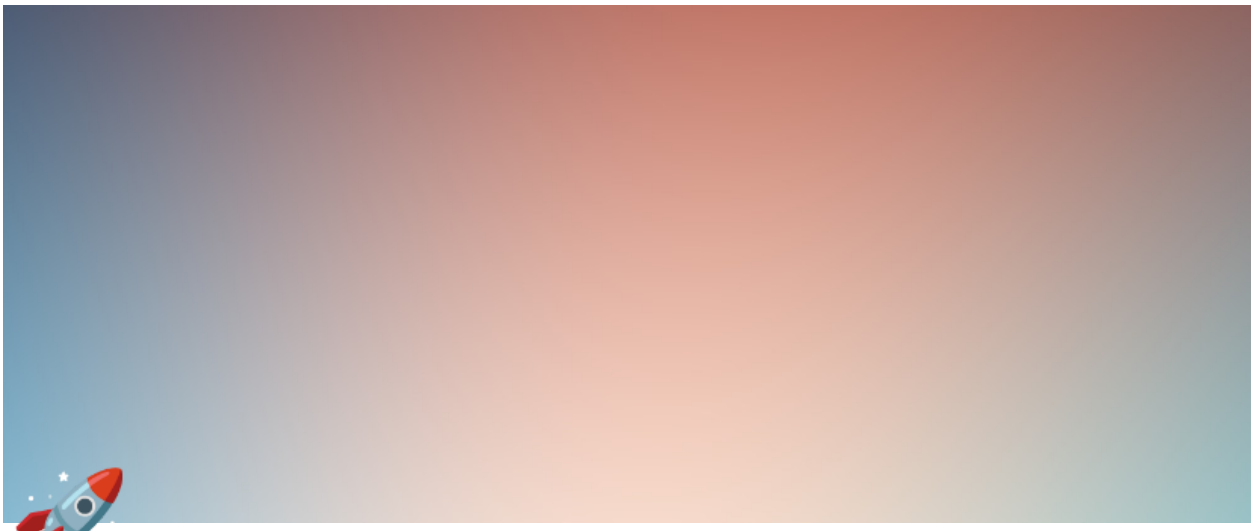


Inception! (Namaste-React)

💡 Please make sure to follow along with the whole ***"Namaste React"*** series, starting from Episode-1 and continuing through each subsequent episode. The notes are designed to provide detailed explanations of each concept along with examples to ensure thorough understanding. Each episode builds upon the knowledge gained from the previous ones, so starting from the beginning will give you a comprehensive understanding of React development.

💡 I've got a quick tip for you. To get the most out of these notes, it's a good idea to watch **Episode-1** first. Understanding what ***"Akshay"*** shares in the video will make these notes way easier to understand.

> **So Let’s begin our Namaste React Journey**
>
> - In this course we will study how the React concepts are actually applied into the industry i.e. into the real world projects.
> - So are you ready to fall in love with React????

Introducing React.

Q) What is React? Why React is known as ‘React’?

React is a JavaScript Library. The name ‘React’ was chosen because the library was designed to allow developers to react to changes in state and data within an application, and to update the user interface in a declarative and efficient manner.

Q) What is Library?

Library is a collections of prewritten code snippets that can be used and reused to perform certain tasks. A particular JavaScript library code can be plugged into application code which leads to faster development and fewer vulnerabilities to have errors.

Examples: React, jQuery

Q) What is Framework?

Framework provides a basic foundation or structure for a website or an application.

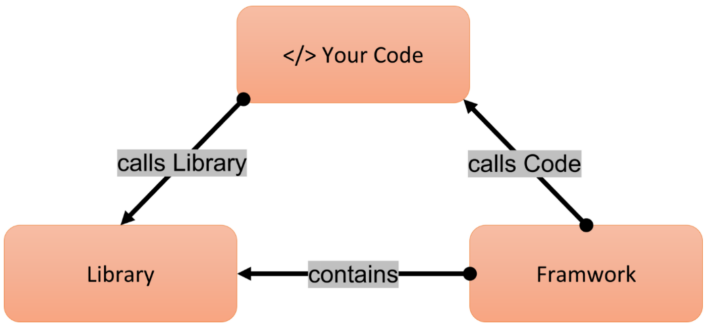
Examples: Angular

Q) Similarities between Library and Framework?

Frameworks and libraries are code written by third parties to solve regular/common problems or to optimise performance.

Q) Difference between Library and Framework?

A key difference between the two is Inversion of control. When using a library, the control remains with the developer who tells the application when to call library functions. When using a framework, the control is reversed, which means that the framework tells the developer where code needs to be provided and calls it as it requires.



* Emmet:

Emmet is the essential toolkit for web-developers. It allows you to type shortcuts that are then expanded into full-fledged

boiler plate code for writing HTML and CSS.

Q) Create Hello World Program using only HTML?

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
  </head>
  <body>
    <div id="root">
      <h1>Hello World! using only HTML</h1>
    </div>
  </body>
</html>
```

Q) Create Hello World Program using only JavaScript?

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

  <body>
    <div id="root"></div>
  </body>
  <script>
    const heading = document.createElement("h1");
    heading.innerHTML = "Hellow World! From Javascript";

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

    divElement.appendChild(heading);
  </script>
</html>
```

Q) Create Hello World Program using only React?

```html
<body>
  <div id="root"></div>

  <script
    crossorigin
    src="https://unpkg.com/react@18/umd/react.development.js"
  ></script>
  <!--This file i.e. react.development.js contains the React Code which is plain JS-->

  <script
    crossorigin
    src="https://unpkg.com/react-dom@18/umd/react-dom.development.js"
  ></script>
  <!--This file i.e. react-dom.development.js is useful for DOM operations or use to modify the DOM-->

  <script>
    const heading = React.createElement("h1", {}, "Hello World from React!");
    const root = ReactDOM.createRoot(document.getElementById("root"));
    root.render(heading);
  </script>
</body>
```

* Cross Origin:

The crossorigin attribute in the script tag enables Cross-Origin Resource Sharing (CORS) for loading external JavaScript files from different origin than the hosting web page. This allows the script to access resources from the server hosting the script, such as making HTTP requests or accessing data.

Q) What is {} denotes in above code?

```
<div id="root">
  <h1 id="title">Hello World!</h1>
</div>
```

This (id='title'), classes, etc should come under {}. Whenever I'm passing inside {}, will go as tag attributes of h1.

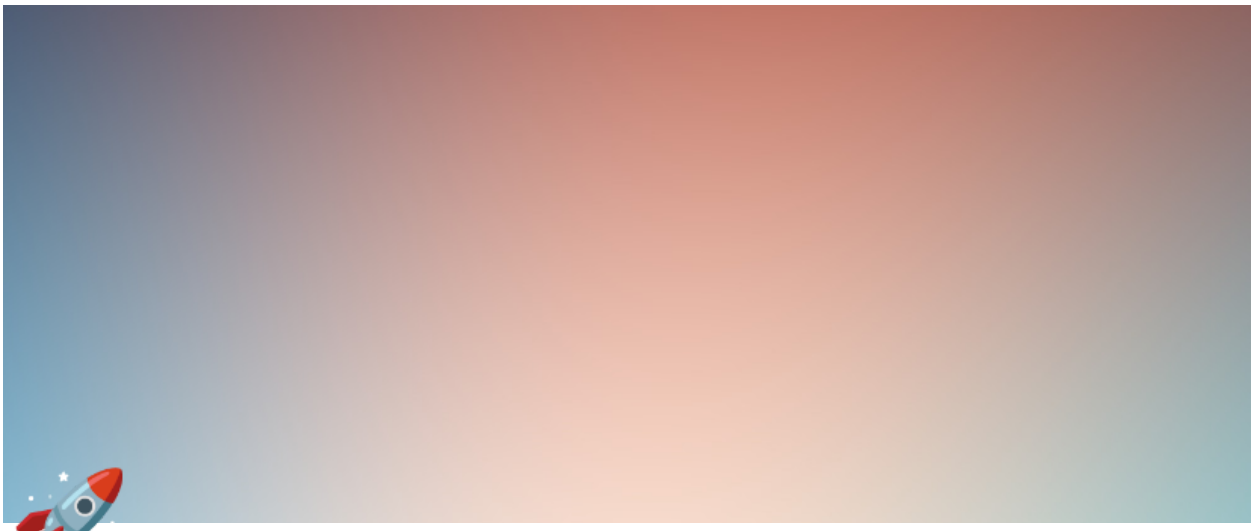
📢 **NOTE: React will overwrite everything inside "root" and replaces with whatever given inside render.**

Q) Do the below HTML code in React?

```
<div id="parent">
  <div id="child1">
    <h1>Heading 1</h1>
    <h2>Heading 2</h2>
  </div>
  <div id="child2">
    <h1>Heading 1</h1>
    <h2>Heading 2</h2>
  </div>
</div>
```

To build the structure like this in React

```
const parent = React.createElement("div", { id: "parent" }, [
  React.createElement("div", { id: "child1" }, [
    React.createElement("h1", { id: "heading_1" }, "Heading 1"),
    React.createElement("h2", { id: "heading_2" }, "Heading 2"),
  ]),
  React.createElement("div", { id: "child2" }, [
    React.createElement("h1", { id: "heading_1" }, "Heading 1"),
    React.createElement("h2", { id: "heading_2" }, "Heading 2"),
  ]),
]);

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

root.render(parent);

```



Igniting Our App! (Namaste-React)

💡 Please make sure to follow along with the whole ***"Namaste React"*** series, starting from Episode-1 and continuing through each subsequent episode. The notes are designed to provide detailed explanations of each concept along with examples to ensure thorough understanding. Each episode builds upon the knowledge gained from the previous ones, so starting from the beginning will give you a comprehensive understanding of React development.

💡 I've got a quick tip for you. To get the most out of these notes, it's a good idea to watch **Episode-2** first. Understanding what ***"Akshay"*** shares in the video will make these notes way easier to understand.

> **So far, here's what we have learned in the previous episode.**
>
> - We studied about Libraries, Frameworks, their differences.
> - We have also created Hello World! using HTML, JavaScript, and React.
> - We have also studies about what is Emmet, CORS (Cross Origin)

Igniting Our App.

Q) To make our app production ready what should we do?

- Minify our file (Remove console logs, bundle things up)
- Need a server to run things

📢 **NOTE: Minify –> Optimization –>Clean console –> Bundle**

* Bundlers:

- A bundler is a tool that bundles our app, packages our app so that it can be shipped to production.
- Examples of Bundlers:

- Webpack
- Vite
- Parcel

> 📢 **NOTE: In create-react-app, the bundler used is webpack.**

* Package Manager:

- Bundlers are packages. If we want to use a package in our code, we have to use a package manager.
- We use a package manager known as npm or yarn

* Configuring the Project:

```
npm init
```

- It creates a package.json file.
- Now to install parcel we will do:

```
npm install -D parcel
```

- Now we will get a package-lock.json file.

* package.json:

- Package.json file is a configuration for NPM. Whatever packages our project needs, we install those packages using npm install <packageName>.

- Once package installation is complete, their versions and configuration related information is stored as dependencies inside package.json file.

* package-lock.json:

- Package-lock.json locks the exact version of packages being used in the project.

Q) What is difference between package.json and package.lock.json?

- In package. json we have information about generic version of installed packages whereas in package.lock.json we have information about the specific or exact version of installed packages.

* node_modules:

- Which gets installed is like a database for the npm.
- Every dependency in node_module will have its package.json.
- Node modules are very heavy so we should always put this in git ignore.

📢 **NOTE: Never touch node_modules and package-lock.json**

* To ignite our app:

```
npx parcel index.html
```

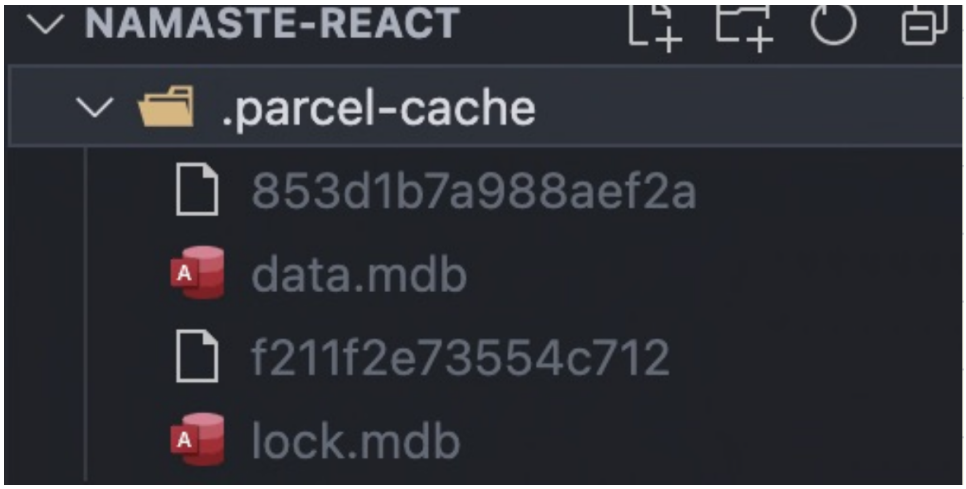
- npx means ‘execute using npm’
- index.html is the entry point

* Hot Module Replacement (HMR):

- It means that parcel will keep a track of all the files which you are updating.
- There is File Watcher Algorithm (written in C++). It keeps track of all the files which are changing realtime and it tells the server to reload.
- These are all done by PARCEL

* parcel-cache:

- Parcel caches code all the time.
- When we run the application, a build is created which takes some time in ms.
- If we make any code changes and save the application, another build will be triggered which might take even less time than the previous build.
- This reduction of time is due to parcel cache.
- Parcel immediately loads the code from the cache every time there is a subsequent build.
- On the very first build parcel creates a folder .parcel-cache where it stores the caches in binary codeformat.
- Parcel gives faster build, faster developer experience because of caching.

- 

* dist:

- It keeps the files minified for us.
- When bundler builds the app, the build goes into a folder called dist.
- The `/dist` folder contains the minimized and optimised version the source code.
- Along with the minified code, the /dist folder also comprises of all the compiled modules that may or may not be used with other systems.
- When we run command:

```
npx parcel index.html
```

- This will create a faster development version of our project and serves it on the server.
- When I tell parcel to make a production build:

```
npx parcel build index.html
```

- It creates a lot of things, minify your file.
- And the parcel will build all the production files to the dist folder.

* Parcel features at a glance:

- Hot Module Replacement (HMR)
- File Watcher Algorithm - C++
- Bundling
- Minify Code
- Cleaning our code
- Dev and production build

- Super fast build algorithm
- Image Optimization
- Caching while development
- Compression
- Compatible with older browser versions
- Https on dev
- Image Optimization
- Port No
- Consistency Hashing Algorithm
- Zero Config
- Tree Shaking

* Transitive Dependencies :

- We have our package manager which takes care of our transitive dependencies of our code.
- If we’ve to build a production ready app which uses all optimisations (like minify, bundling, compression, etc), we need to do all these.
- But we can’t do this alone, we need some dependencies on it. Those dependencies are also dependent on other dependencies.

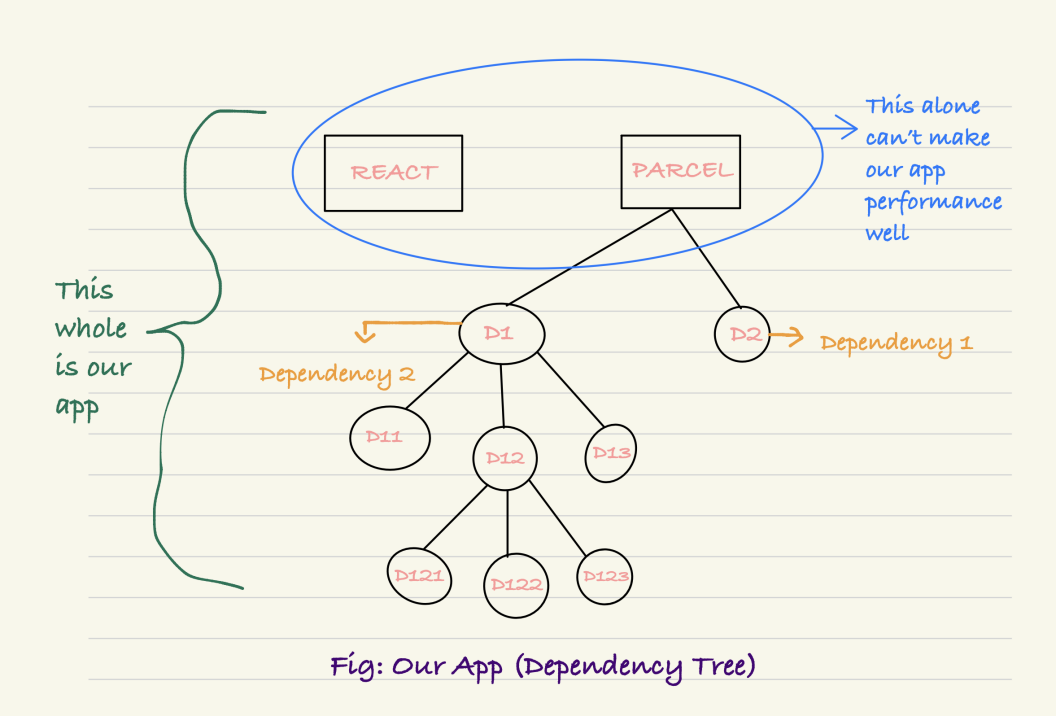
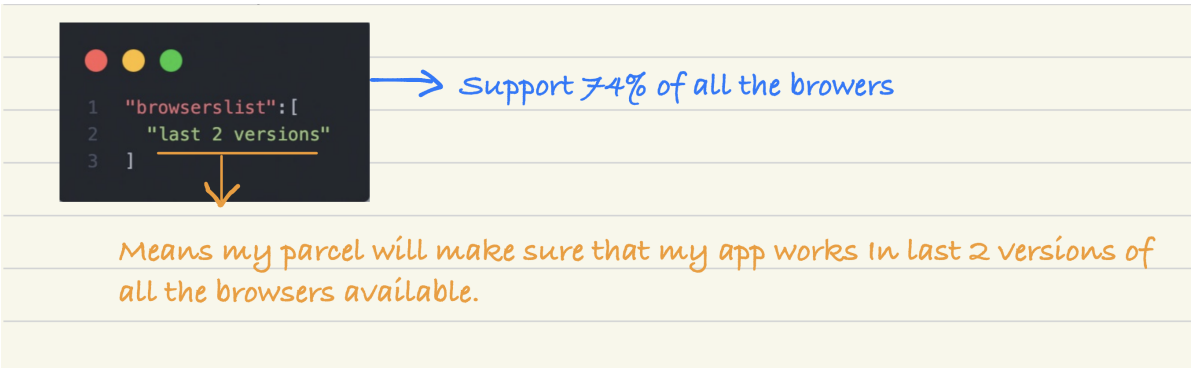


Fig: Our App (Dependency Tree)

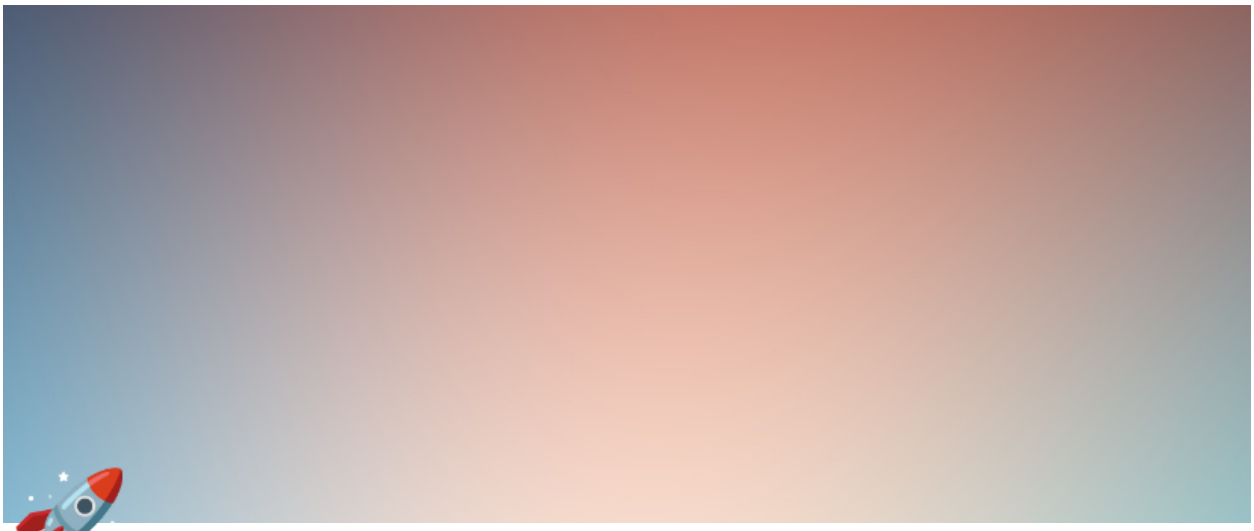
* Browserslist:

- Browserslist is a tool that specifies which browsers should be supported/compatible in your frontend app.
- It makes our code compatible for a lot of browsers.
- In package.json file do:



* Tree Shaking:

- Tree shaking is a process of removing the unwanted code that we do not use while developing the application.
- In computing, tree shaking is a dead code elimination technique that is applied when optimizing code.



Laying the Foundation! (Namaste-React)

💡 Please make sure to follow along with the whole ***"Namaste React"*** series, starting from Episode-1 and continuing through each subsequent episode. The notes are designed to provide detailed explanations of each concept along with examples to ensure thorough understanding. Each episode builds upon the knowledge gained from the previous ones, so starting from the beginning will give you a comprehensive understanding of React development.

💡 I've got a quick tip for you. To get the most out of these notes, it's a good idea to watch **Episode-2** first. Understanding what ***"Akshay"*** shares in the video will make these notes way easier to understand.

> **So far, here's what we've learned in the previous episode**
>
> - We learned that npm is anything but not node package manager and what is npx.
> - We included node-modules and React in our project.
> - We got to know the difference between package.json and package-lock.json.
> - We also explored the concept of bundlers.
> - We learned how to start our app.
> - Don’t forget “Parcel is a Beast”.

Part-1

Q) What is another way of starting the build of the project?

- We will be creating scripts instead of using *“npx parcel index.html”.* We can create different scripts for starting our project in Development and Production.
- In `package.json`, in the script section write the following command.

