**Best practices to write code:**

1. Use the optional chaining.
2. Destructure the props properly

Instead of this:

Instead of this:

const RestaurantCard = (props) =>{

const { resData } = props;

return(

<div>

<h3>{resData.info.name}</h3>

<h4>{resData.info.cuisines.join(", ")}</h4>

<h4>{resData.info.costForTwo}</h4>

<h4>{resData.info.avgRating} stars</h4>

<h4>{resData.info.sla.deliveryTime} minutes</h4>

</div>

)

}

Do this:

const { resData } = props;

const {name, cuisines, costForTwo, avgRating} = resData?.info;

const {deliveryTime} = resData?.info?.sla;

<h3>{name}</h3>

<h4>{cuisines.join(", ")}</h4>

<h4>{costForTwo}</h4>

<h4>{avgRating} stars</h4>

<h4>{deliveryTime} minutes</h4>

1. Whenever you are looping over something always give a key to uniquely identify each item.

When there is no unique identifier we can use index as the key, but React official documentation itself says don’t use the index as key when the order of items changes. (Ask your backend developers to add the unique key for each item)

{ResData.map((restaurant, index)=> <RestaurantCard resData={restaurant} key={index}></RestaurantCard>)}

Major learning’s in terms of latest React:

* .
* .
* .
* .

Yes! If you can **successfully implement JWT authentication** in your simple web app with:

✅ **Login & Logout functionality**  
✅ **Storing & using the token correctly** (localStorage or HttpOnly cookies)  
✅ **Protecting private routes**  
✅ **Handling token expiration & redirects**

Then you are **good enough** with authentication for a frontend developer role. 🎯

Learning’s to be done:

1. JS DOM manipulation. (Everything like innerHTML vs innerText and all)
2. Check what is CDN? (from original episode 1 notes)
3. Check what is cross-origin in CDN links? (from original episode 1 notes)
4. Why do we need two separate two CDN links like React and React-DOM why cannot we mix those into a single file? (from original episode 1 notes)
5. Why React is called as Library but not a Framework? (from original episode 1 notes)
6. Async defer in script tag?
7. More about arrow functions and different scenarios with arrow functions.
8. What is the difference between the ~ and the ^ before the version in package.json?
9. What is the difference between the package.json and package-lock.json?
10. What is node\_modules folder?
11. What are transitive dependencies?
12. Why your React app is fast?
13. I need to know the exact difference between <Component/> vs <Component></Component>
14. Read about: ^ - caret and ~ - tilda ● Read about Script types in html (MDN Docs)
15. Read about Script types in html (MDN Docs)
16. Role of type attribute in script tag? What options can I use there?
17. Entire CSS.
18. Add responsive design to the Swiggy clone application.
19. How to efficiently compare two JS objects.
20. How to Debug React applications.

**Episode 1:**

* React is the most popular JavaScript library for building the large scale frontend applications.
* **Hello world program in JavaScript:**

Creating an element:

const heading=document.createElement(“h1”);

Adding content to the element:

heading.innerHTML=”Hello world from JavaScript”

finding out the element which is having “id” as “root”:

const root = document.getElementById(“root”)

Adding the element to the root:

root.appendChild(heading); //added the newly created h1 tag to the element which is having root as the id.

Browsers understands the above terms like document, createElement, getElementById etc because every browser have the JavaScript engine, which executes the JavaScript code.

* But the browser cannot understand the React code, we need to manually add the React into our project so that browser will understand.
* There are different ways in which we can add the React into our project:

1. Using CDN i.e., content delivery network:

Go to chrome and search for “CDN react”, in the official page, we can find the CDN links of React and ReactDOM, copy those links and paste it in your code in the body tag.

1. Using create-react-app package. (npx create-react-app swiggyclone)
2. Using npm (npm install React, npm install react-dom)

* **Hello world program in React:**

Creating an element in React:

Creating an element in React is the responsibility of the React, we should use createElement of React to create the element. It takes three arguments:

1. Element to be created.
2. Any attributes
3. Children

const heading = React.createElement(“h1”, {}, “Hello world from React JS”)

Creating root:

Creating root and rendering something inside the root, is the responsibility of the ReactDOM.

const root = ReactDOM.createRoot(document.getElementById(“root”));

Rendering h1 tag into the root:

root.render(heading);

* React.createElement() will create a React element, React element is nothing but a normal JavaScript object. This object contains the type property which have the value of the type of element we are creating, it also have the props, props contains the attributes and children of the element.
* The render() method is responsible for converting this JavaScript object into HTML element and put it on to the DOM.

