function components. The first component, 'Title', returns a single JSX element: an h1 tag with id 'title' and key 'h2' containing the text 'Namaste React'. The second component, 'HeaderComponent', returns a JSX fragment containing three elements: a 'Title' component (which is being passed as a prop), an h2 tag with the text 'Namaste React functional component', and another h2 tag with the text 'This is a h2 tage'. The code is written in a functional style using 'const' and '=>' for arrow functions.

```
const Title = () => (
  <h1 id="title" key="h2">
    Namaste React
  </h1>
);

const HeaderComponent = () => {
  return [
    <div>
      <Title />
      <h2>Namaste React functional component</h2>
      <h2>This is a h2 tage</h2>
    </div>
  ];
};
```

React Reconciliation

When you use React, at a single point in time you can think of the **render()** function

as creating a tree of React elements. On the next state or props update, that `render()` function will return a different tree of React elements. React then needs to figure out how to efficiently update the UI to match the most recent tree.

There are some generic solutions to this algorithmic problem of generating the minimum number of operations to transform one tree into another. However, the algorithms have a complexity in the order of $O(n^3)$ where n is the number of elements in the tree.

If we used this in React, displaying 1000 elements would require in the order of one billion comparisons. This is far too expensive. Instead, **React implements a heuristic $O(n)$ algorithm based on two assumptions:**

- 1. Two elements of different types will produce different trees.**
- 2. The developer can hint at which child elements may be stable across different renders with a **key prop**.**

In practice, these assumptions are valid for almost all practical use cases.