Fig 3.1

- To run these scripts, enter the following commands in the terminal,

To start:

```
npm run start
```

or

```
npm start
```

For Production Build:

```
npm run build
```

> 💡 If you're not sure how to start the project in a new company then find these scripts in `package.json` and use them.

Part-2

Revision of previous Episodes

Part-3

Introducing JSX.

Before we begin, we have to remove the existing React Code from `App.js` where we used *React.createElement()* for displaying content on the webpage but its syntax is very bad. It’s not developer-friendly, and very hard to read. To solve this problem Facebook developers built JSX.

JSX makes developer life easy as we no longer have to write our code using *React.createElement()*

📢 NOTE: We write code for both Machines and Humans but first for Human understanding as it is read by a lot of developers

Q) What is JSX?

JSX is HTML-like or XML-like syntax. JSX stands for JavaScript XML. It's a syntax extension for JavaScript.

- It is not a part of React. React apps can be built even without JSX but the code will become very hard to read.
- It is not HTML inside JavaScript.
- JavaScript engine cannot understand JSX as it only understands ECMAScript

```
// React.createElement => Object => HTMLElement(render)

// Using Pure React
const heading = React.createElement(
    "h1",
    { id: "heading" },
    "Namaste React"
);

// Using JSX
const jsxHeading = <h1>Namaste React using JSX</h1>
```

```
// React.createElement => Object => HTMLElement(render)

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

// JSX
const jsxHeading = <h1>Namaste React using JSX</h1>
```

When we log *heading* and *jsxHeading*, it gives the same object.

From this point, we will not be using *React.createElement()*

Introducing Babel

Q) Is JSX a valid JavaScript?

The answer is yes and no.

- JSX is not a valid Javascript syntax as it's not pure HTML or pure JavaScript for a browser to understand. JS does not have built-in JSX. The JS engine does not understand JSX because the JS engine understands ECMAScript or ES6+ code

Q) If the browser can't understand JSX how is it still working?

This is because of Parcel because *"Parcel is a Beast".*

Before the code gets to JS Engine it is sent to Parcel and Transpiled there. Then after transpilation, the browser gets the code that it can understand.

Transpilation ⇒ Converting the code in such a format that the browsers can understand.

Parcel is like a manager who gives the responsibility of transpilation to a package called Babel.

Babel is a package that is a compiler/transpiler of JavaScript that is already present inside 'node-modules'. It takes JSX and converts it into the code that browsers understand, as soon as we write it and save the file. It is not created by Facebook. Learn more about Babel on babeljs.io

JSX (transpiled by Babel) ⇒ React.createElement ⇒ ReactElement ⇒ JS Object ⇒ HTML Element(render)

Q) What is the difference between HTML and JSX?

JSX is not HTML. It's HTML-like syntax.

- HTML uses ‘class’ property whereas JSX uses ‘className’ property
- HTML can use hypens in property names whereas JSX uses camelCase syntax.

Single Line and Multi Line JSX Code

Single line code:

```
const jsxHeading = <h1>Namaste React</h1>
```

Multi-line code:

If writing JSX in multiple lines then using ‘()’ parenthesis is mandatory. To tell Babel from where JSX is starting and ending.

```
const jsxHeading = (
  <div>
   <h1>Namaste React</h1>
  </div>
)
```

📢 NOTE:

1) Use “Prettier - Code Formatter” VS Code Extension to make your code look beautiful with proper formatting
2) Use “ES lint” VS Code Extension for linting
3) Use “Better Comments” VS Code Extension to beautify your comments

Code all of these things discussed until now for better understanding.

Part-4

Introducing React Components

Everything inside React is a component.

Q) What are Components?

There are 2 types of components:

1.Class-based Components - Old way of writing code, used rarely in industry

2.Functional Components - New way of writing code, most commonly used

Q) What is a React Functional Components?

It is just a JavaScript Function that returns some JSX or a react element.

Always name React Functional Component with Capital Letters otherwise you will confuse it with normal function

```
// All are the same for single-line code
const HeadingComponent1 = () => (
  <h1>Namaste</h1>
)
```

```
const HeadingComponent2 = () => {
 return <h1>Namaste</h1>
}

const HeadingComponent3 = () => <h1>Namaste</h1>
```

To render a functional component we call them '<Heading1 />'. This is the syntax that Babel understands.

You can also call them using these ways,

'<Title></Title>'

or

'{Title()}'

Components Composition

A component inside a component.

Calling a component inside another component is Component Composition.

```
const Title = () => <h1>Namaste React</h1>

const HeadingComponent = () => (
 <div id="container">
  <Title />
 </div>
)
```

Code inside the 'Title' component will be used inside the 'HeadingComponent' component as the 'Title' component is called inside it. It will become something like this,

```
const HeadingComponent = () => (
 <div id="container">
   <h1>Namaste React</h1>
 </div>
)
```

Part-5

Q) How to use JavaScript code inside JSX?

Inside a React Component when ‘{}’ parenthesis is present we can write any JavaScript expression inside it.

```
const number = 10000;

const HeadingComponent = () => (
 <div id="containter">
   {number}
   <h1>Namaste React</h1>
 </div>
)
```

Q) How to call React Element in JSX?

We can use ‘{}’ parenthesis.

```
const elem = <span> React Element </span>

const HeadingComponent = () => (
 <div id="containter">
```

```
    {elem}
    <h1>This is Namaste React</h1>
  </div>
 )
```

Q) What will happen if we call 2 elements inside each other?

If we put 2 components inside each other, then it will go into an infinite loop and the stack will overflow. It will freeze your browser, so it’s not recommended to do so.

Advantages of using JSX.

1) Sanitizes the data

If someone gets access to your JS code and sends some malicious data which will then get displayed on the screen, that attack is called cross-site scripting.

It can read cookies, local storage, session storage, get cookies, get info about your device, and read data. JSx takes care of your data.

If some API passes some malicious data JSX will escape it. It prevents cross-site scripting and sanitizes the data before rendering

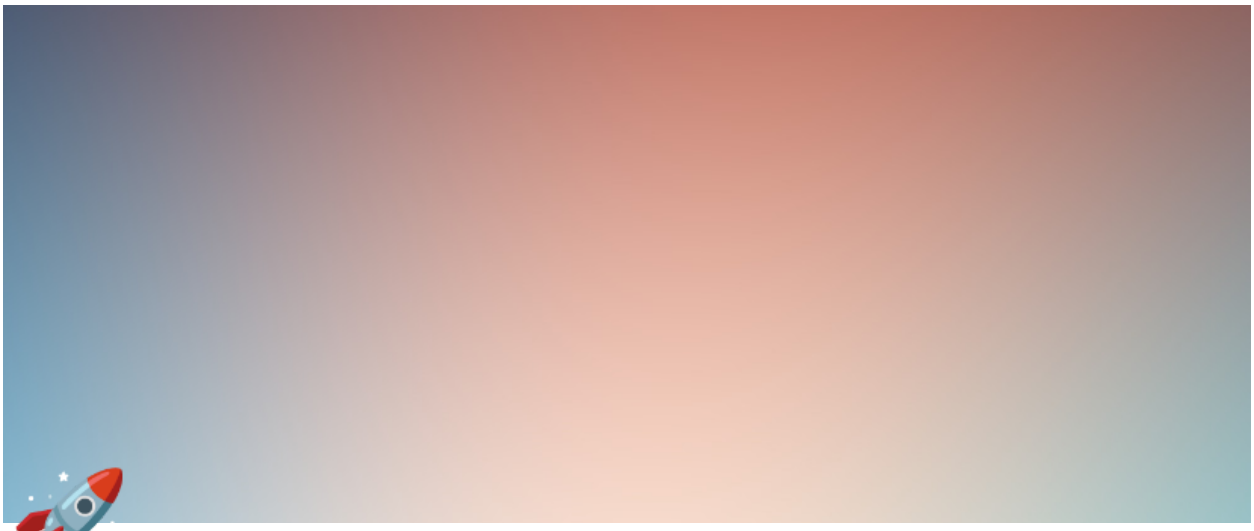
2) Makes code readable

JSX makes it easier to write code as we are no longer creating elements using React.createElement()

3) Makes code simple and elegant

4) Show more useful errors and warnings

5) JSX prevents code injections (attacks)



Talk is Cheap, Show me the Code! (Namaste-React)

💡 Please make sure to follow along with the whole ***"Namaste React"*** series, starting from Episode-1 and continuing through each subsequent episode. The notes are designed to provide detailed explanations of each concept along with examples to ensure thorough understanding. Each episode builds upon the knowledge gained from the previous ones, so starting from the beginning will give you a comprehensive understanding of React development.

💡 I've got a quick tip for you. To get the most out of these notes, it's a good idea to watch **Episode-3** first. Understanding what ***"Akshay"*** shares in the video will make these notes way easier to understand.

> **So far, here's what we've learned in the previous episode**
>
> - We learned what’s JSX.
> - We explored what is transpilation and Babel.
> - We got to know the difference between Class Based Components and Functional Components.
> - We also explored the concept of bundlers.
> - We learned what is component composition.

Part-1

In this episode, we will start actual coding by starting a new project. Our app is going to a Food Ordering App.

Planning for the UI

Before we start coding, plan things out. Planning will make things easier to understand. We should know exactly what to build:

- Name the App
- UI Structure

```
Header
 - Logo
 - Nav Items
```

```
Body
 - Search
 - Restaurant Container
   - Restaurant Card
     - Dish Name
     - Image
     - Restaurant Name
     - Rating
     - Cuisines
     - Time to Deliver

Footer
 - Copyright
 - Links
 - Address
 - Contact
```

- Keep that as a reference and start coding the app.

Let’s start coding!

It is recommended that you code on your own but for some examples, we have mentioned some pieces for you along with the component name.

Main components = AppLayout

```
const AppLayout = () => {
 return (
  <div className="app">
   <Header/>
   <Body/>
  </div>
```

```
 )
}
```

Header Component

```
const Header = () => {
 return(
  <div className="header">
    <div className="logo-container">
      <img className="logo" src="url" />
    </div>
    <div className="nav-items">
      <ul>
        <li>Home</li>
        <li>About Us</li>
        <li>Contact Us</li>
        <li>Cart</li>
      </ul>
    </div>
  </div>
 )
 }
```

Inline Styling

Writing the CSS along with the element in the same file. It is not recommended to use inline styling. So you should avoid writing it.

```
<div
className="red-card"
style={{ backgroundColor: "#f0f0f0" }}
>
```

```
 <h3> Meghana Foods </h3>
</div>
```

In '*style={{ backgroundColor: "#f0f0f0" }}*', first bracket is to tell that whatever is coming next will be JavaScript and the second bracket is for JavaScript object

or you can store the CSS in a variable and then use it

```
const styleCard = { backgroundColor: "#f0f0f0" };

<div
className="red-card"
style={styleCard}
>
 <h3> Meghana Foods </h3>
</div>
```

Part-2

Introducing Props.

Short form for properties. To dynamically send data to a component we use props. Passing a prop to a function is like passing an argument to a function.

Passing Props to a Component

Example,

```
<RestaurantCard
  resName="Meghana Foods"
  cuisine="Biryani, North Indian"
/>
```

‘resName’ and *‘cuisine’* a props and this is prop passing to a component.

Receiving props in the Component

Props will be wrapped and send in Javascript object

Example,

```
const RestaurantCard = (props) => {
  return(
    <div>{props.resName}</div>
  )
}
```

Destructuring Props

Example,

```
const RestaurantCard = ({resName, cuisine}) => {
  return(
    <div>{resName}</div>
  )
}
```

Config Driven UI.

It is a user Interface that is built and configured using a declaration configuration file or data structure, rather than being hardcoded.

Config is the data coming from the api which keeps on changing according to different factors like user, location, etc.

To add something in the elements of array

Example, Adding “,” after every value

```
resData.data.cuisine.join(", ")
```

Good Practices

Destructuring props-
Optional Chaining
Example,

```
const {name, avgRating, cuisine} = resData?.data;
```

Repeating ourselves (repeating a piece of code again and again)-
Dynamic Component listing using JS map() function to loop over an array and pass the data to component once instead of hard coding the same component with different props values

Avoid ❌

```
<RestaurantCard
  resName="Meghana Foods"
/>
<RestaurantCard
  resName="KFC"
/>
<RestaurantCard
  resName="McDonald's"
/>
<RestaurantCard
  resName="Dominos"
/>
```

Follow ✅

```
const resList = [
 {
  resName: "Meghana Foods"
 },
 {
  resName: "KFC"
 },
 {
  resName: "McDonald's"
 },
 {
  resName: "Dominos"
 }
]

const Body = () => {
 return(
  <div>
   {resList.map((restaurant) => (
```

```
        <RestaurantCard resData={restaurant} />
      ))}
    </div>
  )
}
```

Unique Key id while using map-

Each item in the list must be uniquely identified

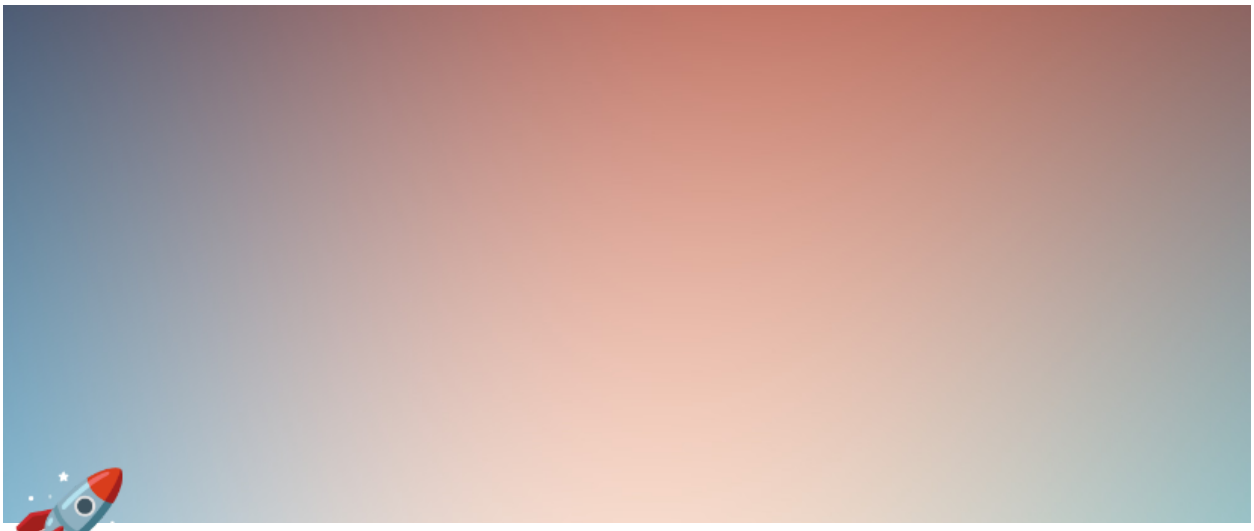
Why?

When we have components at same level and if a new component comes on the first without ID, DOM is going to re-render all the components again. As DOM can’t identify where to place it.

But if we give each of them a unique ID then react knows where to put that component according to the ID. It is a good optimization and performance thing.

Note* Never use index as keys in map. It is not recommended.

```
const Body = () => {
 return(
  <div>
   {resList.map((restaurant) => (
      <RestaurantCard key={restaurant.id} resData={restaurant} /
    ))}
  </div>
 )
}
```



Let’s Get Hooked! (Namaste-React)

💡 Please make sure to follow along with the whole ***”Namaste React"*** series, starting from Episode-1 and continuing through each subsequent episode. The notes are designed to provide detailed explanations of each concept along with examples to ensure thorough understanding. Each episode builds upon the knowledge gained from the previous ones, so starting from the beginning will give you a comprehensive understanding of React development.

💡 I've got a quick tip for you. To get the most out of these notes, it's a good idea to watch **Episode-5** first. Understanding what ***”Akshay”*** shares in the video will make these notes way easier to understand.

> **So far, here's what we've learned in the previous episode**
>
> - We built a food ordering app that displayed restaurant cards using real-time data from an API (like Swiggy's).
> - We also explored the concept of config-driven UI.

Before we start to learn today's episode, a common question many of us may encounter is:

Q) Why do we use React? Some of us might wonder why we don't just stick to HTML, CSS, and JAVASCRIPT for everything we've been doing?

> 👉🏽 Of course! It's absolutely possible to accomplish everything using regular ***HTML, CSS*** and ***JAVASCRIPT*** without using ***REACT***. However, we chose React because it enhances our developer experience, making it more seamless and efficient.

Part-1

Introducing React-Hooks.

Before we begin, the first thing we need to do is clean up our app. Up until now, we've placed all our components inside a single `App.js` file, but this isn't considered good practice. It's best to create separate files for each component.

Q) How can we achieve this?

To achieve this, Let's discuss the folder structure. Currently, all our files are located at the root level of our project.
(
If you're following along with the course, you can view the current structure of your project on the left side of your code editor)

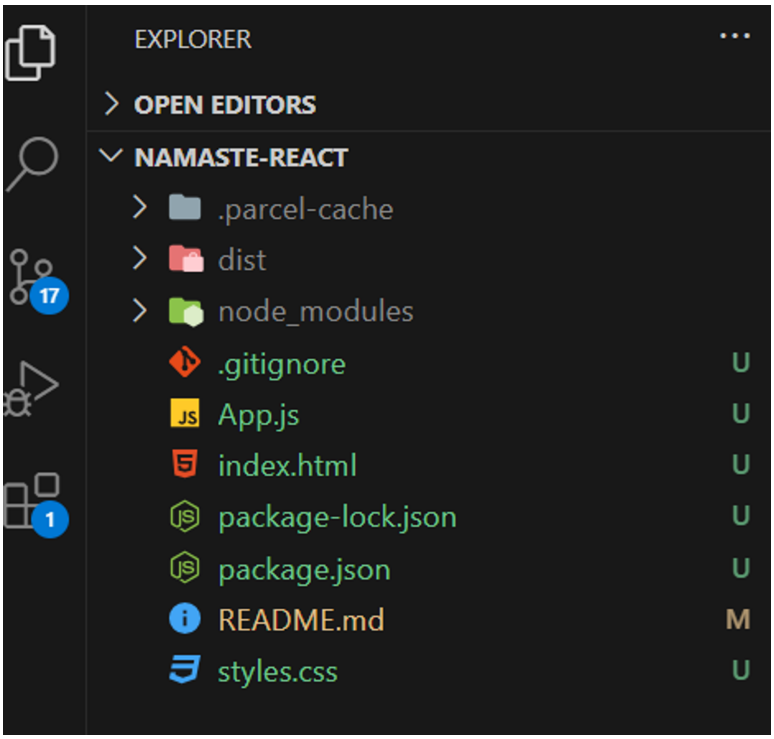


Fig 6.1

We are going to Restructure our project folder **"NAMASTE-REACT"**

- there is a very good convention in the industry that all the code in a React project is kept in a '**src-**folder', there is no compulsion to use a '**src-***folder*' in a project. But here we are following what the industry follows.
- We are creating and moving our `App.js` file in the '**src-***folder*' and whatever new files we create we put them in the '**src-***folder*'.

> 📢 NOTE: Don't forget to update the path of the `App.js` file in `index.html` other we will get an Error.

The best practice is to make separate files for every component.

We have the following components.

```
1. Header
2. RestaurantCard
3. Body
```

We put all the above components inside the folder named '**components**'(child-folder) which has been placed inside the '**src-** folder'(parent folder). When we are creating separate component files inside the '**components**-folder' always start with a capital letter like.

```
In this course, we are using (.js) as an extension.
1. Header.js
2. RestaurantCard.js
3. Body.js

NOTE: We can use (.jsx) as an extension instead of (.js) it's
```

```
up to the developer's wish.
 1. Header.jsx
 2. RestaurantCard.jsx
 3. Body.jsx
```

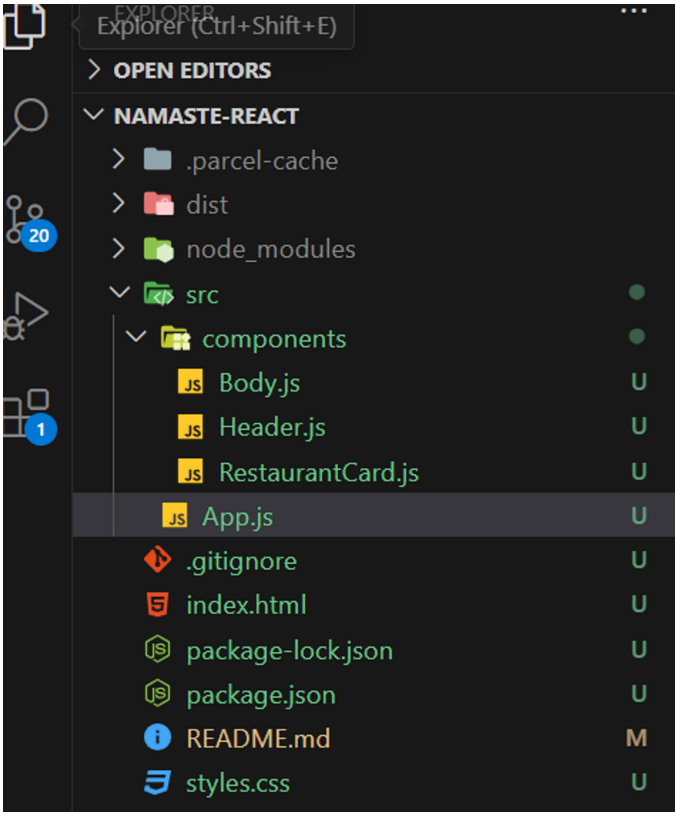


Fig 6.2

📢 NOTE: NO Thumb Rule to use this convention, we could use any folder structure you wish.

Understanding Export and Import in React.

Two types of export/import in React, We will understand each of them in details.

1.Default export/import.
2.Named export/import.

As we know we move each component's code and create new files individually.
Still, Our React project throws an error because our `App.js` file doesn't have components in it and we are using components inside the `App.js`
to solve this we need to import the components from their respective files which have been kept inside the ***components-folder*** in *src*.(refer fig 6.2)

to understand it well let's take an example of the `Header.js` component.

Example:
The
`Header.js` component is missing in `App.js` so we are not able to use it, if we try to use it, It throws an error, To solve this, here we have to import `Header.js` inside the `App.js` and before the import, we have to 1st initially export `Header.js` component.