/\*I want this HTML structure:

<div id="parent">

    <div id="child">

        <h1>I am h1 tag</h1>

    </div>

</div>

\*/

Solution:

const parent = React.createElement("div", {id: "parent"}, React.createElement(

    "div", {id:"child"}, React.createElement(

        "h1", {}, "I am h1 tag"

    )

))

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

root.render(parent);

console.log(parent);

-------------------------------------------------------------------------------------------------------------------------------

/\*I want this HTML structure: Siblings inside a div

<div id="parent">

    <div id="child">

        <h1>I am h1 tag</h1>

        <h2>I am h2 tag</h2>

    </div>

</div>

\*/

const parent = React.createElement("div", {id:"parent"}, React.createElement(

    "div", {id:"child"}, [React.createElement("h1",{}, "I am h1 tag"),

        React.createElement("h2", {}, "I am h2 tag")

    ]

))

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

root.render(parent)

By using array for the third argument we can achieve the siblings structure.

/\*I want this HTML structure:

<div id="parent">

    <div id="child1">

        <h1>I am child1 h1 tag</h1>

        <h2>I am child1 h2 tag</h2>

    </div>

    <div id="child2">

        <h1>I am child2 h1 tag</h1>

        <h2>I am child2 h2 tag</h2>

    </div>

</div>

\*/

const parent = React.createElement("div", {id:"parent"}, [

    React.createElement("div", {id:"child1"}, [React.createElement("h1",{},"I am child1 h1 tag"),

        React.createElement("h2",{}, "I am child1 h2 tag")

    ])

], React.createElement("div", {id:"child2"}, [React.createElement("h1",{}, "I am child2 h1 tag"),

    React.createElement("h2", {}, "I am child2 h2 tag")

]))

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

root.render(parent);

Here for creating simple HTML structures we need to write very complex code in the React, React is making our life hard. To avoid this type of complexity we can use the **JSX**, **JSX** makes our lives easy when we are creating the React elements.

* render means replace not append.

**Async and defer**:

* When you load a webpage there are two major things happen in your browser, they are HTML parsing and loading of the scripts.
* Loading of scripts contains two parts fetching the script from the network and second one is executing the script.

**Without async or defer attributes:**

* HTML parsing will start, once it sees the script tag the HTML parsing will be stopped then the script will be loaded and executed, then only the HTML parsing will gets continued.
* If we are having multiple script tags then the order of execution of scripts tags are maintained.

**With async attribute:**

* HTML parsing will start, once it sees the script tag, the script will start getting loaded/downloaded and also the HTML parsing will continue until the script is loaded/downloaded, once the script is loaded/downloaded then the HTML parsing will stop and script will be executed.
* If we are having multiple scripts then the script which is loaded first will be executed and there is no guarantee for order of execution.

**With defer attribute:**

* HTML parsing will start, if it sees the script tag then the script is downloaded and will only be executed after the entire HTML parsing is completed.
* If there are multiple script tags, then the order of execution of scripts is guaranteed. First script is executed before the second script is executed.

**Key Takeaways**

✅ **Use async** for scripts that don’t depend on each other or the DOM (e.g., analytics, ads).  
✅ **Use defer** for scripts that depend on the full DOM (e.g., manipulating elements).  
✅ **If no async or defer is used**, place scripts **at the end of <body>** to prevent blocking.

**Episode 2:**

* For knowing the current branch name: **git branch --show-current**
* For renaming the branch: **git branch –M oldBranchName newBranchName**
* When we run git init in VS Code, the default branch is master, but GitHub's default branch is main. To keep them consistent, I renamed my local branch from master to main in VS Code.
* git add .
* git commit –m “Episode2 part1”
* git push origin main
* when we do the npm init, package.json file will be created, it is just a configuration of NPM , it keeps track of what dependencies are installed and what version of dependencies are installed.
* With npm init we configured our project with npm, now we can get all the packages we required into our project.
* bundler just bundles your code into a single file (compressed, cleaned, minified) and makes it ready for production. Webpack, Parcel, Vite are examples of Bundlers.
* When we install any package, that package will be added into the package.json file. Package-lock.json file will be created if it not already exists, node modules folder will be created if not already exists.
* React is fast and also it uses the bundlers which will make our React application faster.