1.Default export/import. 👇🏽

Step-1 (export)————————>
We use the ‘export’ and ‘default’ keywords with the component name at the end of the component file.
In (figure 6.3) on line-no-19 we see that we are exporting Header componet. Understand the Syntax properly.

```
// Syntax
export default Header;
```

```
// Or we can write with extension.
export default Header.js;
```

```
3  //Header
4  const navItems = (···
10 );
11
12 const Header = () => (···
17 );
18
19 export default Header;
```

Fig 6.3

Step-2 (import)————————>

In (fig 6.4), we import the Header on line-no-3, inside `App.js` We use ‘import’ with the component name at the start of the file. Understand the Syntax properly.

```
// Syntax
import Header from "./components/Header"
```

```
1  import React from "react";
2  import ReactDOM from "react-dom/client";
3  import Header from "./components/Header";
4
5
6
7
8
```

Fig 6.4

📢 NOTE:
1. If you are using Vs-Code Editor during *import* it automatically tracks the path of the component and gives suggestions to us. so we don't have to worry about the path.

2. We don't have to put an extension in the file in the import statement. If want to put then completely Fine.

Follow the same method for the rest of the components.

In the case of RestaurantCard, we are using it inside the body component.
follow the same steps

Step-1 (export)————————> exporting `RestaurantCard.js`

Step-2 (import)————————> Importing `RestaurantCard.js` inside the `Body.js`

If we attempt to run our project, we'll encounter the 'resLists not defined' error.

This happens because '`resList`' is being used inside '`Body.js`' without being defined in that file. While one solution is moving '`resLists`' inside '`Body.js`', it's not considered the best practice.

📢 NOTE:

`resList` contains hard-coded data and we never put any hard-coded data like the *source-URL* of *logo* and *images* inside the component file. That's not the best practice the industry follows.

Q) So where should we keep it?

- We've established a new directory within the '**src**' folder called '**utils**', and inside it, we've created a file named '`constants.js`' to store all hard coded data. You can choose any name for this file, but we've opted for lowercase letters since it's not a component. Additionally, we've included the source of logos and images in this file.
- We've stored our mock data for '`resList`' in a file named '`mockData.js`', which resides within the '**utils**' folder.
- We need to export data from '`constants.js`' and '`mockData.js`', and then import it into the necessary component files where it will be used

---

But there's a catch: If we intend to export multiple items simultaneously, from single file ‘default export/import’ won't work; it'll result in an error (refer Fig 6.5). For instance, in our '`constants.js`' file, we've stored `URLs` for logos and images using separate variables, which means default export/import won't be feasible (refer Fig 6.5).

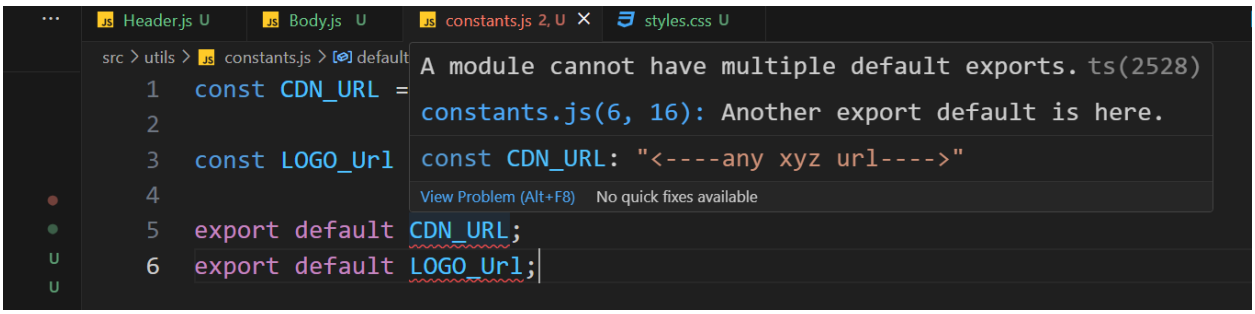


Fig 6.5

Instead, we can employ ‘named export/import’ to handle this scenario effectively.

2.Named export/import. 👇🏽

just write the ‘`export`’ keyword before the variables we want to export (refer Fig 6.6). We won’t get any error.

Step-1 (export)————————>

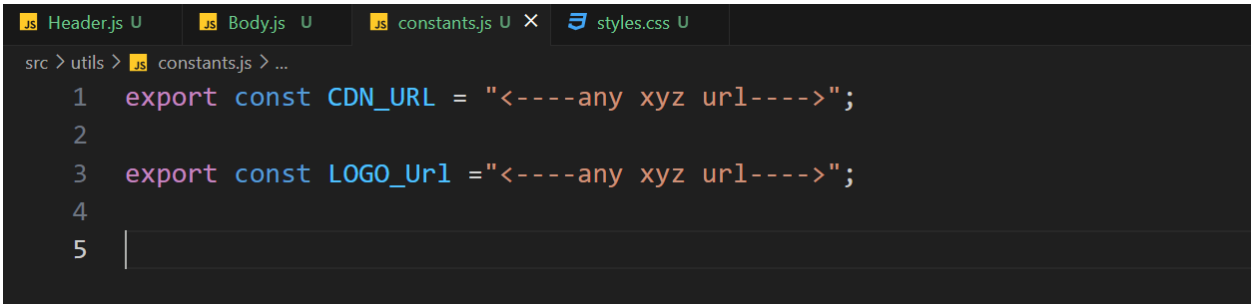


Fig 6.6

Step-2 (import)————————>

when we import there is a slight difference in syntax we use curly braces.

- We import LOGO_URL inside the `Header.js` and
- Import the CDN_URL inside the `RestaurantCard.js`

```
import { LOGO_URL } from "../utils/constants";
```

```
import { CDN_URL } from "../utils/constants";
```

Q) Can we use default export with named export ?

👉🏽 yes

Part-2

Let's make our app more lively and engaging.

In this tutorial series, we're learning by actually building things. As we go along, we'll keep adding new features to our app.

For now, we're going to add a button. When you click on this button, it will show you the best-rated restaurants.

Inside the body component, instead of having a search box, let's replace it with a '`<button>`'. We'll add this button inside a '`<div>`' and give it any class name we want. Similarly, we'll also give a class name to the button, as per our preference.

We are adding `onclick` evenhandler inside the button, `onclick` has call-back function which will be called when we clicked on the button, there is condition when we clicked on the restaurant we will get a restaurant which avg-rating is more than 4.2 (avg-rating > 4.2)

Q) How these cards are coming on the screen?

- These cards are appear on the screen because we're using the `map()` method to go through each restaurant in our mock data (`resList`). This method helps us display the information from each restaurant on individual cards. so any thing changes in reslist the cards displayed on the screen will also change.

- We're simplifying our mock data to improve our understanding, or you can use the data provided below (which is the actual data from the Swiggy API).

```
let listOfRestaurant = [
  {
    data: {
      id: "255655",
      name: "Cake & Cream",
      cloudinaryImageId: "ac57cc371e73f96f812613f58457aca3",
      areaName: "Jairaj Nagar",
      costForTwo: "₹200 for two",
      cuisines: ["Bakery", "Hot-dog", "pastery", "Cake", "Thi
ck- shake"],
      avgRating: 4.3,
      veg: true,
      parentId: "54670",
      avgRatingString: "4",
      totalRatingsString: "20+",
    },
  },
  {
    data: {
      id: "350363",
      name: "Haldiram's Sweets and Namkeen",
      cloudinaryImageId: "25c3a7d394d6c5556b134385f7d665b0",
      avgRating: 4.6,
      veg: true,
      cuisines: [
        "North Indian",
        "South Indian",
        "Chinese",
        "Pizzas",
        "Fast Food",
      ],
```

```
      parentId: "391465",
      avgRatingString: "4.6",
      totalRatingsString: "100+",
    },
  },
  {
    data: {
      id: "154891",
      name: "Rasraj Restaurant",
      cloudinaryImageId: "egbr63ulc8h1zgliivd8",
      locality: "Civil Line",
      areaName: "Civil Lines",
      costForTwo: "₹250 for two",
      cuisines: [
        "North Indian",
        "South Indian",
        "Street Food",
        "Chinese",
        "Pizzas",
        "Fast Food",
      ],
      avgRating: 4.2,
    },
  },
  {
    data: {
      id: "745961",
      name: "Balaji Restaurant",
      cloudinaryImageId: "b8672fe52944c3599ea324d99d608300",
      locality: "Sai Rubber Stamp",
      areaName: "Jairaj Nagar",
      costForTwo: "₹149 for two",
      cuisines: ["South Indian", "North Indian"],
      avgRating: 4.8,
      veg: true,
    },
```

```
  },
  {
    data: {
      id: "798745",
      name: "Friends Restaurant",
      cloudinaryImageId: "b14cd9fc40129fcfb97aa7e621719d07",
      locality: "Gayatri Nagar",
      areaName: "Jairaj Nagar",
      costForTwo: "₹150 for two",
      cuisines: ["North Indian", "Chinese", "Biryani", "Tando
or", "Kebabs"],
      avgRating: 4.2,
      parentId: "84308",
    },
  },
  {
    data: {
      id: "314737",
      name: "RASOI the KITCHEN",
      cloudinaryImageId: "yjymo9nhyn7rhvafsrd3",
      locality: "Sriram Chowk",
      areaName: "Bazar Ward",
      costForTwo: "₹200 for two",
      cuisines: ["North Indian", "Maharashtrian", "Chinese",
"Thalis"],
      avgRating: 3.9,
      parentId: "167341",
    },
  },
  {
    data: {
      id: "201454",
      name: "Morsels restaurants",
      cloudinaryImageId: "aafe71251ef5328784652dc838cd91f3",
      locality: "Bazar Ward",
      areaName: "Chandrapur Locality",
```

```
      costForTwo: "₹300 for two",
      cuisines: ["North Indian", "South Indian"],
      avgRating: 3.2,
      veg: true,
      parentId: "139266",
      avgRatingString: "4.2",
    },
  },
  {
    data: {
      id: "266124",
      name: "Trimurti Restaurant",
      cloudinaryImageId: "8135c0066b06e2925c66930be4e9ffb5",
      locality: "Bazar Ward",
      areaName: "Chandrapur Locality",
      costForTwo: "₹150 for two",
      cuisines: ["Desserts"],
      avgRating: 3.9,
      veg: true,
      parentId: "217751",
      avgRatingString: "4.4",
    },
  },
  {
    data: {
      id: "509254",
      name: "Saha Restaurant",
      cloudinaryImageId: "z1ez4uc9idul2uj2v87g",
      areaName: "Jairaj Nagar",
      costForTwo: "₹300 for two",
      cuisines: ["North Indian", "Biryani", "Thalis", "Bevera
ges"],
      avgRating: 3.7,
      parentId: "174585",
      avgRatingString: "3.7",
    },
```

```
  },
];
```

We utilize the data mentioned above, incorporating a condition within the callback function. This condition triggers when we click on the button, displaying only the restaurants whose average rating is above 4.2 . That’s what we are expecting.

```
<button
    onClick={() => {
        const filterLogic = listOfRestaurant.filter((res)
        => {
            return res.info.avgRating > 4.2;
        });
        console.log(filterLogic);
    }}
>
    Top Restaurant
</button>
```

Fig 6.7

Despite implementing the code in Figure 6.7, no visible changes occur on the screen. However, we successfully filter the data, which we can confirm by checking the filtered results using `console.log(filterlogic).`

📢 Note:
whenever we have react App we have a UI layer and data layer, UI layer will display what is being sent by the data layer.

Q) How can we display filtered restaurants dynamically on UI(display screen) ?

👉 Here, we're utilizing data retrieved from the `listOfRestaurant` variable, which stores an array of objects. It's treated as a regular variable within our codebase. However, for this functionality, we require a superpowerful React variable known as a ‘state variable’.

Q) How do we create Super-powerful variable ?

👉 for that we use ‘React Hooks’.

Q) What is Hook ?

👉 It's simply a regular JavaScript function. However, it becomes powerful when used within React, as it's provided to us by React itself. These pre-built functions have underlying logic developed by React developers. When we install React via npm, we gain access to these superpowers.

Two crucial hooks we frequently utilize are:

1. useState()

2. useEffect()

1. useState()

(import)————————>

- First, we have to import as a named import from ‘react’.
- We are using `useState()` inside the body component to create a ‘state variable‘. Look at the syntax below.

```
import { useState } from "react";
```

```
// syntax of useState()
const [listOfRestaurant] = useState([]);
```

In the provided code, we pass an empty array `[]` as the initial value inside the `useState([])` method. This empty array serves as the default value for the `listOfRestaurant` variable.

If we pass the `listOfRestaurant` were we stored all the restaurant data inside the `useState()` as the default data, it will render the restaurants on the screen using that initial data.

Q) How could we modify the list of restaurant ?

To modify we pass the second argument `setListOfRestaurant` we can name as we wish. `setListOfRestaurant` used to update the list.

```
import { useState } from "react";
// syntax of useState()
const [listOfRestaurant , setListOfRestaurant] = useState
([]);
```

Q) How can we display filtered restaurants dynamically on UI(display screen)? I am repeating the same question which we had previously.

We are using `setListOfRestaurant` inside the call-back function to show the filtered restaurant. on clicking the button we will see the Updated top-rated restaurant.

```
<button
  className="filter-btn"
  onClick={() => {
    const filtertheRestaurant = listOfRestaurant.filter((res)
=> {
      return ( res.data.avgRating > 4));
    });
```

```
    setListOfRestaurant(filtertheRestaurant);
  }}
>
  Top Rated Restaurant
</button>;
```

📢 NOTE:
- The crucial point about State variables is that whenever they update, React triggers a reconciliation cycle and re-renders the component.
- This means that as soon as the data layer changes, React promptly updates the UI layer. The data layer is always kept in sync with the UI layer.
- To achieve this rapid operation, React employs a reconciliation algorithm, also known as the diffing algorithm or React-Fibre which we will delve into further below.

> **React is often praised for its speed, have you ever wondered why? 🤔**

At the core lies React-Fiber - a powerhouse reimplementation of React's algorithm. The goal of React Fiber is to increase its suitability for areas like animation, layout, and gestures. Its headline feature is incremental rendering: the ability to split rendering work into chunks and spread it out over multiple frames.

These days, we can use ***JavaScript*** and ***React*** alongside popular libraries like ***GSAP*** (GreenSock Animation Platform) and ***Three.js***.

These tools allow us to create animations and 3D designs using the capabilities of ***JavaScript*** and ***React***.

But how does it all work behind the scenes?

👉🏽 When you create elements in React, you're actually creating ***virtual DOM objects***. These virtual replicas are synced with the **real DOM**, a process known as "Reconciliation" or the React "diffing" algorithm.

Essentially, every rendering cycle compares the new UI blueprint (updated VDOM) with the old one (previous VDOM) and makes precise changes to the actual DOM accordingly.

It's important to understand these fundamentals in order to unlock a world of possibilities for front-end developers!

Do you want to understand and dive deep into it?

👉🏽 Take a look at this awesome React Fiber architecture repository on the web: https://github.com/acdlite/react-fiber-architecture

Exploring The World!(Namaste-React)

💡 I've got a quick tip for you. To get the most out of these notes, it's a good idea to watch **Episode-6** first. Understanding what ***"Akshay"*** shares in the video will make these notes way easier to understand.

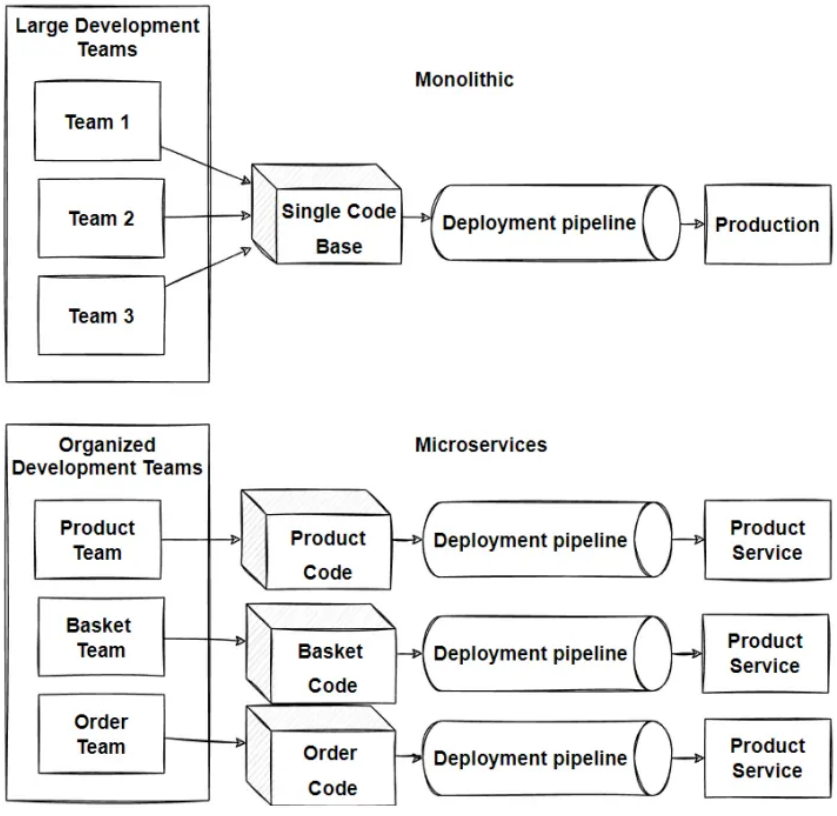
> Here are two quick stories:
>
> ⬇️ In the web development world, we often hear that the trend is leaning from ***'Monolithic'*** towards the ***'Microservices'*** architecture.
>
> 1️⃣ **Atlassian's Shift**: Back in 2018, '**Atlassian**' faced some growing pains. To keep up with demand and stay flexible, they switched to microservices. This move helped them stay agile and scale up smoothly.

> 2️⃣ **Netflix's Transformation :** The popular video streaming platform over in the world '**Netflix**' runs on AWS. They started with a monolith and moved to microservices.
>
> 📢 In today's episode, we discussed the trend towards lighter and more adaptable web architectures.

PART-1

Q) What are 'Monolithic' and 'Microservices' architectures exactly?

👉🏽 Understanding '**Monolith**' and '**Microservices**' architectures is a big deal in software development, but as developers, it's important to grasp the basics. So, in this episode, we'll break it down into simple terms.



Monolithic Architecture

In the past, we used to build large projects where everything was bundled together. Imagine building an entire application where all the code—APIs, user interface, database connections, authentication, even notification services—resides in one massive project with single code base.

- **Size and Complexity Limitation**: Monolithic applications become too large and complex to understand.
- **Slow Startup**: The application's size can slow down startup time.
- **Full Deployment Required**: Every update requires redeploying the entire application.
- **Limited Change Understanding**: It's hard to grasp the full impact of changes, leading to extensive manual testing.
- **Difficult Continuous Deployment**: Implementing continuous deployment is challenging.
- **Scaling Challenges**: Different modules may have conflicting resource needs, making scaling difficulty.
- **Reliability Concerns**: Bugs in any module can crash the whole application, affecting availability.
- **Adoption of New Technologies**: Making changes in frameworks or languages is expensive and time-consuming since it affects the entire application.

Microservices Architecture

The idea is to split your application into a set of smaller, interconnected services instead of building a single monolithic application. Each service handles a specific job, like handling user accounts or managing payments. Inside each service, there's a mini-world of its own, with its own set of rules (business logic) and tools (adapters). Some services talk to each other in different ways, like using REST or messaging. Others might even have their own website!

- **Simpler Development**: Microservices break down complex applications into smaller, easier-to-handle services. This makes development faster and maintenance easier.
- **Independent Teams**: Each service can be developed independently by a team focused on that specific task.
- **Flexibility in Technology**: Developers have the freedom to choose the best technologies for each service, without being tied to choices made at the project's start.
- **Continuous Deployment**: Microservices allow for independent deployment, enabling continuous deployment for complex applications.
- **Scalability**: Each service can be scaled independently, ensuring efficient resource usage.

- **Separation of Concerns**: With each task having its own project, the architecture stays organized and manageable.
- **Single Responsibility**: Every service has its own job, following the principle of single responsibility. This ensures focused and efficient development.

Q) Why Microservices?

👉🏽 Breaking things down into microservices helps us work faster and smarter. We can update or replace each piece without causing a fuss. It's like having a well-oiled machine where each part does its job perfectly.

Q) How do these services interact with each other?

👉🏽 In our setup, the UI microservice is written in React, which handles the user interface.

Communication Channels

These services interact with each other through various communication channels. For instance, the UI microservice might need data from the backend microservice, which in turn might need to access the database.

Ports and Domain Mapping

Each microservice runs on its specific port. This means that different services can be deployed independently, with each one assigned to a different port. All these ports are then mapped to a domain name, providing a unified access point for the entire application.

PART-2

Connecting to the External World

In this episode, we're going to explore how our React application communicates with the outside world. We'll dive into how our application fetches data and seamlessly integrates it into the user interface. It's all about understanding data exchange that makes our app come alive.

In our Body component, we're displaying a list of restaurants. Initially, we used mock data inside the '`useState()`' hook to create a state variable. However, in this episode, we're stepping up our game by fetching real-time data from Swiggy's API and displaying it dynamically on the screen. How cool is that? 😍

Before diving in, let's understand two approaches to fetch and render the data :

1. Load and Render:

We can make the API call as soon as the app loads, fetch the data, and render it.

2. Render First Fetch Later:

Alternatively, we can quickly render the UI when the page loads we could show the structure of the web page, and then make the API call. Once we get the data, we

re-render the application to display the updated information.

In React, we're opting for the second approach. This approach enhances user experience by rendering the UI swiftly and then seamlessly updating it once we receive the data from the API call.

PART-3

Today, we're diving into another important topic ‘`useEffect()`’ We've mentioned it before in a previous episode. Essentially, ‘`useEffect()`’ is a *Hook React* provides us, it is a regular JavaScript function, to help manage our components.

To start exploring its purpose, let's first import it from React.

```
import { useEffect } from "react";
```

‘`useEffect()`' takes two arguments .

1. Callback function.
2. Dependency Array.

```
// Syntax of useEffect()
// We passed Arrow function as callback function.

useEffect(() => {}, []);
```

Q) When will the callback function get called inside the useEffect()?

👉🏽 Callback function is getting called after the whole component get rendered.

In our app we are using ’`useEffect()`' inside Body component. So it will get called once Body component complete its render cycle.

If we have to do something after the rendercycle complets we can pass it inside the ‘`useEffect()`' . this is the actual use case of *useEffect*. It is really helpful to render data which we will get after the ‘`fetch()`' operation and we are going to follow second approach which we have discussed already.

Q) Where we fetch the data?

👉🏽 inside the ‘`useEffect()`' we use ‘`fetchData()`' function to fetch data from the external world. don’t worry we will see each and every steps in detail.

logic of fetching the data is exactly the same that we used to do in javascript.

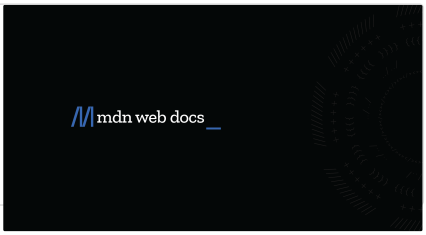
here we are fetching the swiggy’s API.

📢 IMPORTANT:
If getting difficulty to understand `fetch()`, Don’t worry please read about how `fetch()` works.👇🏽

Fetch basic concepts - Web APIs | MDN

The Fetch API provides an interface for fetching resources (including across the network). It will seem familiar to anyone who has used XMLHttpRequest, but it provides a more powerful and

https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Basic_concepts



Q) How can we use Swiggys API in our App?

👉🏽 We know that ‘`fetch()`’ always return promise to us. we can handle response using ‘`.then()`’ method.

but here we are using newer approach using ‘**async/await**’ to handle the promise.

we convert this data to javascript object by using ‘`.json`'

```
// here once the body component would have been rendered , we will fetch the d
ata
useEffect(() => {
  fetchData();
}, []);

const fetchData = async () => {
  const data = await fetch(
    "https://www.swiggy.com/dapi/restaurants/list/v5?lat=19.9615398&lng=79.296
1468&is-seo-homepage-enabled=true&page_type=DESKTOP_WEB_LISTING"
  );

  const json = await data.json();
  console.log(json);
};
```

Q) By using above code let’s see can we able to call swiggy’s api sucessfully or not?

👉🏽 We got an error 😅 (refer fig 6.1)

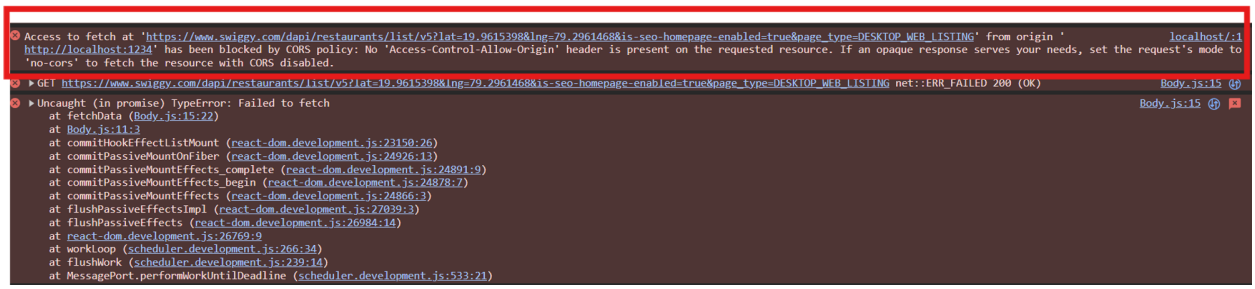


fig 6.1

Q) What is the reasone we got that error?

👉🏽 Basically calling swiggy’s API from local host has been blocked due to CORS policy.

Q) What exactly the CORS policy is?

👉🏽 (Cross-Origin Resource Sharing) is a system, consisting of transmitting HTTP headers, that determines whether browsers block frontend JavaScript code from accessing responses for cross-origin requests.

In simpler terms, CORS (Cross-Origin Resource Sharing) is a security feature implemented by browsers that restricts web pages from making requests to a different origin (domain) than the one from which it was served. Therefore, when trying to call Swiggy's API from localhost, the browser blocks the request due to CORS restrictions.

📢 IMPORTANT:
If getting difficulty to understand CORS, Don’t worry please read the below document 👇🏽

CORS - MDN Web Docs Glossary: Definitions of Web-related terms | MDN

CORS (Cross-Origin Resource Sharing) is a system, consisting of transmitting HTTP headers, that determines whether browsers block frontend JavaScript code from accessing responses for cross-origin requests.

https://developer.mozilla.org/en-US/docs/Glossary/CORS

---

https://www.youtube.com/watch?v=tcLW5d0KAYE

📢 IMPORTANT!

To prevent CORS errors when using APIs, utilize a CORS extension and activate it.

📢 IMPORTANT!
In future swiggy definately change their API data so always remember go to swiggy's website and copy the updated URL of API to fetch data.

To show the new data on our page, we just need to update the '`listOfRestaurant`' with the fresh info. React will then refresh the page to display the updated data.

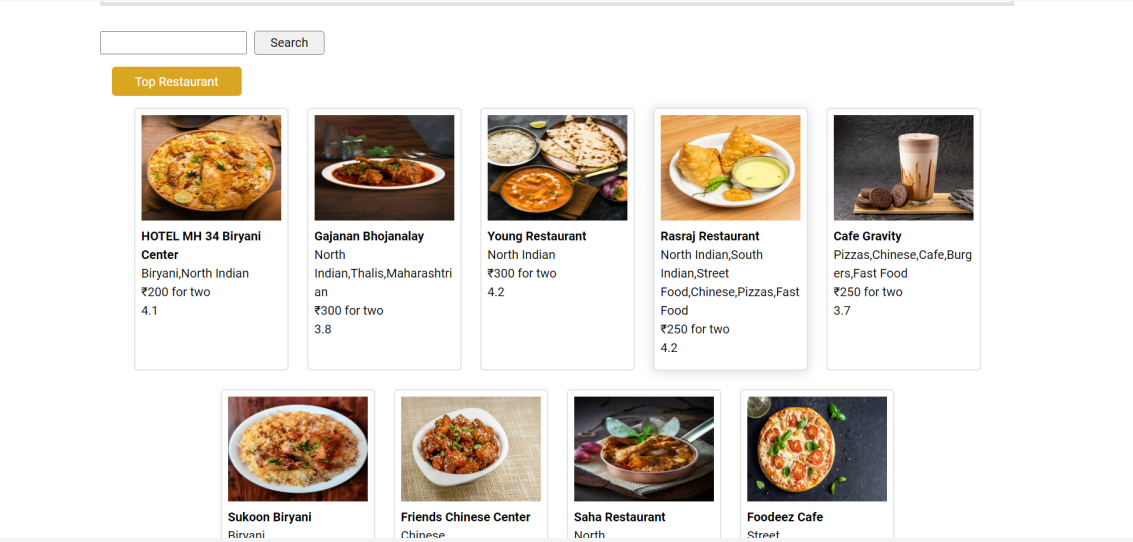
Q) How do we Update the data ?

👉🏽 We're updating the '`listOfRestaurant`' using a state variable we've already defined. We simply use the '`setlistOfRestaurant()`' function to replace the old data with the new.

```
const fetchData = async () => {
  const data = await fetch(
    "https://www.swiggy.com/dapi/restaurants/list/v5?lat=19.9615398&lng=79.296
1468&is-seo-homepage-enabled=true&page_type=DESKTOP_WEB_LISTING"
  );
  const json = await data.json();
  setListOfRestaurant(
    json.data.cards[4].card.card.gridElements.infoWithStyle.restaurants
  );
};
```

🌈 our App making live call to the external world getting the data and render on the screen. Amazing

Congratulations we have successfully fetch and render the data 🎉🎊



📢 IMPORTANT:

As we delve into the JSON data, it's essential to note its complexity. Our focus lies solely on extracting cards which have restaurant information for our project.

Attempting to directly implement the code snippet provided here definitely results in errors due to potential changes in Swiggy's API structure. Your understanding is greatly appreciated during this phase. Focus on the concept of whatever *Akshay* taught us in this Episode.

NOTE: In the upcoming episode, *'Akshay'* addresses all API-related issues, ensuring a smoother experience. So, don't stress—everything will be resolved in the upcoming episode.

Happy coding!

PART-4

After fetching the data, there's a noticeable one-second delay before it appears on the screen. This delay occurs because the APIs take some time to load. Improving this can enhance the user experience.

Q) How could we improve it?

👉 To enhance the user experience, we could add a spinning loader that appears while we wait for the data to load from the APIs. This provides visual feedback to the user and indicates that the application is working to retrieve the information.

We could implement a condition to display a spinning loader if our list of restaurants hasn't received any data yet.

```
if (listOfRestaurant.length === 0) {
  return <h1>loading. . .</h1>;
}
```

Refreshing the page to see the result,but this isn't an ideal approach. Instead, we can enhance the user experience by implementing a *'Shimmer UI'*.

Shimmer UI

Shimmer UI is a technique that shows placeholder content while data is loading, reducing wait time and keeping users engaged.

Instead of displaying a generic "loading" message, we'll integrate a `<shimmer/>` component within our app to provide visual feedback while data is loading. this concept is known as **'conditional rendering'**.

```
// conditional rendering
if (listOfRestaurant.length === 0) {
  return <Shimmer/>;
}
```

PART-5

Use of ternery oerator in our code base.

```
return listOfRestaurant.length === 0 ? (
  <Shimmer />
) : (
  <div className="container body">
    <div className="filter">
      <button
        className="filter-btn"
```

```
          onClick={() => {
            const filterLogic = listOfRestaurant.filter((res) => {
              return res.info.avgRating > 4.2;
            });
            setListOfRestaurant(filterLogic);
          }}
        >
          Top Restaurants
        </button>
      </div>
      <div className="ReastaurantContainer">
        {listOfRestaurant.map((restaurant) => (
          <RestaurantCard key={restaurant.info.id} resData={restaurant} />
        ))}
      </div>
    </div>
  );
```

Q) Why do we need State variable?

👉 Many developers have this confusion today we will see that Why with the help of following example:

to understand this we will introduce on feature in our app is a '**login/logout**' button

Inside Header component we are adding the button look at the code given below. also we want to make that login keyword dynamic it should change to logout after clicking.

step1 ——>

We create '`btnName`' variable with login string stored in it and we are going use that `btnName` as a button text look at the code below

step2 ——>

Upon clicking this button, it changes to 'logout'.

```
// Step 1-->
const btnName = "Login";

return (
  <div className="container header">
    <a>logo</a>
    {navItems}
    <button
      className="login"
      onclick={() => {
        btnName = "Logout";
```

```
        }}
      >
        {btnName}
      </button>
    </div>
  );
```

But it will not change 😒.

It's frustrating that despite updating the '`btnName`' value and seeing the change reflected in the console, the UI remains unchanged. This happens because we're treating '`btnName`' as a regular variable. To address this issue, we need a mechanism that triggers a UI refresh whenever '`btnName`' is updated
To ensure UI updates reflect changes in '`btnName`', we may need to use state management that automatically refreshes the UI when data changes. that the reasone we need state variable '`useState()`'.

Let's utilize '`reactBtn`' as a state variable using '`useState()`' instead of *btnName*
Here's the code:

```
const [reactBtn, setReactBtn] = useState("login");
```

To update the default value of '`reactBtn`', we use '`setReactBtn`'function.

> 📢 NOTE:
> In React, we can't directly update a state variable like we would use a normal JavaScript variable. Instead, we must use the function provided by the '`useState()`' hook. This function allows us to update the state and triggers a re-render of the component, ensuring our UI is always up-to-date with the latest state.

With the code provided below, we've enhanced the functionality of our app. Now, we can seamlessly toggle between "login" and "logout" states using a ternary operator. This addition greatly improves the user experience.

```
const [reactBtn, setReactBtn] = useState("login");

return (
        <div className="container header">
            <a>logo</a>
            {navItems}
            <button
```

```
                    className="login"
                    onClick={() => {
                        reactBtn === "login"
                            ? setReactBtn("logout")
                            : setReactBtn("login");
                    }}
                >
                    {reactBtn}
                </button>
            </div>
        );
```

📢 NOTE :
The interesting aspect of the above example is how we manage to modify a `const` variable like '`reactBtn`', which traditionally isn't possible. However, because React rerenders the entire component when a state variable changes, it essentially creates a new instance of '`reactBtn`' with the updated value. So, in essence, we're not updating '`reactBtn`'; instead, React creates a new one with the modified value each time the state changes. This is the beauty of React.

PART - 6

lets add another feature in our React app , search functionality.

When you input text into the search field, it provides suggestions based on the data related to restaurants that we already have.

step 1 ——>

Let's create a search bar within a `<div>` element and assign any class name of your choice to it. Additionally, we'll give class names to the input field and button inside the search bar.

step 2 ——>

Upon clicking the button, filter the restaurant cards and update the UI to retrieve data from the input box. To link our input to the button, we'll use the `value` attribute within the input field and bind that value to a local state variable. We'll create a local state variable named `searchText` along with a function named `setSearchText` to update the value. lets see below code will work or not by simple puting the call back function.

```
const [searchText, setsearchText] = useState("");

<div className="search">
  <input type="text" className="search-box" value={searchText} />
  <button className="searchBtn" onClick={() => {
  console.log(searchText);
  }}>
    Search
  </button>
</div>;
```

we could see that our input not taking value. we unable to type any thing.

- we knew already we have bind this searchText to the input field. what ever is inside the searchText variable will inside the value attribute of the input field.
- when we will change the value of input field by typing on it still it will tied to the searchText but searchText is not Updating . because default value of search text is empty string .this is most import point to understand whole concept. this input box not changed unless we change the search text

Q) How could we solve this problem ?

👉 to solve this we have to add '`onChange`' eventHandler inside the input field, so as soon as input changes the onchange call back function should also be changed the input text.

inside the onchange event handler we have event '**e**' inside the call back . so access that typed input by using event '**e**' see the code

```
<input type="text" className="search-box" value={searchText}
onchange={(e)=>{setSearchText(e.target.value)}/>
```

based on the on change we have make in the code now we can type inside the search box and see the output inside the console.

> 📢 NOTE: when ever search text is change on the every key press state variable re render the component. its find the difference between every updated V-DOM with new text added inside the input field with older one.

Step 3 ——>

👉 We're currently filtering the list of restaurants to update the UI. When we type a word in the input field, it filters out the restaurant cards based on

whether the typed word matches any restaurant names. However, we're facing a challenge with the input field being case-sensitive. We want the suggestions to be based solely on the word typed, without considering whether it's in uppercase or lowercase.

Q) How could we solve this problem ?

👉 To fix the problem, we just need to use the code provided. It uses '`toLowerCase()`' to make our search bar insensitive to capitalization.

```
<button
  className="searchBtn"
  onClick={() => {
    // filter the Restaurant and update the UI
    const filtertheRestaurant = listOfRestaurant.filter((res) => {
      return res.info.name.toLowerCase().includes(searchText.toLowerCase());
    });

    setListOfRestaurant(filtertheRestaurant);
  }}
>
  Search
</button>;
```

Step 4 ——>

We've encountered another issue in our app: after searching for a restaurant, the UI doesn't render anything when we search again. Instead, we only see the Shimmer UI.

How could we solve this problem ?

👉 here problem is when we search 1st time we are updating '`listOfRestaurents`'. If we try to search it again it is searching from previous updated list thats the problem. simple solution for this instead of filtering the original data we simple make a copy to of that original data in our case it is nothing but a '`listOfRestaurant`' and stored the copy with new variable `filteredRestaurant`.

```
//original
const [listOfRestaurant, setListOfRestaurant] = useState([]);

//copy
const [filteredRestaurant, setFilteredRestaurant] = useState([]);
```

In our code, when we fetch data using the '`fetchData`' function, it's important to update the rendering to display the new data. We achieve this by updating the state variables '`listOfRestaurant`' and '`filteredListOfRestaurant`' using functions provided

by the ‘`useState()`' hook. Initially, both arrays are empty, but after fetching data, we fill them with the retrieved information. otherwise we won’t see any thing on the page.

```
const fetchData = async () => {
        const data = await fetch(
            "https://www.swiggy.com/dapi/restaurants/list/v5?lat=19.9615398&ln
g=79.2961468&is-seo-homepage-enabled=true&page_type=DESKTOP_WEB_LISTING"
        );
        const json = await data.json();
// here we are filling both the variable with new data with the help of their
functions.
        setListOfRestaurant(
            json.data.cards[4].card.card.gridElements.infoWithStyle.restaurant
s
        );
        setFilteredRestaurant(
            json.data.cards[4].card.card.gridElements.infoWithStyle.restaurant
s
        );
    };
```

📢 IMPORTANT:

here there is two important points to remember

1) When we need to modify the list Of Restaurants based on certain conditions, we're essentially using the original data we fetched and stored within the ‘`listOfRestaurant`' variable. (original)

2) To display data on the UI, we use a copy of ‘`listOfRestaurant`' called ‘`filteredRestaurant`'. (copy)

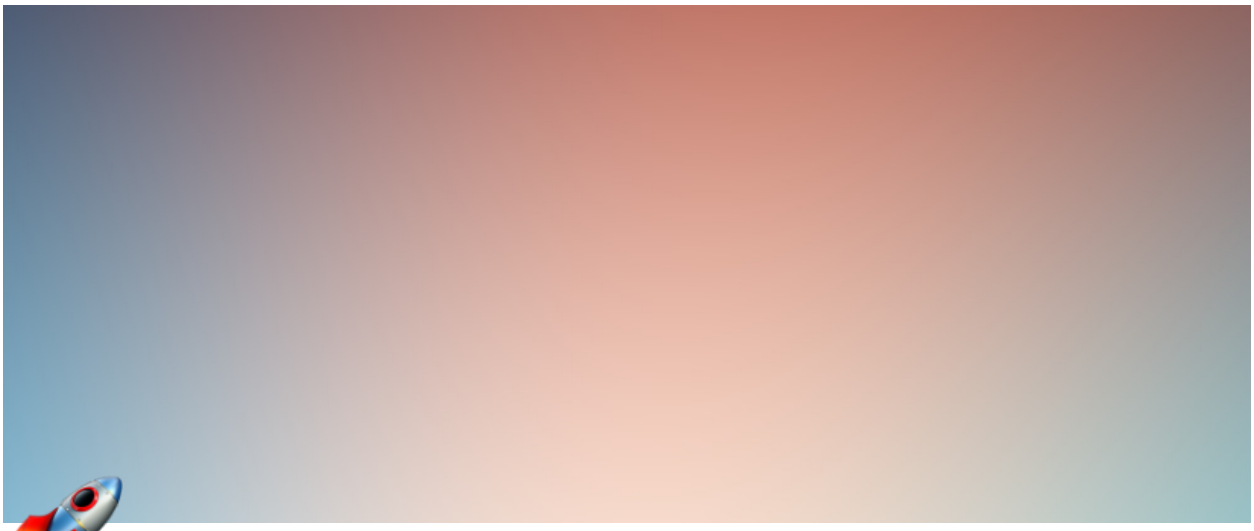
Final code base given below.

```
import { useState, useEffect } from "react";
import RestaurantCard from "./RestaurantCard";
import Shimmer from "./Shimmer";

const Body = () => {
  const [listOfRestaurant, setListOfRestaurant] = useState([]);
  const [filteredRestaurant, setFilteredRestaurant] = useState([]);
```

```
  const [searchText, setSearchText] = useState("");

  useEffect(() => {
    fetchData();
  }, []);

  const fetchData = async () => {
    const data = await fetch(
      "https://www.swiggy.com/dapi/restaurants/list/v5?lat=19.9615398&lng=79.2
961468&is-seo-homepage-enabled=true&page_type=DESKTOP_WEB_LISTING"
    );
    const json = await data.json();

    setListOfRestaurant(
      json?.data?.cards[4]?.card?.card?.gridElements?.infoWithStyle?.restauran
ts
    );
    setFilteredRestaurant(
      json?.data?.cards[4]?.card?.card?.gridElements?.infoWithStyle?.restauran
ts
    );
  };

  return listOfRestaurant.length === 0 ? (
    <Shimmer />
  ) : (
    <div className="container body">
      <div className="filter-btn">
        <div className="search">
          <input
            type="text"
            className="search-box"
            value={searchText}
            onChange={(e) => {
              setSearchText(e.target.value);
            }}
          />
          <button
            className="searchBtn"
            onClick={() => {
              // filter the Restaurant and update the UI.

              const filteredRestaurant = listOfRestaurant.filter((res) => {
                return res.info.name
```

```
                .toLowerCase()
                .includes(searchText.toLowerCase());
            });

            setFilteredRestaurant(filteredRestaurant);
          }}
        >
          Search
        </button>
      </div>

      <button
        onClick={() => {
          const filterLogic = listOfRestaurant.filter((res) => {
            return res.info.avgRating > 4;
          });
          setFilteredRestaurant(filterLogic);
        }}
      >
        Top Restaurants
      </button>
    </div>
    <div className="ReastaurantContainer">
      {filteredRestaurant.map((restaurant) => (
        <RestaurantCard key={restaurant.info.id} resData={restaurant} />
      ))}
    </div>
  </div>
);
};

export default Body;
```

THANK YOU!



Episode-07 | Finding The Path

💡 Please make sure to follow along with the whole "**Namaste React**" series, starting from Episode-1 and continuing through each subsequent episode. The notes are designed to provide detailed explanations of each concept along with examples to ensure thorough understanding. Each episode builds upon the knowledge gained from the previous ones, so starting from the beginning will give you a comprehensive understanding of React dvelopment.

💡 I've got a quick tip for you. To get the most out of these notes, it's a good idea to watch **Episode-07** first. Understanding what "**Akshay**" shares in the video will make these notes way easier to understand.

Q) What are various ways to add images to our App? Explain with code examples

There are several ways to add and display images.

```
1. Importing images using ES6 Modules
2. Using public folder
3. Loading images from a remote source
4. Using image assets within CSS
```

1 `Importing images using ES6 Modules` - We can import images directly using ES6 modules. This is a common approach for `small to medium-sized apps`, and it's straightforward. Firstly, We have to place our image in the project directory (e.g., in the src folder or a subfolder).

Example:

```
import React from 'react';
import myImage from './my_image.jpg';

function App() {
    return (
        <div>
            <img src={myImage} alt="My Image" />
        </div>
    );
}

export default App;
```

2 `Using public folder` - If we want to reference images in the public folder, we can do so without importing them explicitly. This method is useful for handling large image assets or for dynamic image URLs. Place your image in the public directory.

```
// public/my_image.jpg
```

Then, reference it in your code:

```
import React from 'react';

function App() {
    return (
        <div>
            <img src={process.env.PUBLIC_URL + '/my_image.jp
g'} alt="My Image" />
        </div>
    );
}

export default App;
```

3 `Loading images from a remote source` - We can load images from a remote source, such as an external URL or a backend API, by specifying the image URL directly in our img tag.