#Parcel

- Builds the development build

- Build the production build

- Creates the local server and hosts our application on it and runs our application on port number 1234

- HMR = Hot Module Replacement (Refreshes our React page in browser automatically, when we make some code changes)

- But how do parcel automatically refresh the page? It uses the "file watch algorithm" (written in C++) which will look for changes and if something changes it will do the build again and the React page in the browser gets refreshed automatically.  (Check this once again, my page is not refreshing automatically)

- Parcel also uses the cache for the faster builds. When you build it first time it will create a folder called .parcel-cache and will store the cache there, so for the next builds Parcel uses that cache for the faster building.

- Parcel will also do the image optimization. (The most expensive thing in your browser is to load images into your page)

- When we do the production build the Parcel will minify the file.

- Parcel will do bundling

- Parcel will do file compression.

- Parcel will do consistent hashing.

- Parcel will do Code splitting

- Parcel will do differential bundling- (Means creates different bundles so your application will be compatible to run on different browsers and different versions)

- Parcel gives you better Error handling suggestions.

- Parcel by default hosts your application on http, but Parcel also gives a way to host it on the https.

- Parcel supports the tree shaking: tree shaking algorithm will remove the unused code for you. Suppose if there is some code and you are not using it, then Parcel will remove it.

- Parcel supports the lazy loading.

- Parcel create the different bundles for development and production.

In the <script src=”app.js”> line make it as <script type=”module” src=”app.js”>

But why we making type as module?

Ans: If we don’t use type as module, we will get a error like: “**Browser scripts cannot have imports or exports**”. Without type as module, the script will be treated like a normal browser script or a normal JavaScript script, but our script is a React script, to tell it is a React script we need to write the type as module.

* Always import react-dom from ‘react-dom/client’

import ReactDOM from 'react-dom/client';

How to make our application compatible with the older versions?

We need to make some configuration for the browserslist package in package.json:

Eg: Manually add like this in the package.json

1. browserslist: [“last 10 Chrome version”]: It means our application will definitely be compatible to run on the last 10 versions of google chrome and it may or may not run on other versions or other browsers.
2. browserslist: [“last 10 Chrome version”, “last 2 Firefox version”]
3. browserslist: [“last 2 versions”]: Works on the last 2 versions of all the browsers.

**Episode 3:**

**React element:**

const heading = <h1> This is a heading element </h1>

**React component:**

const Heading = () =>{

return(

<h1>This is a Heading component</h1>

)

}

Read the complete notes of Episode 3 its just simply awesome, no one teaches such content of React.

**Episode 4:**

| **Syntax** | **Meaning** | **Example** | **Where to Use** |
| --- | --- | --- | --- |

|  |  |  |  |
| --- | --- | --- | --- |
| * ./ | Current folder | ./Images/foodlogo.png | HTML, CSS, JS |

|  |  |  |  |
| --- | --- | --- | --- |
| * ../ | Move up one level | ../Images/foodlogo.png | HTML, CSS, JS |

|  |  |  |  |
| --- | --- | --- | --- |
| * / | Root directory | /Images/foodlogo.png | Web servers, React public folder |

.res-card:hover{

    border: 1px solid black;

    cursor: pointer;

}

**.res-info h6** means: **Apply styles only to** <h6> elements **that are inside** an element with the .res-info class.

 white-space: nowrap; /\* Prevents text from breaking into new lines \*/

 overflow: hidden; /\* Hides extra text  \*/

 /\* text-overflow: ellipsis; \*/

Q: see currently on my res card I have added on hover border 1px solid black, but in the real swiggy app I can see when I hover on it the card goes deep, I mean it really looks like someone has pressed it.

Ans: Yes! That effect is called a **"press down" or "depth" effect**, and it’s achieved using **box-shadow and transform** in CSS. Check this.

**Topics to Learn for a Better Understanding:**

**1️⃣ Transforms (CSS transform Property)**

* translateX() and translateY() → Move elements horizontally/vertically.
* scale() → Resize elements.
* rotate() → Rotate elements.
* skew() → Tilt elements.
* Learn how transform: translateY(2px); makes the card "press down".