Example:

```
import React from 'react';

function App() {
    const imageUrl = 'https://example.com/my_image.jpg';

    return (
        <div>
            <img src={imageUrl} alt="My Image" />
        </div>
    );
}

export default App;
```

4 `Using image assets within CSS` - We can also use images as background images or in other CSS styling. In this case, we can reference the image in your CSS file.

Example CSS (styles.css):

```css
.image-container {
    background-image: url('/my_image.jpg');
    width: 300px;
    height: 200px;
}
```

Then, apply the CSS class to your JSX:

```jsx
import React from 'react';
import './styles.css';

function App() {
    return (
        <div className="image-container">
            {/* Content goes here */}
        </div>
    );
}

export default App;
```

Choose the method that best fits your project's requirements and organization. Importing images using ES6 modules is the most common and convenient approach for most React applications, especially for small to medium-sized projects. For larger projects with many images, consider the folder structure and organization to keep our code clean and maintainable.

Q) What would happen if we do console.log(useState())?

> If you use `console.log(useState())` in a React functional component, it will display the result of calling the useState()

> function in our browser's developer console. The useState() function is a React Hook that is typically used to declare a state variable in a functional component. When we call useState(), it `returns an array with two elements: the current state value and a function to update that state value`.

For example:

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

In this example, **count** is the current state value, and **setCount** is the function to update it.

If we do console.log(useState()), we will see something like this in the console:

```
[0, Function]
```

The first element of the array is the initial state value (in this case, 0), and the second element is the function to update the state. However, using console.log(useState()) directly in our component without destructuring the array and assigning names to these elements isn't a common or recommended practice. Normally, we would destructure the array elements when using useState() to make our code more readable and maintainable.

So, it's more typical to use useState() like this:

```
const [count, setCount] = useState(0);
console.log(count); // Logs the current state value
console.log(setCount); // Logs the state update function
```

This way, we can access and work with the state and state update function in our component.

Q) How will useEffect behave if we don't add a dependency array?

> In React, when we use the `useEffect` hook `without providing a dependency array`, the effect will be executed on every render of the component. This means that the code inside the useEffect will run both after the initial render and after every subsequent render.

💡 Here's an example of using `useEffect without a dependency array`

```
import React, { useEffect } from 'react';

function MyComponent() {
  useEffect(() => {
    // This code will run on every render
    console.log('Effect executed');
  });

  return (
    <div>
      {/* Component content */}
    </div>
  );
}
```

In this example, the `useEffect without a dependency array doesn't specify any dependencies, so it will run after every render of MyComponent`. This behavior can be useful in some cases, but it's essential to be cautious when using useEffect without a dependency array because it can lead to performance issues, especially if the effect contains expensive operations.

When we don't provide a dependency array, the effect is considered to have an empty dependency array, which is equivalent to specifying every value as a dependency. Therefore, it's important to understand the consequences of running the effect on every render and to use this pattern judiciously.

In many cases, we might want to include a dependency array to control when the effect should run based on changes in specific variables or props. This can help optimize the performance of our component and prevent unnecessary re-renders.

💡 **Syntax**

```
useEffect(() => {}, []);
```

`Case 1, when the dependency array is not included as an argument in the useEffect hook`, the callback function inside useEffect will be executed every time the component is initially rendered and subsequently re-rendered. This means that the effect runs on every render cycle, and there are no dependencies that control when it should or should not execute.

Here's the relevant code again for reference:

```
import React, { useEffect } from 'react';

function MyComponent() {
  useEffect(() => {
    // This code will run on every render
    console.log('Effect executed');
  });

  return (
    <div>
      {/* Component content */}
    </div>
  );
}
```

The callback function in the useEffect will log `Effect executed` to the console every time MyComponent is rendered or re-rendered. This behavior can be useful in some cases but should be used carefully to avoid excessive or unnecessary

executions of the effect. If we want more control over when the effect should run, we can include a dependency array to specify the dependencies that trigger the effect when they change.

`In Case 2, when the dependency array is empty (i.e., []) in the arguments of the useEffect hook`, the callback function will indeed be executed once during the initial render of the component. However, it won't be limited to the initial render only. It will run after the initial render and then on every re-render of the component.

Here's an example of using `useEffect with an empty dependency array`:

```
import React, { useEffect } from 'react';

function MyComponent() {
  useEffect(() => {
    // This code will run after the initial render and on eve
ry re-render
    console.log('Effect executed');
  }, []);

  return (
    <div>
      {/* Component content */}
    </div>
  );
}
```

That's not accurate. In Case 2, when the dependency array is empty (i.e., []) in the arguments of the useEffect hook, the callback function will indeed be executed once during the initial render of the component. However, it won't be limited to the initial render only. It will run after the initial render and then on every re-render of the component.

Here's an example of using useEffect with an empty dependency array:

```
import React, { useEffect } from 'react';

function MyComponent() {
```

```
  useEffect(() => {
    // This code will run after the initial render and on eve
ry re-render
    console.log('Effect executed');
  }, []);

  return (
    <div>
      {/* Component content */}
    </div>
  );
}
```

In this case, the callback function in the useEffect with an empty dependency array will run once after the initial render and then on every subsequent re-render of MyComponent. It won't run if the component is unmounted and then re-mounted, but it will run whenever the component is re-rendered, even if there are no dependencies to watch for changes.

If you want the effect to run only once, and not re-run on re-renders, you can specify an empty dependency array like this:

```
useEffect(() => {
  // This code will run only once, after the initial render
  console.log('Effect executed');
}, []);
```

In this case, the effect will run only after the initial render, and it won't run again on subsequent re-renders.

`Case 3 - When the dependency array in the arguments of the useEffect hook contains a condition` (a variable or set of variables), the callback function will be executed once during the initial render of the component and also on re-renders if there is a change in the condition.

Here's an example of using `useEffect with a condition in the dependency array`:

```jsx
import React, { useEffect, useState } from 'react';

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

  useEffect(() => {
    // This code will run after the initial render and whenev
er 'count' changes
    console.log('Effect executed');
  }, [count]);

  return (
    <div>
      <button onClick={() => setCount(count + 1)}>Increment C
ount</button>
      <p>Count: {count}</p>
    </div>
  );
}
```

In this case, the useEffect has count as a dependency in the array. This means that the effect will run after the initial render and then again whenever the count variable changes. If we click the Increment Count button, the count state will change, triggering the effect to run again. If the condition specified in the dependency array doesn't change, the effect won't run on re-renders.

This allows us to control when the effect runs based on specific conditions or dependencies. It's a useful way to ensure that the effect only runs when the relevant data or state has changed.

Q) What is `SPA` ?

> SPA stands for `Single Page Application`. It's a type of web application or website that interacts with the user by

> dynamically rewriting the current web page rather than loading entire new pages from the server. In other words, a single HTML page is loaded initially, and then the content is updated dynamically as the user interacts with the application, typically through JavaScript.

`Key characteristics of SPAs include`:

`Dynamic Updates` - In SPAs, content is loaded and updated without requiring a full page reload. This is achieved using JavaScript and client-side routing.

`Smooth User Experience` - SPAs can provide a smoother and more responsive user experience because they can update parts of the page without the entire page needing to be refreshed.

`Faster Initial Load` - While the initial load of an SPA might take longer as it downloads more JavaScript and assets, subsequent interactions with the application can be faster because only data is exchanged with the server and not entire HTML pages.

`Client-Side Routing` - SPAs often use client-side routing to simulate traditional page navigation while staying on the same HTML page. This is typically achieved using libraries like React Router or Vue Router.

`API-Centric` - SPAs are often designed to be more API-centric, where the client communicates with a backend API to fetch and send data, usually in JSON format. This allows for decoupling the front end and back end.

`State Management` - SPAs often use state management libraries (e.g., Redux for React or Vuex for Vue) to manage the application's state and data flow.

Popular JavaScript frameworks and libraries like React, Angular, and Vue are commonly used to build SPAs. They offer tools and patterns to create efficient and maintainable single-page applications.

Q) What is the difference between `Client Side Routing` and `Server Side Routing`?

A: Client-side routing and server-side routing are two different approaches to handling routing and navigation in web applications. They have distinct characteristics and are often used for different purposes. Here's an overview of the key differences between them:

`Client-Side Routing`: `Handling on the Client` - In client-side routing, routing and navigation are managed on the client side, typically within the web browser. JavaScript frameworks and libraries, such as React Router (for React applications) or Vue Router (for Vue.js applications), are commonly used to implement client-side routing.

`Faster Transitions` - Client-side routing allows for faster page transitions since it doesn't require the server to send a new HTML page for each route change. Instead, it updates the DOM and URL dynamically without full page reloads.

`Single-Page Application (SPA)` - Client-side routing is often associated with single-page applications (SPAs), where the initial HTML page is loaded, and subsequent page changes are made by updating the content using JavaScript.

`SEO Challenges` - SPAs can face challenges with search engine optimization (SEO) because search engine crawlers may not fully index the content that relies heavily on client-side rendering. Special techniques like server-side rendering (SSR) or pre-rendering can be used to address this issue.

`Route Management` - Routing configuration is typically defined in code and managed on the client side, allowing for dynamic and flexible route handling.

- - `Server-Side Routing`:

`Handling on the Server` - Server-side routing manages routing and navigation on the server. When a user requests a different URL, the server generates and sends a new HTML page for that route.

`Slower Transitions` - Server-side routing tends to be slower in terms of page transitions compared to client-side routing, as it involves full page reloads.

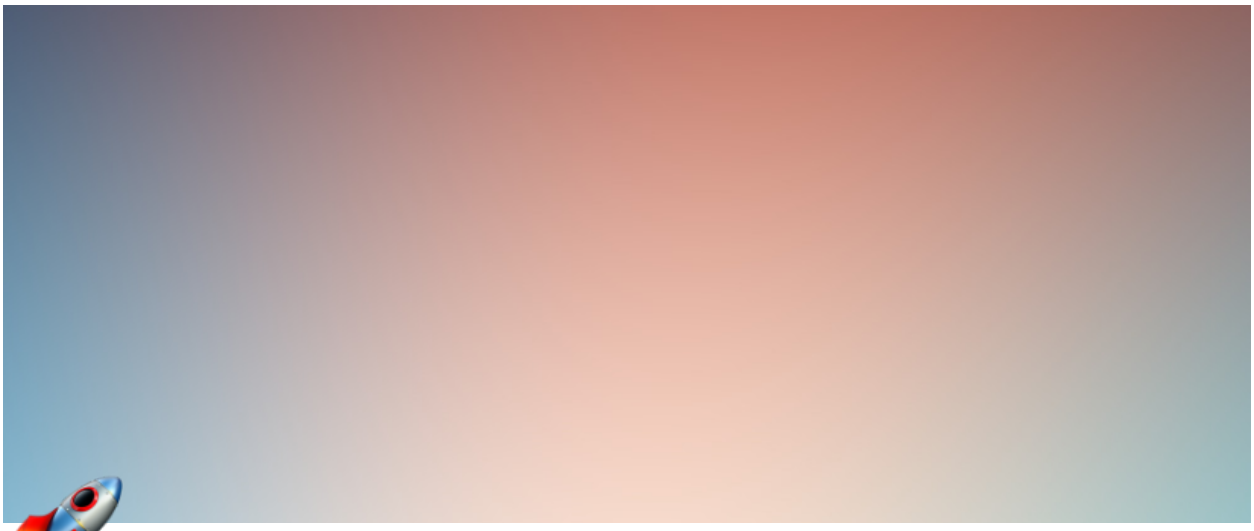
`Traditional Websites` - Server-side routing is commonly used for traditional multi-page websites where each page is a separate HTML document generated by the server.

`SEO-Friendly` - Server-side routing is inherently more SEO-friendly, as each page is a separate HTML document that can be easily crawled and indexed by search

engines.

`Route Configuration` - Routing configuration in server-side routing is typically managed on the server, and URLs directly correspond to individual HTML files or routes.

In summary, client-side routing is suitable for building SPAs and offers faster, more interactive user experiences but can pose SEO challenges. Server-side routing is more SEO-friendly and is used for traditional websites with separate HTML pages, but it can be slower in terms of page transitions. The choice between these two routing approaches depends on the specific requirements and goals of a web application or website. In some cases, a hybrid approach that combines both client-side and server-side routing techniques may be used to achieve the best of both worlds.



Episode-08 | Let’s Get Classy

💡 Please make sure to follow along with the whole "**Namaste React**" series, starting from Episode-1 and continuing through each subsequent episode. The notes are designed to provide detailed explanations of each concept along with examples to ensure thorough understanding. Each episode builds upon the knowledge gained from the previous ones, so starting from the beginning will give you a comprehensive understanding of React dvelopment.

💡 I've got a quick tip for you. To get the most out of these notes, it's a good idea to watch **Episode-08** first. Understanding what "**Akshay**" shares in the video will make these notes way easier to understand.

Q) How do you `create Nested Routes react-router-dom configuration` ?

> In React applications using react-router-dom, we can create nested routes by nesting our `<Route>` components inside each other within the route configuration. This allows you to define routes and components hierarchically, making it easier to manage the routing structure of your application.

💡 Here's a step-by-step guide on how to create nested routes using react-router-dom:

1 `Install react-router-dom if we haven't already`:

```
npm install react-router-dom
```

2 `Import the necessary components from react-router-dom in your application file`:

```
import { BrowserRouter as Router, Route, Switch } from 'react
-router-dom';
```

1. Defining our route hierarchy by nesting components within each other. Typically, this is done within a component that acts as a layout or container for the nested routes. For example, if we have a layout component called Layout:

```
import React from 'react';
import { Route } from 'react-router-dom';

// Import your nested route components
import Home from './Home';
import About from './About';

function Layout() {
  return (
    <div>
      <h1>My App</h1>
```

```
      <Route path="/home" component={Home} />
      <Route path="/about" component={About} />
    </div>
  );
}

export default Layout;
```

1. In our main application file, wrap our entire application with the Router component, and use the component to render only the first matching route:

```
import React from 'react';
import { BrowserRouter as Router, Route, Switch } from 'react
-router-dom';

// Import your Layout component that defines nested routes
import Layout from './Layout';

function App() {
  return (
    <Router>
      <Switch>
        <Route path="/" component={Layout} />
      </Switch>
    </Router>
  );
}

export default App;
```

Now we have a simple example of nested routes. In this case, the Layout component defines the `/home` and `/about` routes, and these nested routes can have their own components and nested routes as well. We can continue to nest routes further by adding more `<Route>` components inside the Home and About components to create a more complex routing structure. Remember that this is

just a basic example, and we can customize our routing structure based on the requirements of our application. We can also use the exact prop on routes to ensure that only the exact path is matched if needed.

Q) Read about `createHashRouter`, `createMemoryRouter` from React Router docs.

1. `createHashRouter` - createHashRouter is part of the React Router library and provides routing capabilities for single-page applications (SPAs). It's commonly used for building client-side navigation within applications. Unlike traditional server-side routing, it uses the fragment identifier (hash) in the URL to manage and handle routes on the client side. This means that changes in the URL after the # symbol do not trigger a full page reload, making it suitable for SPAs.

To use createHashRouter, we typically import it from the React Router library and define our routes using Route components. Here's a basic example of how you might use it:

```
import { createHashRouter, Route } from 'react-router-dom';

const App = () => (
  <createHashRouter>
    <Route path="/" component={Home} />
    <Route path="/about" component={About} />
    <Route path="/contact" component={Contact} />
  </createHashRouter>
);
```

1. `createMemoryRouter` - createMemoryRouter is another routing component provided by React Router. Unlike createHashRouter or BrowserRouter, createMemoryRouter is not associated with the browser's URL. Instead, it allows you to create an in-memory router for testing or other scenarios where you don't want to interact with the actual browser's URL.

💡 Here's a simple example of how to use createMemoryRouter:

```
import { createMemoryRouter, Route } from 'react-router-dom';

const App = () => (
  <createMemoryRouter>
    <Route path="/" component={Home} />
    <Route path="/about" component={About} />
    <Route path="/contact" component={Contact} />
  </createMemoryRouter>
);
```

In both cases, we define our application's routes within the router component and specify the components to render for each route. The choice between `createHashRouter` and `createMemoryRouter` depends on our specific use case, such as whether we're building an SPA that interacts with the browser's URL or a scenario where we need an in-memory router for testing.

Q) What is the `order of life cycle method calls in Class Based Components` ?

`Constructor` - The constructor method is the first to be called when a component is created. It's where we typically initialize the component's state and bind event handlers.

`Render` - The render method is responsible for rendering the component's UI. It must return a React element (typically JSX) representing the component's structure.

`ComponentDidMount` - This method is called immediately after the component is inserted into the DOM. It's often used for making AJAX requests, setting up subscriptions, or other one-time initializations.

`ComponentDidUpdate` - This method is called after the component has been updated (re-rendered) due to changes in state or props. It's often used for side effects, like

updating the DOM in response to state or prop changes.

`ComponentWillUnmount` - This method is called just before the component is removed from the DOM. It's used to clean up resources or perform any necessary cleanup.

For more reference React-Lifecycle-methods

Q) Why do we use `componentDidMount` ?

The `componentDidMount` lifecycle method in React class-based components is used for a specific purpose: it is called immediately after a component is inserted into the DOM (Document Object Model). This makes it a crucial point in the component's lifecycle and provides a valuable opportunity to perform various tasks that require interaction with the DOM or external data sources. Here are some common use cases for componentDidMount:

`Fetching Data` - It's often used to make asynchronous requests to fetch data from APIs or external sources. This is a common scenario for components that need to display dynamic content.

`DOM Manipulation` - When we need to interact with the DOM directly, such as selecting elements, setting attributes, or applying third-party libraries that require DOM elements to be present, we can safely do so in componentDidMount. This is because the component is guaranteed to be in the DOM at this point.

```
class MyComponent extends React.Component {
  componentDidMount() {
    // Fetch data from an API
    fetch('https://api.example.com/data')
      .then(response => response.json())
      .then(data => {
        // Update the component's state with the fetched data
        this.setState({ data });
      })
      .catch(error => {
        // Handle any errors
        console.error(error);
      });
```

```
  }

  render() {
    // Render component based on state
    return (
      <div>{/* Display data from this.state.data */}</div>
    );
  }
}
```

By using `componentDidMount`, we can ensure that the data fetching or other side effects happen after the initial render and that our component interacts with the DOM or external data sources at the right time in the component's lifecycle.

Q) Why do we use `componentWillUnmount`? Show with example.

> The `componentWillUnmount` lifecycle method in React class-based components is used to perform cleanup and teardown tasks just before a component is removed from the DOM. It's a crucial part of managing resources and subscriptions to prevent memory leaks and ensure that the component's behavior is properly cleaned up. Here's why and when we should use componentWillUnmount:

1 `Cleanup Resources` - If your component has allocated any resources, such as event listeners, subscriptions, timers, or manual DOM manipulations, it's essential to release these resources to prevent memory leaks. componentWillUnmount is the appropriate place to do this.

2 `Cancel Pending Requests` - If your component has initiated any asynchronous requests, such as AJAX calls or timers, you should cancel or clean them up to avoid unexpected behavior after the component is unmounted.

💡 Here's an example of using componentWillUnmount to remove an event listener when a component is unmounted:

```
class MyComponent extends React.Component {
  constructor() {
    super();
    this.handleResize = this.handleResize.bind(this);
  }

  componentDidMount() {
    // Add a window resize event listener when the component
is mounted
    window.addEventListener('resize', this.handleResize);
  }

  componentWillUnmount() {
    // Remove the window resize event listener when the compo
nent is unmounted
    window.removeEventListener('resize', this.handleResize);
  }

  handleResize(event) {
    // Handle the resize event
    console.log('Window resized:', event);
  }

  render() {
    return <div>My Component</div>;
  }
}
```

In this example, the component adds a resize event listener to the window when it's mounted, and it removes that listener in the componentWillUnmount method.

This ensures that the event listener is properly cleaned up when the component is unmounted, preventing memory leaks or unexpected behavior.

By using componentWillUnmount, we can ensure that any cleanup tasks are executed reliably when the component is no longer needed, helping to maintain the integrity of our application and avoiding potential issues.

Q) (Research) Why do we use `super(props)` in constructor?

> In JavaScript, when you define a class that extends another class (inherits from a parent class), we often use the super() method with props as an argument in the constructor of the child class. This is commonly seen in React when you create class-based components. The super(props) call is used for the following reasons:

`Access to Parent Class's Constructor` - When a child class extends a parent class, the child class can have its constructor. However, if the child class has a constructor, it must call super(props) as the first statement in its constructor. This is because super(props) is used to invoke the constructor of the parent class, ensuring that the parent class's initialization is performed before the child class's constructor code is executed. It is essential to maintain the inheritance chain correctly.

`Passing Props to the Parent Constructor` - By passing props to super(props), we ensure that the props object is correctly passed to the parent class's constructor. This is important because the parent class may need to set up its properties or handle the props somehow. By calling super(props), we make the props available for the parent class's constructor to work with.

💡 Here's an example of how super(props) is used in a React component:

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props); // Call the constructor of the parent class
(React.Component)
    // Initialize your component's state or perform other set
up
  }

  render() {
    // Render the component based on its state and props
    return <div>{this.props.someProp}</div>;
  }
}
```

In this example, the super(props) call ensures that the `React.Component` class's constructor is called, which is necessary for React to set up the component correctly. This is especially important because React uses the props object to pass data from parent components to child components. By calling super(props), we make sure that the props are properly handled in the parent class's constructor, and we can access them in our child component.

In modern JavaScript and React, it's also common to define a constructor without explicitly calling super(props), and it will be automatically called for us. However, if we define a constructor in a child class, and the parent class has its constructor, it's a good practice to include super(props) to ensure that the parent class's constructor is invoked correctly.

Q) (Research) Why can't we have the `callback function` of `useEffect async` ?

A: In React, the `useEffect` hook is designed to handle side effects in functional components. It's a powerful and flexible tool for managing asynchronous operations, such as data fetching, API calls, and more. However, useEffect itself cannot directly accept an async callback function. This is because useEffect expects its callback function to return either nothing (i.e., undefined) or a cleanup

function, and it doesn't work well with Promises returned from async functions. There are a few reasons for this:

`Return Value Expectation` - The primary purpose of the useEffect callback function is to handle side effects and perform cleanup. React expects us to either return nothing (i.e., undefined) from the callback or return a cleanup function. An async function returns a Promise, and it doesn't fit well with this expected behavior.

`Execution Order and Timing` - With async functions, we might not have fine-grained control over the execution order of the asynchronous code and the cleanup code. React relies on the returned cleanup function to handle cleanup when the component is unmounted or when the dependencies specified in the useEffect dependency array change. If you return a Promise, React doesn't know when or how to handle cleanup.

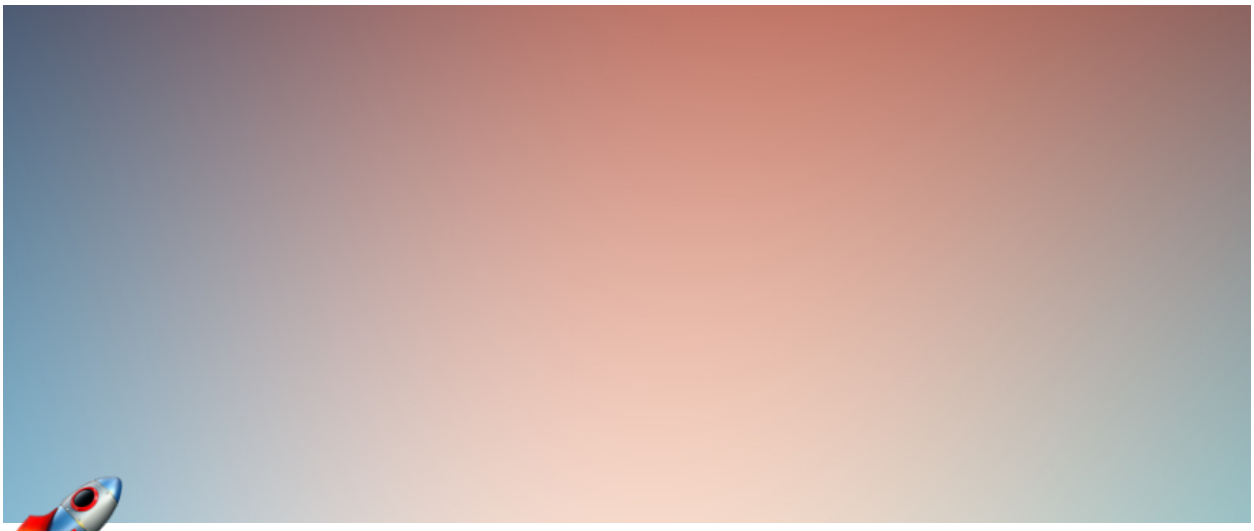
To work with async operations within a useEffect, we can use the following pattern:

```
useEffect(() => {
  const fetchData = async () => {
    try {
      // Perform asynchronous operations
      const result = await someAsyncOperation();
      // Update the state with the result
      setState(result);
    } catch (error) {
      // Handle errors
      console.error(error);
    }
  };

  fetchData(); // Call the async function

  return () => {
    // Cleanup code, if necessary
    // This function will be called when the component unmoun
ts or when dependencies change
```

```
  };
}, [/* dependency array */]);
```

In this pattern, we define an async function within the useEffect callback, perform our asynchronous operations, and then call that function. Additionally, we return a cleanup function from the useEffect to handle any necessary cleanup tasks when the component unmounts or when specified dependencies change.

By using this approach, we can effectively manage asynchronous operations with useEffect while adhering to React's expectations for the callback function's return value.



Episode-09 | Optimising Our App

💡 Please make sure to follow along with the whole "**Namaste React**" series, starting from Episode-1 and continuing through each subsequent episode. The notes are designed to provide detailed explanations of each concept along with examples to ensure thorough understanding. Each episode builds upon the knowledge gained from the previous ones, so starting from the beginning will give you a comprehensive understanding of React dvelopment.

💡 I've got a quick tip for you. To get the most out of these notes, it's a good idea to watch **Episode-09** first. Understanding what "**Akshay**" shares in the video will make these notes way easier to understand.

Q) When and why do we need `lazy()` ?

> The `lazy()` function is a feature in React that allows us to `load components dynamically, or lazily, only when they are needed`. This can be beneficial for improving the performance and load times of our web application, especially if it contains a large number of components or if some components are rarely used. Here's when and why we might need to use lazy():

1. `Code Splitting and Reducing Initial Bundle Size` - *In large React applications, bundling all components into a single JavaScript file can result in a large initial bundle size. This can lead to slower load times for users. By using lazy(), we can split our code into smaller, more manageable chunks. These chunks are loaded on-demand, reducing the initial bundle size and improving the time it takes for our application to load.*
2. `Improved Performance` - *Lazy loading can lead to better application performance. Components that are only loaded when needed reduce the amount of code that needs to be executed during the initial page load, which can lead to faster rendering times and a smoother user experience.*
3. `Faster Initial Load` - *When we use lazy(), only the essential code is loaded initially. Less code to parse and execute means that our application can start up faster, especially in scenarios where not all components are used right away.*
4. `Better User Experience` - *By deferring the loading of components until they are needed, we can provide a more responsive user experience. Users don't have to wait for unnecessary components to load, and they can interact with the parts of the application that are immediately visible.*
5. `Reducing Browser Caching Overhead` - *Smaller initial bundles produced by lazy() can also benefit from browser caching. Since components are loaded as separate chunks, once loaded, they are less likely to change frequently. This can result in a better caching strategy and faster subsequent visits to our site for returning users.*
6. `Optimizing Mobile Performance` - *On mobile devices with limited bandwidth and processing power, lazy loading is even more important. Smaller initial bundles*

can make our application more accessible and usable on mobile devices.

📌 Here's an example of how to use lazy() to load a component dynamically :

```
import React, { lazy, Suspense } from 'react';

const LazyComponent = lazy(() => import('./LazyComponent'));

function App() {
  return (
    <div>
      <Suspense fallback={<div>Loading...</div>}>
        <LazyComponent />
      </Suspense>
    </div>
  );
}

export default App;
```

In this example, the `LazyComponent` is only loaded when it is rendered. The `Suspense` component allows us to specify a loading indicator while the component is being loaded. This way, we can ensure a smooth user experience even during the asynchronous loading process.

In summary, we need to use lazy() when we want to optimize the performance and user experience of our React application by reducing the initial bundle size and deferring the loading of components until they are needed. This is particularly beneficial in large applications or when targeting slower connections and devices.

Q) What is `suspense` ?

> In React, `Suspense` is a feature that allows us to declaratively manage asynchronous data fetching and code-splitting in our applications. It is primarily used in combination with the lazy() function for dynamic imports and with the React.lazy() component to improve the user experience when loading data or components asynchronously.

`Here are the main aspects and use cases of Suspense:`

`Data Fetching` - Suspense can be used to handle the loading of asynchronous data, such as data from an API. It provides a way to specify a fallback UI (e.g., a loading spinner or a message) that is displayed while the data is being fetched. This is especially useful for making our application more user-friendly and responsive.

`Code Splitting` - When used with lazy() or React.lazy(), Suspense can manage the loading of code-split components. We can specify a fallback component or loading indicator to display while the component is being loaded. This helps in reducing the initial bundle size and improving the application's performance.

`Error Handling` - Suspense can also handle errors that might occur during data fetching or code splitting. We can specify how to render an error component or message in case an error occurs during the asynchronous operation.

📌 Here's a basic example of using Suspense for data fetching:

```
import React, { Suspense } from 'react';

const fetchData = () => {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve("Data fetched!");
    }, 2000);
  });
};
```

```
function DataFetchingComponent() {
  const data = fetchData();

  return (
    <div>
      <Suspense fallback={<div>Loading data...</div>}>
        <AsyncDataComponent data={data} />
      </Suspense>
    </div>
  );
}

function AsyncDataComponent({ data }) {
  return <div>{data}</div>;
}
```

In this example, when the `DataFetchingComponent` is rendered, it starts fetching data asynchronously. The Suspense component wraps the AsyncDataComponent, specifying a fallback UI to display while the data is being fetched.

Suspense can also handle errors by using an error boundary. If an error occurs during data fetching or code-split component loading, we can catch and handle the error gracefully.

While Suspense simplifies managing asynchronous operations and loading states in our React application, it's essential to be aware of the version of React we are using. Suspense for data fetching was introduced in React 18 and may have different usage patterns compared to Suspense for code-splitting, which has been available since React 16.6. Depending on the version of React, we might need to adjust our code accordingly.

Q) Why do we get this error: `A component was suspended while responding to synchronous input. This will cause the UI to be replaced with a loading indicator. To fix this, updates that suspend should`

`be wrapped with start transition` ? How does suspense fix this error?

> The error message you provided, "A component was suspended while responding to synchronous input. This will cause the UI to be replaced with a loading indicator. To fix this, updates that suspend should be wrapped with start transition," is related to React's Suspense feature and is typically encountered in asynchronous contexts where components are fetching data or handling code splitting.

To understand this error and how to fix it, you need to know a bit about how Suspense works and why it's important. Suspense is used to manage asynchronous data fetching and code-splitting, allowing you to display a loading indicator while the data or code is being fetched. When React encounters a Suspense boundary (created using), it knows that there might be a delay in rendering, and it can handle that situation gracefully.

The error message you received is telling you that a component that was responding to synchronous input (meaning it's not supposed to be waiting for anything) encountered a suspension. This should not happen because Suspense is primarily designed to handle asynchronous operations, and you generally don't want to introduce delays in the rendering of synchronous user interactions.

> 💡 Here's how to fix this error:

The error message you provided, "A component was suspended while responding to synchronous input. This will cause the UI to be replaced with a loading indicator. To fix this, updates that suspend should be wrapped with start transition," is related to React's Suspense feature and is typically encountered in asynchronous contexts where components are fetching data or handling code splitting.

To understand this error and how to fix it, you need to know a bit about how Suspense works and why it's important. Suspense is used to manage asynchronous data fetching and code-splitting, allowing you to display a loading indicator while the data or code is being fetched. When React encounters a Suspense boundary (created using), it knows that there might be a delay in rendering, and it can handle that situation gracefully.

The error message you received is telling you that a component that was responding to synchronous input (meaning it's not supposed to be waiting for anything) encountered a suspension. This should not happen because Suspense is primarily designed to handle asynchronous operations, and you generally don't want to introduce delays in the rendering of synchronous user interactions.

💡 Here's how to fix this error:

1. `Identify the Issue` - You should identify which part of your code is causing the synchronous component to suspend. This could be due to a network request, a dynamic import of a component, or another asynchronous operation.
2. `Wrap Asynchronous Code` - Ensure that the asynchronous code, which might suspend, is wrapped within a Suspense boundary (using) and that you provide a fallback UI to display while waiting for the operation to complete.

📌 Here's an example of how to properly structure your code:

```
import React, { Suspense, lazy } from 'react';

const AsyncComponent = lazy(() => import('./AsyncComponen
t'));

function App() {
  // Synchronous code
  return (
    <div>
```

```
        <h1>Your App</h1>
        <Suspense fallback={<div>Loading...</div>}>
          <AsyncComponent />
        </Suspense>
      </div>
    );
}

export default App;
```

In this example, the AsyncComponent is loaded asynchronously, and it is wrapped within a boundary. The fallback attribute specifies what to display while the component is loading. The rest of the application, which is synchronous, doesn't get affected and will continue to respond to user input without unnecessary delays.

Suspense helps in maintaining a smooth and responsive user experience by handling asynchronous operations gracefully and ensuring that synchronous interactions are not interrupted by loading indicators.

Q) Advantages and Disadvantages of using this `code splitting pattern` ?

> Code splitting is a technique used to break down a large monolithic JavaScript bundle into smaller, more manageable pieces, which can be loaded on-demand. This pattern has several advantages and some potential disadvantages, depending on how it's implemented and the specific use case. Let's explore the advantages and disadvantages of using code splitting:

Advantages:

`Faster Initial Load Time` Smaller initial bundles result in faster load times for your web application. Users can start interacting with the application sooner because

they don't have to download unnecessary code.

`Improved Performance` - Code splitting can lead to better performance, as smaller bundles can be parsed and executed more quickly. This can reduce the time it takes to render the initial page and improve the overall responsiveness of the application.

`Optimized Resource Usage` - Code splitting helps optimize resource usage. Components or features that are rarely used may never be loaded unless needed. This conserves bandwidth and memory, making your application more efficient.

`Enhanced Caching` - Smaller bundles can benefit from browser caching. Since they are less likely to change frequently, browsers can cache them, resulting in faster subsequent visits for returning users.

`Simpler Maintenance` - Smaller bundles are easier to maintain. When you make updates to specific parts of your application, you can be more confident that you won't introduce unexpected issues in unrelated components.

`Better Mobile Performance` - On mobile devices with limited bandwidth and processing power, code splitting can significantly enhance the user experience by reducing the amount of data that needs to be loaded and processed.

Disadvantages:

`Complex Configuration` - Setting up code splitting and configuring it correctly can be complex, especially in large applications. You may need to make adjustments to your build tools and bundler settings.

`Initial Loading Delay` - When a component is loaded on-demand, there may be a slight delay the first time it is needed, which can impact user perception of your application's speed. However, this delay is usually minimal, and it's often a trade-off for the benefits of code splitting.

`Asynchronous Loading` - Handling asynchronous loading and rendering of components requires careful design to ensure a seamless user experience. You need to consider scenarios such as loading indicators and error handling.

`Route-Based Splitting` - To maximize the benefits of code splitting, you should implement it on a route or feature basis. This can lead to a more granular structure, but it may require some restructuring of your application.

`Tool and Framework Support` - Not all frameworks and libraries have built-in support for code splitting. You may need to rely on specific tools and configurations, which can vary depending on your stack.

`Testing Complexity` - Testing code-split components can be more challenging because you must ensure that they load correctly in different scenarios and that they don't introduce unexpected issues.

In summary, code splitting is a valuable technique for improving the performance and user experience of your web applications, but it comes with some complexities and trade-offs. The advantages, especially in terms of faster initial load times and optimized resource usage, often outweigh the disadvantages, which can be mitigated with careful implementation and testing.

Q) When do we and why do we `need suspense` ?

> React Suspense is a feature introduced in React to help manage asynchronous operations, such as data fetching and code splitting, in a more declarative and user-friendly manner. You need to use Suspense in your React application when you want to:

`Improve User Experience` - Suspense helps in providing a better user experience by managing the loading state of asynchronous operations. Instead of showing loading spinners or handling loading states manually, Suspense allows you to specify fallback UI components to be displayed while data is being fetched or code is being loaded.

`Optimize Performance` - Suspense, in combination with code splitting, can significantly improve the performance of your application. It allows you to load code and data only when it's needed, reducing the initial bundle size and making your application faster to load.

`Simplify Code` - Suspense simplifies your code by providing a more declarative way to handle asynchronous operations. It reduces the need for complex state management and error handling for data fetching or code splitting.

`Avoid Callback Hell` - In traditional async patterns, managing multiple asynchronous operations can lead to "callback hell" or nested promises. Suspense provides a more structured way to handle multiple asynchronous operations concurrently.

`Error Handling` - Suspense is also useful for handling errors gracefully. You can specify how to render error components or messages when an error occurs during data fetching or code splitting, making it easier to provide a clear user-facing error message.

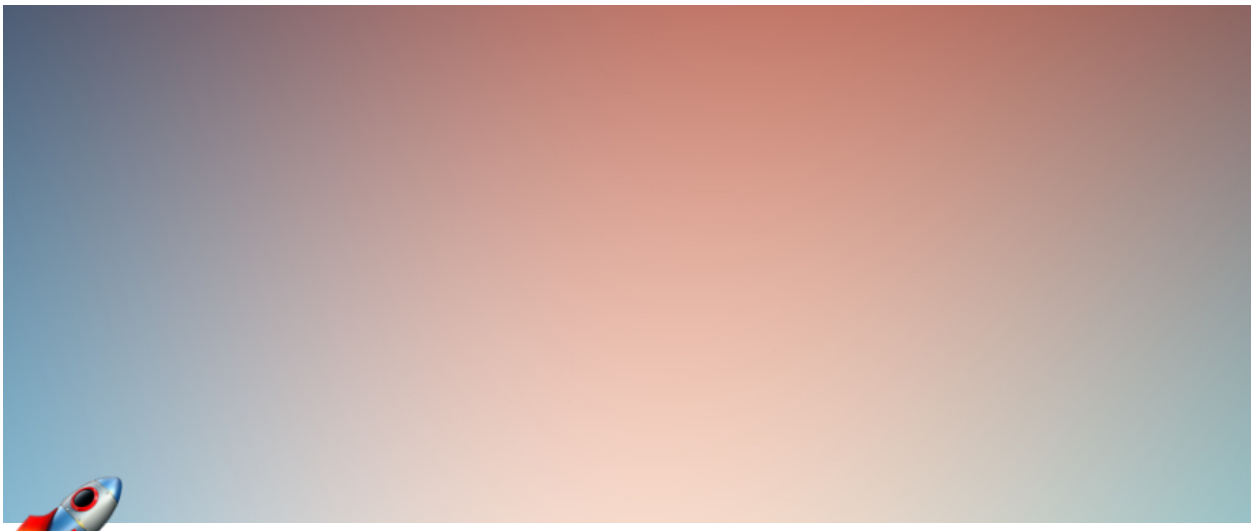
Here's a brief overview of when and why you might need Suspense in different scenarios:

`Data Fetching` - Use Suspense for data fetching when you want to make your application more responsive and provide a smooth loading experience for data-driven components. It simplifies the management of loading states and error handling.

`Code Splitting` - Use Suspense for code splitting when you want to improve your application's initial load time and performance. It allows you to load parts of your application on-demand, which can lead to faster rendering times and better resource usage. Concurrent Mode:

React Suspense is particularly valuable when using React Concurrent Mode. Concurrent Mode leverages Suspense to handle asynchronous rendering and data fetching more concurrently and efficiently.

In summary, you need to use Suspense in React when you want to create a more responsive, efficient, and user-friendly application by simplifying the handling of asynchronous operations and providing a better user experience during data fetching and code splitting.



Episode-10 | Jo Dikhta he Vo Bikta he

💡 Please make sure to follow along with the whole "**Namaste React**" series, starting from Episode-1 and continuing through each subsequent episode. The notes are designed to provide detailed explanations of each concept along with examples to ensure thorough understanding. Each episode builds upon the knowledge gained from the previous ones, so starting from the beginning will give you a comprehensive understanding of React dvelopment.

💡 I've got a quick tip for you. To get the most out of these notes, it's a good idea to watch **Episode-10** first. Understanding what "**Akshay**" shares in the video will make these notes way easier to understand.

Q) Explore all the ways of writing CSS.

Using CSS - CSS can be added to HTML documents in `3 ways` :

```
1. Inline
2. Internal
3. External
```

1 `Inline` - by using the `style attribute` inside HTML elements.

```
<h1 style="color:blue;">A Blue Heading</h1>
<p style="color:red;">A red paragraph.</p>
```

2 `Internal` - by using a `<style>` element in the section.

```
<!DOCTYPE html>
<html>
  <head>
    <style>
      body {background-color: powderblue;}
      h1   {color: blue;}
      p    {color: red;}
    </style>
  </head>
  <body>
    <h1>This is a heading</h1>
    <p>This is a paragraph.</p>
  </body>
</html>
```

3 `External` - by using a `<link>` element to link to an external CSS file.

```
<!DOCTYPE html>
<html>
<head>
  <link rel="stylesheet" href="styles.css">
</head>
```

```
<body>

<h1>This is a heading</h1>
<p>This is a paragraph.</p>

</body>
</html>
```

```
body {
  background-color: powderblue;
}
h1 {
  color: blue;
}
p {
  color: red;
}
```

Q: How do we configure `tailwindcss`?

> Configuring Tailwind CSS involves a few simple steps. Tailwind CSS is often configured using a configuration file where you can customize various settings, such as colors, fonts, breakpoints, and more. Here's a step-by-step guide:

💡 **Step 1: Create a new project (if not already done)**

Ensure you have a new or existing project where you want to use Tailwind CSS.

💡 Step 2: Install Tailwind CSS

You can install Tailwind CSS using npm or yarn. Open your terminal or command prompt and navigate to your project's root directory. Run one of the following commands:

Using npm:

```
npm install tailwindcss
```

💡 Step 3: Create a Configuration File

Create a configuration file for Tailwind CSS. You can generate a basic configuration file using the following command:

```
npx tailwindcss init
```

This command creates a tailwind.config.js file in your project's root directory.

💡 Step 4: Customize the Configuration (Optional)

Open the generated tailwind.config.js file, and you can customize various aspects of Tailwind CSS according to your project's needs. This file includes options for colors, fonts, spacing, breakpoints, and more.

For example, you can customize the colors in the tailwind.config.js file like this:

```
module.exports = {
  theme: {
    extend: {
      colors: {
        primary: '#3490dc',
```

```
        secondary: '#ffed4a',
        // ...add more custom colors as needed
      },
    },
  },
  // ...other configurations
};
```

💡 Step 5: Create CSS File

Create a CSS file where you will import Tailwind CSS and any additional styles. Typically, this file is named styles.css or similar. Import Tailwind CSS using the @import directive.

```
/* styles.css */
@import 'tailwindcss/base';
@import 'tailwindcss/components';
@import 'tailwindcss/utilities';

/* Add your custom styles here */
```

💡 Step 6: Build Your Styles

Include your CSS file in your HTML or import it in your JavaScript file if you are using a bundler like Webpack.

💡 Step 7: Use Tailwind CSS Classes in HTML

Now, you can start using Tailwind CSS classes in your HTML files to apply styles. For example:

```
<div class="bg-primary text-white p-4">
  This is a primary-colored box with white text and padding.
</div>
```

💡 **Step 8: Build Your Project**

Depending on your setup, you might need to build your project to apply the Tailwind CSS styles. If you're using a bundler like Webpack, make sure to run the appropriate build command.

For example, with npm:

```
npm run build or npm start
```

That's it! We've successfully configured and started using Tailwind CSS in your project. Remember to consult the official documentation for more detailed information and advanced configurations.

Q) In `tailwind.config.js` , what does all the keys mean (content, theme, extend, plugins)?

> In tailwind.config.js, the various keys serve different purposes and allow you to customize and configure different aspects of Tailwind CSS. Here's an overview of what each key typically represents:

💡 **1. content Key:**

`Purpose`: Specifies the files that Tailwind CSS should analyze to generate its utility classes. Usage:

```
module.exports = {
  content: [
    './src/**/*.html',
    './src/**/*.js',
    // Add more file paths as needed
  ],
  // ...other configurations
}
```

The content key helps Tailwind CSS identify which files to process and extract utility classes from. It is particularly useful when working with frameworks like React or Vue.

🔐 2. theme Key:

`Purpose`: Defines the default styles and configurations for various aspects of Tailwind CSS, such as colors, spacing, fonts, and more. Usage:

```
module.exports = {
  theme: {
    extend: {
      colors: {
        primary: '#3490dc',
        secondary: '#ffed4a',
        // ...add more custom colors
      },
    },
    // ...other theme configurations
  },
  // ...other configurations
};
```

The theme key allows you to customize default styles and extend or override the default configuration provided by Tailwind CSS. It is where you can define your

project-specific design system.

🔑 3. extend Key:

`Purpose`: Extends or overrides the default configuration provided by Tailwind CSS. Usage:

```
module.exports = {
  extend: {
    colors: {
      primary: '#3490dc',
      secondary: '#ffed4a',
      // ...add more custom colors
    },
    // ...other extensions
  },
  // ...other configurations
};
```

The extend key is often used to add new styles or extend existing ones. It is especially useful for adding project-specific utility classes or modifying existing ones.

📌 4. plugins Key:

`Purpose`: Allows you to use or define custom plugins to extend or modify Tailwind CSS functionality. Usage:

```
module.exports = {
  plugins: [
    require('@tailwindcss/forms'), // Example plugin
    // ...add more plugins as needed
  ],
```

```
  // ...other configurations
};
```

The plugins key lets you incorporate third-party plugins or create your own custom plugins. Plugins can add new features, styles, or utilities to Tailwind CSS.

These keys provide a flexible and powerful way to configure Tailwind CSS based on your project's requirements. They allow you to control the content, define styles, extend default configurations, and enhance functionality through plugins. Remember to consult the official Tailwind CSS documentation for detailed information on each configuration option and best practices.

Q) Why do we have `.postcssrc` file?

> The `.postcssrc` file, often named postcss.config.js, is a configuration file for PostCSS. PostCSS is a tool for transforming styles with JavaScript plugins, and it is commonly used in conjunction with build tools like webpack or parcel for processing and optimizing CSS.

Here are the primary reasons why you might have a .postcssrc file:

- **Plugin Configuration:**

The main purpose of the .postcssrc file is to configure the plugins that PostCSS should use during the CSS transformation process. These plugins can handle tasks such as autoprefixing, minification, and syntax enhancements.

- **Custom Configuration:**

You may need a .postcssrc file if you want to customize the behavior of PostCSS beyond the default settings provided by the build tool (e.g., webpack). This allows you to have fine-grained control over the PostCSS transformations.

- **Presets and Options:**

PostCSS plugins often come with various options and presets that you can configure based on your project's needs. The .postcssrc file is a convenient place

to define these options and presets.

- **Maintainability:**

Separating the PostCSS configuration into its own file makes the build configuration more maintainable and organized. It allows you to centralize PostCSS-related settings and keep them distinct from other build tool configurations.

- **Sharing Configurations:**

Having a dedicated configuration file makes it easier to share and reuse PostCSS configurations across different projects. It can be particularly useful in larger development ecosystems where consistent styles and build processes are desired.

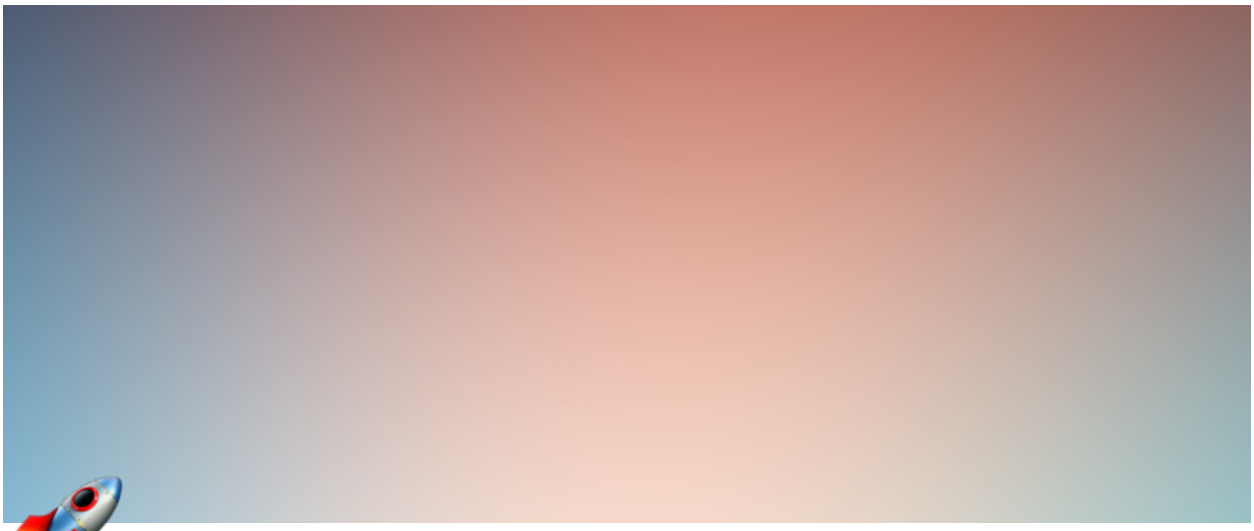
Example .postcssrc file:

```
// postcss.config.js

module.exports = {
  plugins: {
    // Example plugins with options
    'autoprefixer': {},
    'postcss-preset-env': {
      stage: 3,
      features: {
        'nesting-rules': true,
      },
    },
    'cssnano': {
      preset: 'default',
    },
  },
};
```

In this example, the .postcssrc file configures three PostCSS plugins: autoprefixer for adding vendor prefixes, postcss-preset-env for enabling future CSS features,

and cssnano for minification. The options provided for each plugin customize their behavior.

Remember that the specific configuration options and plugins you include in your .postcssrc file will depend on your project's requirements and the PostCSS features we want to leverage.



Episode-11 | Data is The New Oil

💡 Please make sure to follow along with the whole "**Namaste React**" series, starting from Episode-1 and continuing through each subsequent episode. The notes are designed to provide detailed explanations of each concept along with examples to ensure thorough understanding. Each episode builds upon the knowledge gained from the previous ones, so starting from the beginning will give you a comprehensive understanding of React dvelopment.

💡 I've got a quick tip for you. To get the most out of these notes, it's a good idea to watch **Episode-11** first. Understanding what "**Akshay**" shares in the video will make these notes way easier to understand.

Q) What is `prop drilling` ?

> In React, `prop drilling` refers to the process of `passing down props (short for properties) through multiple layers of nested components`. This happens when a piece of data needs to be transferred from a higher-level component to a deeply nested child component, and it must pass through several intermediary components in between.

💡 Here's a simple example to illustrate prop drilling in React:

```
// Top-level component
function App() {
  const data = "Hello, prop drilling!";

  return (
    <div>
      <ParentComponent data={data} />
    </div>
  );
}

// Intermediate component
function ParentComponent({ data }) {
  return (
    <div>
      <ChildComponent data={data} />
    </div>
  );
}

// Deeply nested component that actually uses the data
function ChildComponent({ data }) {
```

```
  return <div>{data}</div>;
}
```

In this example, the `data` prop is passed from the App component through the `ParentComponent down to the ChildComponent`. The ParentComponent itself doesn't use the data prop; it merely passes it down. `This process of passing data through intermediate components that don't use the data is what is referred to as prop drilling`.

Prop drilling can make the code harder to maintain, especially as the application grows and the number of components in the hierarchy increases. To mitigate this, developers often use other state management solutions, like the `React Context API`, `Redux`, or `other state management libraries`, to avoid passing props through multiple layers of components. These alternatives provide a centralized way to manage and access state without the need for prop drilling.

Q) What is `lifting the state up`?

> `Lifting state up` in React refers to the practice of `moving the state from a lower-level (child) component to a higher-level (parent or common ancestor) component in the component tree`. This is done to share and manage state across multiple components.

When a child component needs access to certain data or needs to modify the data, instead of keeping that data and the corresponding state management solely within the child component, we move the state to a shared ancestor component. By doing so, the parent component becomes the source of truth for the state, and it can pass down the necessary data and functions as props to its child components.

💡 Here's a simple example to illustrate `lifting state up`:

```
// Parent component
class ParentComponent extends React.Component {
```

```
  constructor(props) {
    super(props);
    this.state = {
      count: 0,
    };
  }

  incrementCount = () => {
    this.setState((prevState) => ({
      count: prevState.count + 1,
    }));
  };

  render() {
    return (
      <div>
        <p>Count: {this.state.count}</p>
        <ChildComponent count={this.state.count} onIncrement=
{this.incrementCount} />
      </div>
    );
  }
}

// Child component
function ChildComponent({ count, onIncrement }) {
  return (
    <div>
      <p>Child Count: {count}</p>
      <button onClick={onIncrement}>Increment</button>
    </div>
  );
}
```

In this example, the ParentComponent holds the state (count), and it passes both the state value (count) and a function (onIncrement) down to the ChildComponent

as props. The child component can then display the count and trigger an increment when the button is clicked.

By lifting the state up to a common ancestor, you centralize the state management, making it easier to control and share state among components. This pattern is especially useful in larger React applications where multiple components need access to the same data or where the state needs to be synchronized across different parts of the application.

🔗*Lifting Stateup*

Q) What are `Context Provider` and `Context Consumer` ?

> In React, the `Context API` provides a `way to pass data through the component tree without having to pass props manually at every level`. The two main components associated with the `Context API are the Context Provider and Context Consumer`.

> `Context Provider`: The Context Provider is a `component that allows its children to subscribe to a context's changes`. It accepts a value prop, which is the data that will be shared with the components that are descendants of this provider. The Provider component is created using `React.createContext()` and then rendered as part of the component tree. It establishes the context and provides the data to its descendants.

💡 Here's an example:

```
// Creating a context
const MyContext = React.createContext();
```

```
// Parent component serving as the provider
class MyProvider extends React.Component {
  state = {
    data: "Hello from Context!",
  };

  render() {
    return (
      <MyContext.Provider value={this.state.data}>
        {this.props.children}
      </MyContext.Provider>
    );
  }
}
```

`Context Consumer`: The Context Consumer is a component that subscribes to the changes in the context provided by its nearest Context Provider ancestor. It allows components to access the context data without the need for prop drilling. The Consumer component is used within the JSX of a component to consume the context data. It takes a function as its child, and that function receives the current context value as an argument. Here's an example:

```
// Child component consuming the context
class MyConsumerComponent extends React.Component {
  render() {
    return (
      <MyContext.Consumer>
        {(contextData) => (
          <p>{contextData}</p>
        )}
      </MyContext.Consumer>
    );
  }
}
```

By using the Context Provider and Context Consumer, you can avoid prop drilling and make it easier to share global or shared state across different parts of your React application. This is particularly useful when passing data to deeply nested components without explicitly passing the data through each intermediate component.

Q) If we `don't pass a value to the provider does it take the default value` ?

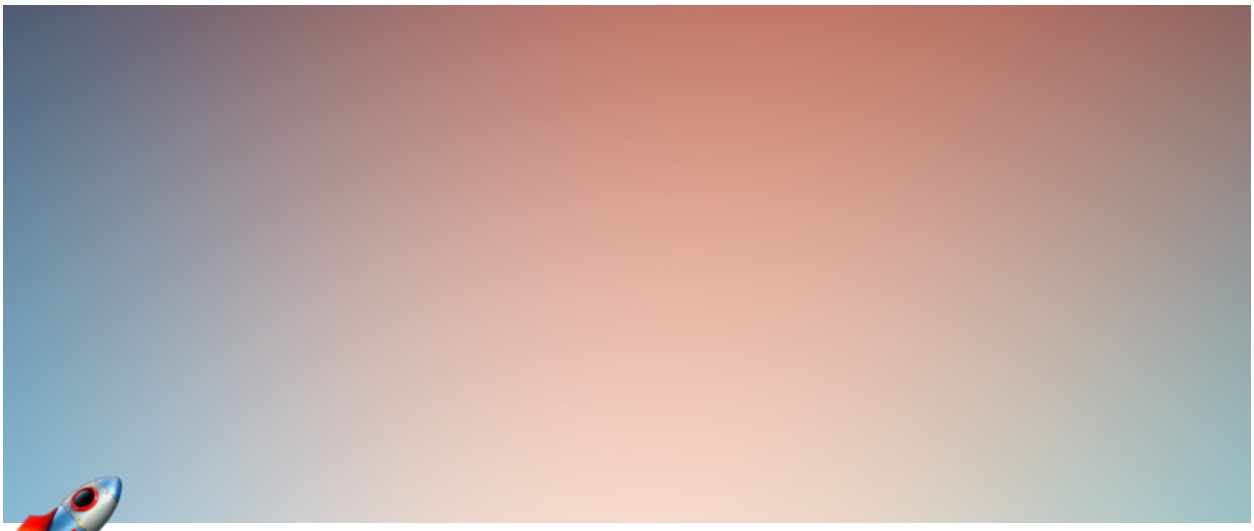
> Yes, If we don't pass a value to the Provider in React's Context API, `it does use the default value specified when creating the context using React.createContext(defaultValue)`.

💡 Here's the corrected explanation:

```
// Creating a context with a default value
const MyContext = React.createContext("Default Value");

// Parent component serving as the provider without providing
a value
class MyProvider extends React.Component {
  render() {
    return (
    <MyContext.Provider>
        {this.props.children}
      </MyContext.Provider>
    );
  }
}
```

In this example, if we don't provide a value to the `MyContext.Provider`, it will use the default value ("Default Value" in this case) specified during the creation of the

context. Any component that consumes this context using `MyContext.Consumer` will receive the default value if there is no Provider higher up the tree providing a different value.



Episode-12 | Let’s Build Our Store

💡 Please make sure to follow along with the whole "**Namaste React**" series, starting from Episode-1 and continuing through each subsequent episode. The notes are designed to provide detailed explanations of each concept along with examples to ensure thorough understanding. Each episode builds upon the knowledge gained from the previous ones, so starting from the beginning will give you a comprehensive understanding of React dvelopment.

💡 I've got a quick tip for you. To get the most out of these notes, it's a good idea to watch **Episode-12** first. Understanding what "**Akshay**" shares in the video will make these notes way easier to understand.

Q) `useContext` vs `Redux`

> `useContext` and `Redux` are both tools used for state management in React applications, but they serve different purposes and have different use cases. Let's explore the key differences between useContext and Redux:

💡 **useContext:**

`Scope`: `useContext` is part of the React core and is used for managing state within the component tree. It provides a way to access the value of a context directly within a component and its descendants. It's typically used for smaller-scale state management needs within a component or a small section of the application.

`Complexity`: useContext is simpler and more lightweight compared to Redux. It's a part of the React library and doesn't introduce additional concepts or boilerplate code.

`Component Coupling`: State managed with useContext is local to the component or a subtree of components where the context is provided. This can lead to more isolated and less globally shared state.

`Integration`: It's seamlessly integrated into React and works well with the component lifecycle. You can create and consume contexts within functional components using the useContext hook.

💡 **Redux:**

`Scope`: Redux is a state management library that provides a global state container for the entire application. It allows you to manage the application state in a predictable and centralized manner.

`Complexity`: Redux introduces a set of concepts, such as actions, reducers, and a store. This can make it more complex compared to using useContext for local state management. However, it becomes valuable in larger and more complex applications.

`Component Coupling`: State managed with Redux is global, which means any component can connect to and access the state. This can be advantageous for sharing state across different parts of the application.

`Integration`: Redux needs to be integrated separately into a React application. You need to create actions, reducers, and a store. Components interact with the global state using the connect function or hooks like useSelector and useDispatch.

> 💡 **Use Cases:**

`Use useContext When`: We have smaller-scale state management needs within a component or a local subtree. We want a lightweight solution without introducing additional complexity. Our state doesn't need to be shared extensively across different parts of the application.

`Use Redux When`: We have a complex application with a large state that needs to be shared across many components. We want a predictable state management pattern with a unidirectional data flow. We need middleware for advanced features like asynchronous actions.

Choose `useContext for simpler and local state management within components or small sections of our application`. Choose `Redux for more complex applications where we need a global state that can be easily shared across different components`.

Q) Advantages of using `Redux Toolkit over Redux`?

`Redux Toolkit` is a set of utility functions and abstractions that simplifies and streamlines the process of working with Redux. It is designed to address some of the common pain points and boilerplate associated with using plain Redux. Here are some advantages of using Redux Toolkit over plain Redux:

`Less Boilerplate Code`: Redux Toolkit helps us write less code. It provides shortcuts that save us from typing a lot of repetitive and verbose code, making our Redux logic cleaner and more concise.

`Easier Async Operations`: If our app deals with things like fetching data from a server, Redux Toolkit makes it simpler. It has a tool called createAsyncThunk that handles async actions in a way that's easy to understand and use.

`Simpler Store Setup`: Setting up your Redux store is easier with Redux Toolkit. It has a function called configureStore that simplifies the process, and it comes with sensible defaults, so you don't have to configure everything from scratch.

`Built-in DevTools Support`: If you use Redux DevTools for debugging, Redux Toolkit has built-in support. Enabling it is as easy as adding one line of code when setting up your store.

`Encourages Best Practices`: Redux Toolkit is recommended by the official Redux documentation. It encourages you to follow best practices in Redux development, making sure your code is more maintainable and aligns with industry standards.

`Handles Immutability for You`: Working with immutable data (making sure you don't accidentally change your data) is usually a bit tricky. Redux Toolkit uses a library called Immer to handle this behind the scenes, so you can write more straightforward and readable code.

`Backward Compatibility`: If you already have a Redux app, you can slowly transition to Redux Toolkit without rewriting everything. It's designed to be compatible with your existing Redux code.

`Faster Development`: With Redux Toolkit, you can get things done more quickly. You spend less time setting up and configuring Redux and more time focusing on building features for your app.

In simple terms, Redux Toolkit is like a set of tools that makes working with Redux easier. It simplifies common tasks, reduces the amount of code you need to write, and encourages good coding practices. If you're starting a new project or thinking about improving an existing one, Redux Toolkit can save you time and make your life as a developer more enjoyable.

Q) Explain `Dispatcher` ?

> In Redux, a `dispatcher` is not a standalone concept; instead, it's a term often used to refer to a function called dispatch. The dispatch function is a key part of the Redux store, and it plays a crucial role in the Redux data flow.

💡 Here's a breakdown of the dispatch function and its role in Redux:

1 `Dispatch Function`: The dispatch function is provided by the Redux store. We use it to send actions to the store. An action is a plain JavaScript object that describes what should change in the application's state.

2 `Usage`: When we want to update the state in our Redux store, we create an action and dispatch it using the dispatch function.

```
const myAction = { type: 'INCREMENT' };
store.dispatch(myAction);
```

- Here, the `INCREMENT` action is an example. The dispatch function is responsible for sending this action to the Redux store.

3 `Middleware`: The dispatch function is also a crucial point in the Redux middleware chain. Middleware can intercept actions before they reach the reducer or modify actions on the way out. Middleware functions receive the dispatch function, allowing them to either pass the action along or stop it.

4 `Redux Store`: The dispatch function is a core method provided by the Redux store. When an action is dispatched, the store passes the action through its reducer, which is a function that specifies how the state should change in response to the action.

5 `Asynchronous Actions`: Redux supports asynchronous actions using middleware like redux-thunk or redux-saga. The dispatch function allows you to handle asynchronous operations by dispatching actions inside functions (thunks) and handling those actions asynchronously.

📌 Here's an example of how you might use dispatch in a React component:

```
import { useDispatch } from 'react-redux';

const MyComponent = () => {
```

```
  const dispatch = useDispatch();

  const handleButtonClick = () => {
    // Dispatching an action to increment the count
    dispatch({ type: 'INCREMENT' });
  };

  return (
    <button onClick={handleButtonClick}>
      Increment Count
    </button>
  );
};
```

In this example, the `useDispatch` hook from react-redux gives us access to the dispatch function, which we then use to send an action to the Redux store when the button is clicked. This action will be processed by the reducer, updating the state accordingly.

Q) Explain `Reducer` ?

> **In Redux Toolkit, the `createSlice function is commonly used to create reducers`. It simplifies the process of defining actions and the corresponding reducer logic, reducing boilerplate code. Let's break down the key concepts related to creating reducers with createSlice in Redux Toolkit :**

`Creating a Slice`: Instead of creating a standalone reducer function, you use createSlice to define a "slice" of your Redux store. A slice includes actions, a reducer, and the initial state.

```
import { createSlice } from '@reduxjs/toolkit';

const counterSlice = createSlice({
```

```
  name: 'counter',
  initialState: { value: 0 },
  reducers: {
    increment: (state) => {
      state.value += 1;
    },
    decrement: (state) => {
      state.value -= 1;
    },
  },
});

// Extracting actions and reducer from the slice
export const { increment, decrement } = counterSlice.actions;
export default counterSlice.reducer;
```

2 `Automatically Generated Action Creators`: `createSlice` automatically generates action creators for each reducer function. In the example above, increment and decrement are automatically created and exported for use in your components.

3 `Immutability with immer`: Redux Toolkit uses the immer library internally to handle immutability. This means you can write reducer logic that appears to directly modify the state, but immer ensures it produces a new state without mutating the original.

```
reducers: {
  increment: (state) => {
    state.value += 1;  // Immer takes care of creating a new
state
  },
},
```

4 `Reducer Function`: The createSlice function returns an object that includes a reducer property. This reducer is a function that you can use in your store's configuration.

```
const rootReducer = combineReducers({
  counter: counterSlice.reducer,
  // ... other reducers
});
```

5 `Initial State`: The initialState property in createSlice defines the initial state of your slice. This is the starting point for your state before any actions are dispatched.

6 `Reducer Logic`: The logic inside each reducer function specifies how the state should change in response to the associated action. In the example, the increment and decrement reducers modify the value property of the state.

7 `Simplifying Reducer Composition`: With createSlice, we can easily compose reducers using combineReducers or by directly adding the slice's reducer to the root reducer. This simplifies the overall reducer composition in our application.

Using createSlice in Redux Toolkit streamlines the process of defining reducers, actions, and initial states, making your Redux code more concise and readable. It encourages best practices, such as immutability and simplicity, while reducing the boilerplate traditionally associated with Redux.

Q) Explain `Slice` ?

> In Redux Toolkit, a `slice` is a collection of Redux-related code, including reducer logic and actions, that corresponds to a specific piece of the application state. Slices are created using the createSlice utility function provided by Redux Toolkit. The primary purpose of slices is to encapsulate the logic related to a specific part of the state, making the code more modular and easier to manage.

🛠️ Here's a breakdown of key concepts related to slices in Redux Toolkit:

Creating a Slice: The createSlice function takes an options object with the following properties:

1 `name (string)` : A string that identifies the slice. This is used as the prefix for the generated action types.

```
import { createSlice } from '@reduxjs/toolkit';

const mySlice = createSlice({
  name: 'mySlice',
  initialState: { /* ... */ },
  reducers: {
    // ...reducers
  },
});
```

2 `initialState (any)`: The initial state value for the slice. This is the starting point for your state before any actions are dispatched. 3 `reducers (object)`: An object where each key-value pair represents a reducer function. The keys are the names of the actions, and the values are the corresponding reducer logic.

```
const mySlice = createSlice({
  initialState: { /* ... */ },
  reducers: {
    increment: (state) => {
      state.value += 1;
    },
    decrement: (state) => {
      state.value -= 1;
    },
  },
});
```

Output: The createSlice function returns an object with the following properties:

`name (string)`: The name of the slice. `reducer (function)`: The reducer function generated based on the provided reducers. This is the function you use in your store configuration. `actions (object)`: An object containing the action creators for each defined reducer. These action creators can be directly used to dispatch actions.

```
const { increment, decrement } = mySlice.actions;
```

Using a Slice: Once we've created a slice, you can use its reducer and actions in your Redux store configuration and in your React components.

1 `Configuring the Store`: We can include the generated reducer in your store configuration.

```
import { configureStore } from '@reduxjs/toolkit';
import mySliceReducer from './path/to/mySlice';

const store = configureStore({
  reducer: {
    mySlice: mySliceReducer,
    // ...other reducers
  },
});
```

2 `Dispatching Actions`: In our React components, we can use the generated action creators to dispatch actions.

```
import { useDispatch } from 'react-redux';
import { increment } from './path/to/mySlice';

const MyComponent = () => {
  const dispatch = useDispatch();

  const handleIncrement = () => {
    dispatch(increment());
  };
```

```
  // ... rest of the component logic
};
```

Using `slices in Redux Toolkit promotes a modular and organized approach to state management`. Each slice encapsulates the logic related to a specific part of the state, making it easier to understand, maintain, and scale your Redux code.

Q) Explain `Selector` ?

> In Redux Toolkit, a `selector` is a function that extracts specific pieces of data from the Redux store. It allows you to compute derived data from the store state and efficiently access specific parts of the state tree. Selectors play a crucial role in managing the state in a clean and efficient way.

Redux Toolkit provides the createSlice and createAsyncThunk utilities along with the createSelector function from the reselect library to help manage selectors easily.

📣 Here's an explanation of how selectors work in Redux Toolkit:

1 `Defining Selectors with createSlice`: When we create a slice using createSlice, we can include selectors in the extraReducers field. These selectors can compute and return specific pieces of data from the state.

```
import { createSlice } from '@reduxjs/toolkit';

const mySlice = createSlice({
  name: 'mySlice',
  initialState: { data: [] },
  reducers: {
    // ...reducers
```

```
  },
  extraReducers: (builder) => {
    builder
      .addCase(otherSliceAction, (state, action) => {
        // logic for handling other slice's action
      })
      .addDefaultCase((state, action) => {
        // default logic for handling actions not handled by
this slice
      });
  },
  selectors: (state) => ({
    // selector functions here
    selectData: () => state.data,
    selectFilteredData: (filter) => state.data.filter(item =>
item.includes(filter)),
  }),
});

export const { selectData, selectFilteredData } = mySlice.sel
ectors;
```

2 `Using Reselect with createSlice`: If we need more advanced memoization and composition of selectors, you can use the createSlice function along with the reselect library.

```
import { createSlice, createSelector } from '@reduxjs/toolki
t';

const mySlice = createSlice({
  // ... other options
  selectors: {
    selectData: (state) => state.data,
    selectFilteredData: createSelector(
      (state) => state.data,
      (_, filter) => filter,
```

```
      (data, filter) => data.filter(item => item.includes(fil
ter))
    ),
  },
});

export const { selectData, selectFilteredData } = mySlice.sel
ectors;
```

3 `Using Selectors in Components`: Once we've defined selectors, we can use them in your React components using the useSelector hook from the react-redux library. This hook allows you to efficiently extract and subscribe to parts of the Redux store.

```
import { useSelector } from 'react-redux';
import { selectData, selectFilteredData } from './mySlice';

const MyComponent = () => {
  const data = useSelector(selectData);
  const filteredData = useSelector(state => selectFilteredDat
a(state, 'someFilter'));

  // ... rest of the component logic
};
```

Selectors help keep your state management logic clean and efficient by allowing us to centralize the computation of derived data from the Redux store. They contribute to better organization, improved performance, and easier maintenance of our Redux code.

Q) Explain `createSlice` and the configuration it takes?

> `createSlice` is a utility function provided by Redux Toolkit that simplifies the `process of creating Redux slices`. A `Redux slice is a piece of the Redux store that includes a set of actions, a reducer, and an initial state`. The createSlice function helps reduce boilerplate code associated with defining actions and the reducer for a specific slice of your Redux store.

Here's an explanation of the configuration options that createSlice takes:

📣 **Syntax:**

```
createSlice(options)
```

📣 **Configuration Options:**

1 `name (string)`: A string that identifies the slice. This is used as the prefix for the generated action types.

```
const mySlice = createSlice({
  name: 'mySlice',
  // ... other options
});
```

2 `initialState (any)`: The initial state value for the slice. This is the starting point for your state before any actions are dispatched.

```
const mySlice = createSlice({
  initialState: { value: 0 },
  // ... other options
});
```

3 `reducers (object)`: An object where each key-value pair represents a reducer function. The keys are the names of the actions, and the values are the corresponding reducer logic.

```
const mySlice = createSlice({
  reducers: {
    increment: (state) => {
      state.value += 1;
    },
    decrement: (state) => {
      state.value -= 1;
    },
  },
  // ... other options
});
```

4 `extraReducers (builder callback)`: A callback function that allows you to define additional reducers outside of the reducers field. It is called with a builder object that provides methods for adding reducers based on other action types.

```
const mySlice = createSlice({
  extraReducers: (builder) => {
    builder
      .addCase(otherSliceAction, (state, action) => {
        // logic for handling other slice's action
      })
      .addDefaultCase((state, action) => {
        // default logic for handling actions not handled by
this slice
      });
  },
  // ... other options
});
```

5 `slice (string)`: An optional string that specifies a slice of the state to be used with the createAsyncThunk utility. This is useful when working with asynchronous

actions.

```
const mySlice = createSlice({
  slice: 'myAsyncSlice',
  // ... other options
});
```

6 `extraReducers (object)`: An alternative way to define extra reducers using an object directly. Each key represents an action type, and the value is the corresponding reducer function.

```
const mySlice = createSlice({
  extraReducers: {
    [otherSliceAction.type]: (state, action) => {
      // logic for handling other slice's action
    },
    // ... additional action types
  },
  // ... other options
});
```

 **Output :**

The createSlice function returns an object with the following properties:

`name (string)`: The name of the slice. `reducer (function)`: The reducer function generated based on the provided reducers and extraReducers. This is the function you use in your store configuration. `actions (object)`: An object containing the action creators for each defined reducer. These action creators can be directly used to dispatch actions.

```
const { increment, decrement } = mySlice.actions;
```

These are the main configuration options for createSlice in Redux Toolkit. It provides a convenient way to define actions, reducers, and initial states for slices of our Redux store, reducing the amount of boilerplate code and promoting best practices.

Time For Test! (Namaste-React)

💡 Please make sure to follow along with the whole ***"Namaste React"*** series, starting from Episode-1 and continuing through each subsequent episode. The notes are designed to provide detailed explanations of each concept along with examples to ensure thorough understanding. Each episode builds upon the knowledge gained from the previous ones, so starting from the beginning will give you a comprehensive understanding of React development.

💡 I've got a quick tip for you. To get the most out of these notes, it's a good idea to watch **Episode-13** first. Understanding the video will make these notes way easier to understand.

> **In this episode, we are going to learn about**
>
> - What are the testcases and how to write
> - Types and importance of testing

👉 Before we start to learn today's episode:

There are many types of testing like QA and developer. But we will learn about developer testing because it's a huge domain in itself

Part-1

Types of testing:

1. Manual testing: Testing the functionality that we have developed. E.g → we have developed a search bar, manual testing is checking the search bar manually by searching the query.

> 💡 This is not a very efficient way because we can’t test every new feature in a big application. A single line can introduce bugs in our whole app because multiple components are connected to each other.

2. Automatic testing: We can write the test cases for testing the functionality. It includes

a. *Unit testing* : Write test cases for the specific part (isolated components)

b. *Integration testing*: writing testcases for the components that are connected like menu page and cart page are connected.

c. *End-to-end testing*: writing testcases from user enters into the website to user leaves the website

Install libraries:

📢 NOTE: If you are using create-react-app or vite. please ignore these installation steps because these packages already include the testing library

1. React Testing library: It is provided by react that allows to test react components

```
npm i -D @testing-library/react
```

2. jest: React testing library uses jest so we need to install it.

```
npm i -D jest
```

Since we are using Babel as a bundler so we need to install some extra libraries.

For more details check the official website: https://jestjs.io/ and https://babeljs.io/docs/usage

3. install extra babel libraries

```
npm install --save-dev @babel/core @babel/cli @babel/preset-env
```

4. create babel.config.js

```
const presets = [
  [
    "@babel/preset-env",
    {
      targets: {
        edge: "17",
        firefox: "60",
        chrome: "67",
        safari: "11.1",
      },
      useBuiltIns: "usage",
      corejs: "3.6.4",
    },
  ],
];

module.exports = { presets };
```

📢 NOTE: If you are following this series, then you must know parcel is using Babel. We just added babel.config file but the parcel also has a babel configuration behind it. that creates a conflict. To solve this we have to disable the parcel config. Create a .parcel.rc file

```
// .parcel.rc
{
  "extends": "@parcel/config-default",
```

```
  "transformers": {
    "*.{js,mjs,jsx,cjs,ts,tsx}": [
      "@parcel/transformer-js",
      "@parcel/transformer-react-refresh-wrap"
    ]
  }
}
```

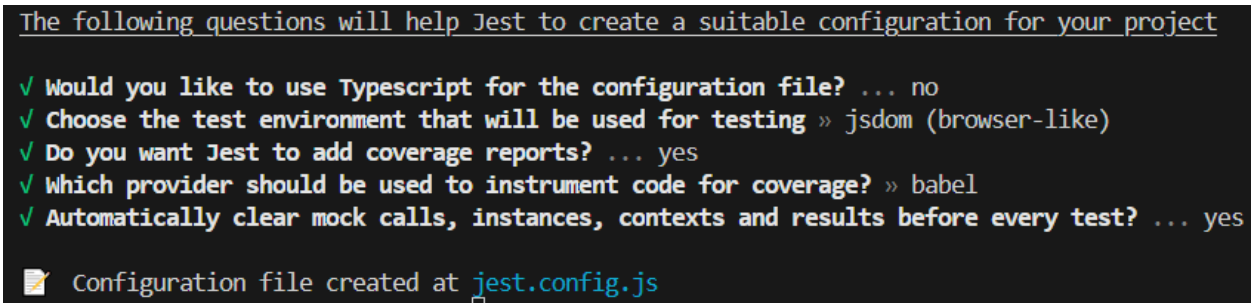
To check that you have installed successfully without an error. Try to run the testcase

`npm run test →` will give no test case found

5. Let’s configure the jest

```
npx jest --init // executing the jest package
```

After running this command, you have to select some options



💡 We are using jsdom as a test environment. When we run test cases, there is no browser or server. For running the test cases we need an environment which is jsdom

6. Install jsdom environment

```
npm install --save-dev jest-environment-jsdom
```

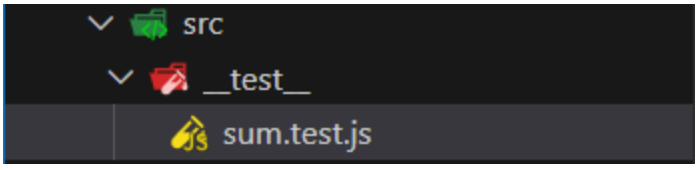
Start writing testcases:

Let’s start with writing testcases for simple function that returns the sum of two numbers

1. create a file for which we need to write the test case

```
//sum.js
export const sum  = (a,b) => {
        return a + b;
}
```

2. create a folder `__test__` and create a file in this folder `sum.test.js`



3. Let’s write out first test case

```
import {sum} from "../components/sum"

test("Function should calculate the sum of two numbers",() => {
```

```
        const result = sum(3,4) // call the function
        expect(result).toBe(7)  // assertion provided by jest.
}

// test will take two arg name and callback function
```

4. Run the test case

```
PS D:\Namaste_React\Namaste-food> npm run test
```

I hope now you have the overview that how testing works. So now let’s write the original test case for our project

Unit test:

Let’s write a test case to check the component is loading or not. To check any component loads or not, we need to check in the jsdom.

```
test('check component loads or not',() => {
        render(<Contactus/>   // render the component that want

    const heading = screen.getByRole("heading")  // check the sp

        expect("heading").toBeInTheDocument()   // check heading
})
```

📢 Note: If you are using parcel, you will encounter an error. The error is we haven’t enable JSX yet. So, we are not able to render the compoent that uses JSX. Let’s enable it

To enable JSX in testing environment:

a. install babel: `npm i -D @babel/preset-react`

b. set babel config:

📢 Note: Install one more library to use the dom functions `npm i -D @testing-library/jest-dom.`

Now, run the testcase. It will pass

We found the heading using “getByRole”. But we can also find using different functions like

```
const button = screen.getByText("submit") // It will find the te

expect(button).toBeInTheDocument()
```

💡 We can use `it` instead of `test` keyword

To write multiple testcases

```
describe('To test the header component',() => {

        it("Should load the header component",() => {})

        it("Should include the button",() => {})
}
```

💡 After creating testcase, git will show many files changed. We don’t need to upload the coverage folder to the github. So, add `coverage` to the `.gitignore` file

Let’s make one more test case for testing `Header` to check button is rendered or not

```
// import render and header
it("should test header compoenent",() => {
        render(<Header/>)
}
```

We will get a few Errors because we are testing the isolated component.

📢 Note: We used redux store in the header (useSelector) but the store is provided to the app component. To test the Header component, we need to provide the store to Header component as well

📢 Note: We used the Link component provided by react-router-dom. But we have provided router to the App component. So, To test the Header we have to provide the router to the Header component

```
gourav garg, 4 months ago | 1 author (gourav garg)
import { fireEvent, render, screen } from "@testing-library/react"
import Header from "../components/Header"
import { Provider } from "react-redux"  4.6k (gzipped: 1.9k)
import appStore from "../utils/appStore"
import { BrowserRouter } from "react-router-dom"  5.1k (gzipped: 2.3k)
import "@testing-library/jest-dom"

it("should render header component with a login button", () => {
    render(
        <BrowserRouter>
            <Provider store={appStore}>
                <Header/>
            </Provider>
        </BrowserRouter>
    );
    // If there are multiple buttons then can find using name showing on the button
    const LoginButton = screen.getByRole("button",{name:"Login"});
    // const LoginButton = screen.getByText("Login")
    expect(LoginButton).toBeInTheDocument();
})
```

To check the button click, you can use the *fireEvent* which behaves like an onclick method

---

Integration Testing:

💡 To run the test automatically, make a script in the package.json like “watch-test”:”jest --watch” . Now we can simple run npm run watch-test.

Let’s write the test case for the Search component. It should show the card in the body when we search some restaurant

Fetch the body component first (because search bar is in body)

```
it("should show the search button",() => {
        render(<Body/>)     // will throw an error
}
```

📢 Note: Error generated because the Body component uses the fetch function which is provided by the browser. But we are using jsdom. So we need to make a mock fetch function with mock data that will replace the original fetch function.

For creating fetch function, we need to create mock data for restaurant list. So, create a new file Mock_Data and store the data by coping the restaurant list from the api.

Create our fetch function similar to browser fetch function

```
// import mock Data
global.fetch = jest.fn(() => {
        return Promise.resolve({          //  fetch function retu
                json: () => {                 // json the promise
```

```
                        return Promise.resolve(Mock_Data)   // r
                }
        })
}
// Mock_Data will be returned in the end from the fetch functio

it("should show the search button",() => {
        render(<Body/>)    // will throw an error
}
```

Run this test, we will get the warning

> 💡 When we use the async operation, we should wrap our component inside act function. It will return a promise

Also provide the router to the component because we are using the `<Link/>`

```
import {act} from "react-dom/test-utils"
// import mock Data
global.fetch = jest.fn(() => {
        return Promise.resolve({         //  fetch function retu
                json: () => {                // json the promise
                        return Promise.resolve(Mock_Data)   // r
                }
        })
}
// Mock_Data will be returned in the end from the fetch functio

it("should show the search button",async() => {
        await act(async() => {
                render(
                    <BrowserRouter>    // because we are using L
                            <Body/>
```

```
                    </BrowserRouter>
                )
        }

        const searchBtn = screen.getByRole("Button",{name: "Sea
        expect(searchBtn).toBeInTheDocument()   // check the bu
}
```

Now, all testcase will pass successfully.

> 💡 If you don’t want to find using getByRole, We can also use getByTestId. It will always work

To use getByTestId, give the testid to the element

```
<input data-testid="searchinput" type="text"/>
```

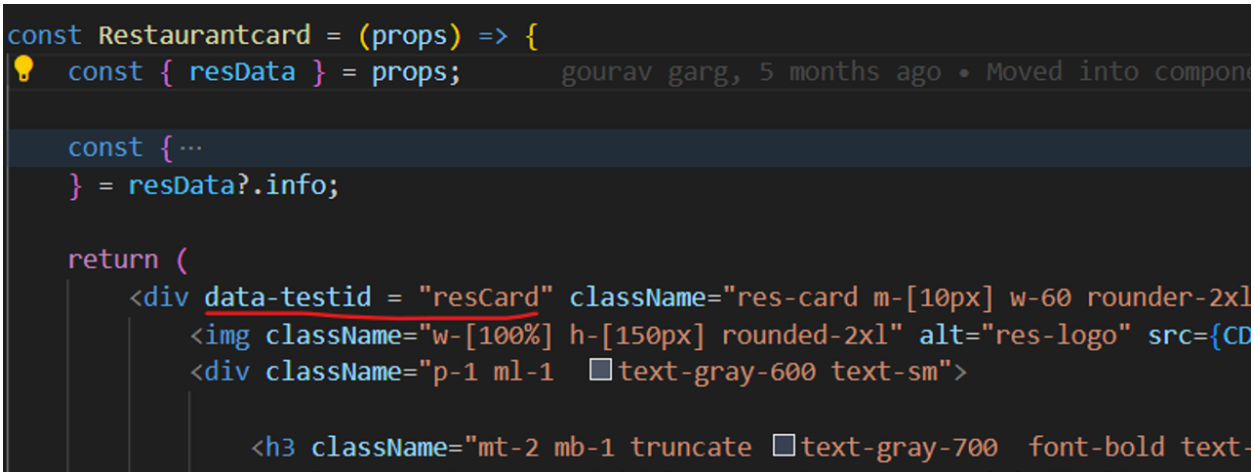
```
const searchInput  = search.getByTestId("searchInput")
```

Now, test the input to get the user query to give the restaurant data.

```
const searchInput  = screen.getByTestId("searchInput")
// fireevent is provided by jest which is used to perform the ev
fireEvent.change(searchInput,{target:{value:"burger"}})    // cl

fireEvent.click(SearchButton) // click the search button
```

Now, find the restaurant card rendered on the screen after clicking the search button

💡 To get the restaurant data div, give the testid




```
const searchCards = screen.getAllByTestId("resCard")  // get the
expect(searchCards.length).toBe(2)  // expect the count that sh
```

Now, all test case will pass.

💡 If we want to write something after and before all the testcases or each testcases. jest provide functions

Final code for Integration testing

```
import { fireEvent, render, screen} from "@testing-library/react
import Body from "../components/Body"
import { BrowserRouter } from "react-router-dom"
import MockResList from "../__tests__/mocks/MockResList.json"
import { act } from "react-dom/test-utils"

global.fetch = jest.fn(() => {
    return Promise.resolve({
        json:() => {
            return Promise.resolve(MockResList)
```

```
        }
    })
})
// describe is used to club multiple testcases
describe("",() =>{
        // to print before all testcases
    beforeAll(() => {
        console.log("Before testcase")
    })
        // to print before each testcase
    beforeEach(() => {
        console.log("Before Each")
    })
    // to print after all testcases
    afterAll(() => {
        console.log("After testcases")
    })
    // to print after each testcase
    afterEach(() => {
        console.log("After each")
    })

    it("Should render body component with search feature",async
        await act(async() =>
            render(
            <BrowserRouter>
                <Body/>
            </BrowserRouter>
        ))
    })


    it("Should search ResList for burger text input",async() =>
        await act(async() =>
            render(
            <BrowserRouter>
```

```
                <Body/>
            </BrowserRouter>
        ))

        const TotalCards = screen.getAllByTestId("resCard")
        expect(TotalCards.length).toBe(18)

        const SearchButton = screen.getByRole("button",{name:"Se

        const searchInput  = screen.getByTestId("searchInput")

        fireEvent.change(searchInput,{target:{value:"burger"}})

        fireEvent.click(SearchButton)

        const searchCards = screen.getAllByTestId("resCard")

        expect(searchCards.length).toBe(2)
    })

    it("Should filter Top rated restaurant after clicking buttor

        await act(async() =>
            render(
            <BrowserRouter>
                <Body/>
            </BrowserRouter>
        ))

        const TotalCards = screen.getAllByTestId("resCard")
        expect(TotalCards.length).toBe(18)

        const Button = screen.getByRole("button",{name:"Top Rate

        fireEvent.click(Button)
```

```
        const searchCards = screen.getAllByTestId("resCard")

        expect(searchCards.length).toBe(6)
    })

    it("should render Username",async() => {
        await act(async() =>
            render(
            <BrowserRouter>
                <Body/>
            </BrowserRouter>
        ))
        const userInput = screen.getByTestId("userInput")
        expect(userInput.value).toBe("Gourav")
    })
})
```