**2️⃣ Box Shadows (CSS box-shadow Property)**

* Syntax: box-shadow: offsetX offsetY blurRadius color;
* Example: box-shadow: 0px 2px 5px rgba(0, 0, 0, 0.2);
* Understand how shadows create **depth effects**.

**3️⃣ Transitions (CSS transition Property)**

* Helps to **animate changes smoothly** instead of instantly.
* Syntax: transition: property duration timing-function;
* Example: transition: all 0.2s ease-in-out;
* Learn different timing functions like ease, ease-in-out, linear.

**4️⃣ Hover Effects (CSS :hover Selector)**

* :hover allows styles to change **when an element is hovered**.
* Used for interactive UI effects.

**When you want to dynamically pass the data to the component, we will use the props.**

Passing different data as props:

1. Strings

<Component title="Welcome" />

// OR (less common, more verbose)

<Component title={"Welcome"} />

1. Numbers

<Component count={5} />

1. Booleans

<Component isActive={true} />

// Or shorthand (true only)

<Component isActive />

1. Arrays

<Component items={["Apple", "Banana", "Cherry"]} />

Or simply an array name:

const fruits = ["Apple", "Banana", "Cherry"];

<Component items={fruits} />

1. Objects

<Component user={{ name: "Alice", age: 25 }} />

1. Functions

<Component onClick={() => console.log("clicked")} />

Or

<Component onClick={handleOnClick} />

1. Variables

const name = "React";

<Component framework={name} />

1. Null/undefined

<Component value={null} />

<Component value={undefined} />

1. **Children (content inside the tags)**

<Component>Hello World</Component>

**Correct Ways to Use onClick in React**

1. **Pass a function directly**

function test(){

console.log(‘test’)

}

<button onClick={test}>click</button>

1. **Use arrow function to pass arguments**

function handleClick(fruit) {

console.log("Clicked:", fruit);

}

<button onClick={() => handleClick("apple")} >click</button>

You **must use an arrow function** here to avoid calling it immediately.

**Common Mistakes to Avoid**

* Calling the function immediately

<button onClick={handleClick()} >Click</button>

This runs the function **while rendering**, not when you click.

Instead of the above write the below one:

<button onClick={handleClick}>Click</button>

* If you want to pass arguments you must wrap the function
  1. <button onClick={()=> handleClick(“test”)}> Click Me </button>

Or

* 1. <button onClick={function() { handleClick(“test”)}}> Click Me </button>

**If you want to pass arguments, you must wrap the function in another function** — either with an arrow function or a regular one.

* Console.log(“anything”) always logs anything and returns undefined.

const result1 = console.log("Hello World");

console.log("Returned:", result1); // 🔥 Logs "Hello World", then "Returned: undefined"

const result2 = console.log(123 + 456);

console.log("Returned:", result2); // Logs 579, then "Returned: undefined"

const result3 = console.log({ name: "Adil" });

console.log("Returned:", result3); // Logs the object, then "Returned: undefined"

Check the quiz of chatgpt on this topic

// src={"https://media-assets.swiggy.com/swiggy/image/upload/fl\_lossy,f\_auto,q\_auto,w\_660/"+resData.info.cloudinaryImageId}

        src={`https://media-assets.swiggy.com/swiggy/image/upload/fl\_lossy,f\_auto,q\_auto,w\_660/${resData.info.cloudinaryImageId}`}

**Episode 5:**

**✅ Meaning of ./, ../**

| **Syntax** | **Meaning** | **Example Use** |
| --- | --- | --- |
| ./ | Same folder (current directory) | import Card from './Card'; |
| ../ | One level up (parent folder) | import App from '../App'; |
| ../../ | Two levels up | import Config from '../../config/setting.js'; |
|  |  |  |

Check the quiz of chatgpt on paths.

**Can we have the both named and default export for a single component like this:**

export Header = () =>{

return(

<div>

<h1>The Header component</h1>

</div>

)

}

export default Header;

**Yes, technically it works**, but it’s a bit unusual and not a recommended practice because there is no real benefit doing this.

* When we say React is fast, we mean React is fast in DOM manipulation React is very efficient in DOM manipulation.
* Whenever the state variable is updated React will re-render the component.

Everything that you can do with React, we can also do with simple JavaScript also but why do we need React?

Answer: React makes the developer's life easy by efficiently updating the DOM. For efficiently updating the DOM React uses the virtual DOM. With React we don’t need to manually handle the UI updates.

The Big Problem in JS is Updating the DOM:

In plain HTML + JS, if you want to update just one part of the page (say, a counter when a button is clicked), you'd have to:

1. Grab the element with document.querySelector.
2. Manually update its content.

Example

<p id="count">0</p>

<button onclick="increase()">Increase</button>

<script>

let count = 0;

function increase() {

count++;

document.getElementById('count').innerText = count;

}

</script>

It’s fine for tiny stuff, but imagine updating 10+ elements, and keeping track of user inputs, conditional UI, etc. It becomes a nightmare.

**React Makes It Elegant with State**

With React, you don’t touch the DOM directly. Instead, you:

* Change the **state**
* React takes care of updating the DOM *only* where needed

Example using React:

import { useState } from "react";

const Counter = () => {

const [count, setCount] = useState(0);

return (

<>

<p>{count}</p>

<button onClick={() => setCount(count + 1)}>Increase</button>

</>

);

};};

 No getElementById

 No manual update

 No DOM headaches

 React just *reacts* to state changes 🧠💥

**State in React – Initial Value Rule:**

**The initial value you pass to useState() is used only during the first render.**

After that, updating the state **won’t re-run the initial value function** or reset the value — it just keeps the latest updated state.

const [data, setData] = useState(() => expensiveFunction());

Even here, expensiveFunction() runs **only once**, not on every re-render.

Here the data state variable initial value is the return value of the expensiveFunction.

**Why pass a function to useState()?**

lazy initialization

When you pass a **function** to useState, it is called **only once** during the initial render to compute the initial state.

This is useful when the initial value is **expensive to calculate**, because it avoids running the logic on every re-render.

Syntax: const [value, setValue] = useState(() => heavyCalculation());

 heavyCalculation() is **not called immediately**.

 Instead, React calls it **only during the first render**.

 On future re-renders, the function is **not run again** — React uses the stored/updated state value.

Compare with:

const [value, setValue] = useState(heavyCalculation());

heavyCalculation() is called **immediately when the file runs**, even before the component renders.

On future re-renders, the function is **not run again** — React uses the stored/updated state value.

* The logic of updating the UI is called as re-rendering, React is super fast in rendering and re-rendering.

Q: Why React is fast?

Answer: React is fast because it can update the DOM efficiently by comparing the new virtual DOM with previous virtual DOM and update the actual DOM using the Diff algorithm and this process is known as React reconciliation or React fiber.

Virtual DOM is nothing but a representation or copy of an actual DOM. It is just a JS object (we can confirm this by doing console log of component eg: console.log(<Body></Body>)).

At first we will have one virtual DOM, for any change the new virtual DOM will be created, now these two are compared to update the actual DOM. Comparing virtual DOMs is easy because comparing JS objects is easy than comparing HTML code, so we are using the concept of virtual DOM.

Very simple basic thing: useState() returns an array

1st way:

const [res, setRes] = useState(resList);

2nd way:

const arr = useState(resList);

const [res, setRes] = arr;

3rd way:

const arr = useState(resList);

const res = arr[0];

const setRes = arr[1];

**Episode 6:**

**Fetching the data:**

const fetchData = async() =>{

const data = await fetch(“URL of the data to be fetched”);

const json = await data.json();

//Now, do whatever you want with the json data.

}

**Searching the data using Debouncing:**

useEffect(()=>{

const timer = setTimeout(()=>{

searchRestaurants();

}, 300)

return ()=>{

clearTimeout(timer)

}

},[searchText])

**Here, when the searchText state variable changes, the useEffect is triggered. It sets a timer to call the searchRestaurants function after 300 ms — but not immediately.**

However, if searchText changes again **before** the 300 ms are up, the component re-renders, and the cleanup function (returned from useEffect) is called. This cleanup function clears the previously scheduled timer.

As a result, searchRestaurants() is **not executed**.

This process continues — the searchRestaurants() function only gets called **after the user stops typing for 300 ms**, effectively implementing a **debounce** behavior.

Question: See most times we use const for useState variable, but how can we modify those if they are const?

const[restaurants, setRestaurants] = useState([]);

Ans: