INCEPTION

Hello world using HTML

Simply type html:5 in vs code it will generate basic html code with header and body by making use of emmet

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <div id="root">
        <h1>Hello world</h1>
    </div>
</body>
</html>
```

Hello world from JavaScript

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <div id="root">
    </div>

    <script>
        let heading = document.createElement('h1');
        heading.innerHTML = 'Hello world form JavaScript';

        let root = document.getElementById('root');

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

Injecting React superpowers using CDN

- We have to add two CDN links one is for React core (react.development.js) another one is for DOM operations (react-dom.development.js).

- React have two CDN because the core react we will be able to use with other react based apps such as React Native, React 3D etc.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link rel="stylesheet" href="./index.css">
    <title>Document</title>
</head>
<body>
    <div id="root"></div>

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

Hello world from React

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

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

    <script>
        let heading = React.createElement('h1', {id: 'heading'}, 'Hello world
from React');

        let root = ReactDOM.createRoot(document.getElementById('root'));

        root.render(heading);
    </script>
</body>
```

- The createElement() method of React is having three arguments they are Tag, Attributes and children respectively
- createElement(<tag>, <attributes>, <children> or [<child1>, <child2>])
- CreateElement() will return an object which is having props, which is a combination of attributes and children
- Root is where all our react elements will get rendered
- All the contents we written inside the root elements will get replaced by the content which we will add with root.render() method
- We will be able to add React to an existing Jquery applications without affecting anything as it is a library
- Order in which the script imports do matter as well as the 'root' where we render the React, that can be small portion of the page as well

Complex structure implementation ex:

```
 *  <div id="parent">
 *      <div id="child">
 *          <h1>I'm an h1 tag :) </h1>
 *          <h2>I'm an h2 tag :) </h2>
 *      </div>
 *      <div id="child2">
 *          <h1>I'm an h1 tag :) </h1>
 *          <h2>I'm an h2 tag :) </h2>
 *      </div>
 *  </div>
 */

let parent = React.createElement('div', { id: 'parent'}, [
        React.createElement('div', { id: 'child'},[
            React.createElement('h1', {}, "I'm an h1 tag"),
            React.createElement('h2', {}, "I'm an h2 tag")
        ]),
        React.createElement('div', { id: 'child2'},[
            React.createElement('h1', {}, "I'm an h1 tag"),
            React.createElement('h2', {}, "I'm an h2 tag")
        ])
    ]);

let root = ReactDOM.createRoot(document.getElementById('root'));

root.render(parent);
```

IGNITING OUR APP

- To make our code production ready code we should remove comments and console, also should do code minification, bundling, splitting, image optimization, chunking,
- React works so fast is not only because of its performance but, it's also with the additional packages we add to work along with it

- NPM is not node package manager instead it will manage packages
- Create-react-app It automatically has NPM inside it
- To make our project use NPM, we can add NPM to our project by using
  npm init command with relevant inputs which generate package.json
- Package.json – It is a configuration for NPM also helps on dependency management by its version
- Most important package: Bundler
- Bundlers are used to bundle/packages our app so that it will be production ready
  IE: minified, cached, compressed, cleaned
- Bundler ex: Webpack, Parcel, Wheat
- create-react-app uses Webpack as the bundler (Webpack and bebel)
- npm install -D parcel
  There are two type of dependencies one is dev and the other is normal (used in all, ie in production as well)
  Dev is used only in the development phase
  -D is used to indicate that it is a dev dependency
- In package.json if we have versions with
  ~ as prefix which indicate that it will allow automatic patch updates
  ^ as prefix, it will allow patch and minor updates
  [major. minor. patch]
- package.json will keep track of what version of package installed into our system and will have ~ or ^ to mark how it can upgrade
- package-lock.json will keep a track of exact version that is being installed
- integrity key is basically having a hash which will make sure the deployed version is same as in the local
- Transitive dependency – dependency chain of dependencies,
  In our case we need parcel as dependency but parcel have its own dependency, that each dependency has their own dependencies and so on. Thus, it's a transitive dependency
- In effect will be having a lot of package.json files because each of our dependencies have its own (1.08 in video revision)
- npm <cmd> is used to install a package, npx <cmd> is used to execute a package
- npx parcel index.html -> used to ignite out app
- As of now we added React into our app using CDN, we will be able to add via NPM as well
- CDN links are not a right way to inject React into our app,
  - If there is a network delay it might affect it.
  - We have to keep on updating the links if there is version update
- Browser scripts cannot have imports or exports. -> We will get such an error if we try to import/export any files in normal JS files to resolve it we have to add type="module" as an attribute in script tag where we import our JS file
- By using type="module" -> It will considered as module
- npx parcel index.html
  Parcel will go to our index.html and build a dev build of our app and host that build into localhost
- Commands to insall React using NPM

npm i react
npm i react-dom

- Just after this installation, if we remove our CDNs and try to execute our app using parcel will fail as we didn't inject React to our app
- Importing React into our app
  import React from 'react'
  import ReactDOM from 'react-dom/client'
  Now if we try to execute our app, it will fail as we added the JS file as normal script import <script src="/App.js"></script
  To make it work use <script type="module" src="/App.js"></script> so that browser will consider it as module

- Parcel
  Dev Build
  Local Server
  HMR = Hot Module Replacement
  File Watching Algorithm – written in C++
  Caching – Faster build (.parcel-cache)
  Image Optimization
  Minification
  Bundling
  Compress
  Consistent Hashing
  Code Splitting
  Differential Bundling – support older browsers
  Diagnostics
  Error Handling
  HTTPs
  Tree Shaking – Remove unused code for us
  Different dev and prod builds

- HMR
  We will be able to see the auto refresh of our app whenever we made any change and save the code, it done with the help of HMR which uses file watching algorithm

- Prod build
  npx parcel build index.html   // 'build' will helps to make prod build
  We have to remove 'main: "App.js"' from package.json as we already give the entry point as index.html for parcel

- browserlist
  We have to use browserlist to make our app support older browser
  We will be adding it in package.json
  Even if we add browser list configuration only for chrome but still it will work in other browsers but there is no guarantee it will not break
  Refer : https://browserslist.dev/?q=bGFzdCAyIHZlcnNpb25z

By adding browserlist only for a specific browser, we can reduce the bundle size

It can be used for country specific bundling as well

LAYING THE FOUNDATION

NPM Scripts

- Instead of writing the command 'npx parcel index.html'. We should make use of npm scripts to start our application for that we have to add the dev and prod build scripts in package.json
  Eg:
  ```
  "scripts": {
    "start": "parcel index.html",
    "build": "parcel build index.html",
    "test": "jest"
   },
  ```
- Once we have the scripts ready, we can execute it using
  npm run <name of the script>
  eg: npm run start
- 'npm run start' is similar to 'npm start'
- We can add 'Not Rendered' inside the content of div id root. It will be helpful to identify there is any problem in rendering

JSX

- Syntax to create React Element, used to write HTML in React
- JSX syntax to create React Element
  ```
  const jsxHeading = <h1 id="heading">Namste React using JSX</h1>
  ```
- React Syntax to create React Element
  ```
  const heading = React.createElement("h1", { id: "heading"}, "Namste React🚀");
  ```
- JSX is not part of React
- We can create React Apps without JSX also
- Which will merge the HTML and JS
- It is not HTML in JS, It is HTML like syntax
- React Element using JSX eg:
  <h1 id="heading"> Namaste React </h1>
- JSX is not a valid pure Javascript because JS engine will understand only ECMA script
  Still we are able to execute it because of Parcel😊
- Parcel will transpiled our entire code before it reaches JS engine by using Babel
- Transpiation – Convert our code to make it understandable by Browser

- Babel
  - Javascript Compiler
  - Babel will transpile our code, so that React can understand it
  - Bable will do the following steps on JSX:

JSX -> Babel transpiles it to React.createElement() -> React Element (JS Object) -> HTML Element(render)

- In JSX the attributes should be in camel case, className is an attribute

```
const jsxHeading = <h1 className="head" >Namste React using JSX</h1>
```

- Multiline JSX code should enclosed between parenthesis
- VS Code Extensions:
  - Prettier
  - ESLint
  - Bracket Pair Colorization Toggler
  - Better Comments

React Component

- Class Based Components (OLD)
- Functional Components (NEW)

Class Based Components

- It uses JavaScript Classes to create components.

Functional Components

- Function that will return JSX code or React Element
- Uses JavaScript Functions to create components.
- We can use any ways to write the function like function statement, function expression or arrow function, Arrow function is the modern standard
- Component name should be in Pascal Case

```
const HeadingComponent = () =>
    <h1 className="heading">Namaste React Functional Compoents</h1>
```

Component Composition

- Component inside another component
- To make it work our incoming component should be in native HTML tag

```
const Title = () => {
  return (
    <h1 className="head" tabIndex="2">
      Namste React using JSX
    </h1>
  )
};
const HeadingComponent = () => (<div><Title />
    <h1 className="heading">Namaste React Functional Compoents</h1>
  </div>)
const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(<HeadingComponent />);
```

JavaScript inside JSX

- We will be able to write/execute any JavaScript code/expression inside JSX by adding it in curly braces

```
const HeadingComponent = () => (
  <div>
    {console.log("Hey")}
    <h1 className="heading">Namaste React Functional Compoents</h1>
  </div>
)
```

- We can add a React element inside React Element

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

const title =  (
    <h1 className="head" tabIndex="2">
      {header}                                -> nested element
      Namste React using JSX
    </h1>
  )
```

- We can add a React element inside React Component

```
const title =  (
    <h1 className="head" tabIndex="2">
      Namste React using JSX
    </h1>
  )

const HeadingComponent = () => (
  <div>
    {title}                                    -> element inside component
    <h1 className="heading">Namaste React Functional Compoents</h1>
  </div>
)
```

- We can add React Component inside React Element

```
const title =  (
  <div>
    <h1 className="head" tabIndex="2">
      Namste React using JSX
    </h1>
    <HeadingComponent/>                        - Component inside element
  </div>
  )
```

- JSX prevents XSS – Cross Site Scripting attacks by sanitizing the code
- We should add JS code or React Components within a parent tag, else it will throw error, It won't allow siblings
- We can add React Components in three different ways

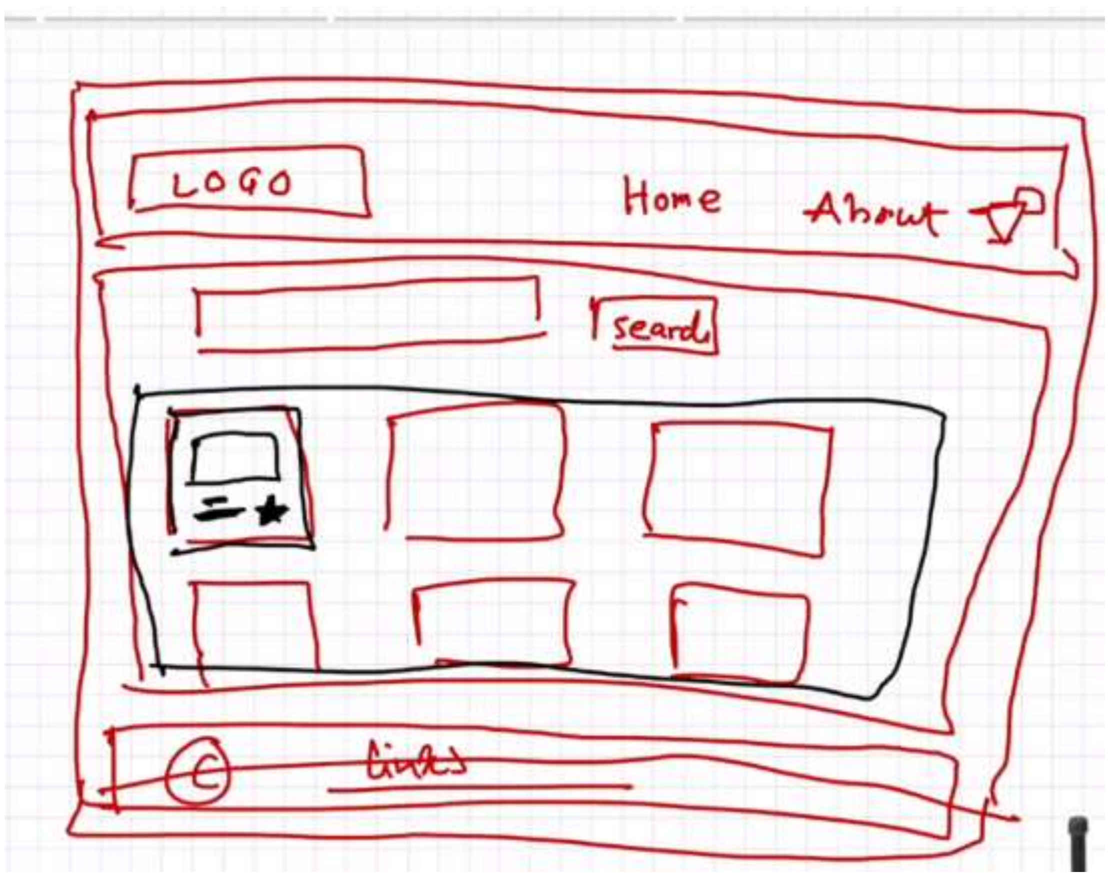
```
<div>
    {Title()}
    <Title/>
    <Title></Title>
    <h1 className="heading">Namaste React Functional Compoents</h1>
  </div>
```

- Custom attributes in JSX should be in lowercase

<u>TALK IS CHEAP SHOW ME THE CODE</u>

Planning

- Planning is key thing while building an application
- We should create a wireframe and identify the various components required for our app



Coding

Style in JSX

- Styles can be inline or external in React
- For inline styles we will be using JS object
  Ex:
  Const styleCard = { backgroundColor: ‘red’ }
  <div style={styleCard}> </div>

Props

- Shortform for properties
- which we can pass to the components as arguments
- Props are normal arguments to a function
- Passing props to a component ~(Similar to) Passing arguments to a functions
- Whenever we need to pass any dynamic data to a component, then we will be making use of props
- React will wrap the props we pass into a component as an object
  Ex:

```
Passing properties
<RestaurantCard resName="Top Form" cuisine="South Indian, Kerala"/>
<RestaurantCard resName="KFC" cuisine="Fast Food"/>
```

```
const RestaurantCard = (props) => {
  const {resName, cuisine} = props;
  return (
    <div className="res-card" style={resCardBackground}>
      <img
        className="res-logo"
        alt="res-logo"
        src="https://media-
assets.swiggy.com/swiggy/image/upload/fl_lossy,f_auto,q_auto,w_660/b530
3a94c367062c158ce278bf6307a3"/>
      <h3>{resName}</h3>
      <h4>{cuisine}</h4>
      <h4>4.4 Stars</h4>
      <h4>40 Minutes</h4>
    </div>
  )
}
```

-

Config Driven UI

- The websites driven by configs, so that by using a single application we can cover different scenarios
  Ex:
  If we consider a food ordering app like Swiggy, it's offer and content will change based on location so the config is basically coming into picture here

Loops

- If we want to loop over components, we will be making use of JavaScript methods (map here).
- While we are looping over components, we should definitely give a 'key' attribute to uniquely identify component in loop. It's basically a keyword
- The 'key' attribute helps in optimization while re-rendering the elements in the loop again with new element, by rendering only the new element and keeps the old elements rendered as it is.

- Key – For render cycle optimization
- As per official documentation, it's not recommended to use index as 'key' for list element in loop

{<TitleComponent/>} vs {<TitleComponent></TitleComponent>} in JSX.

- Opening and closing tag is required when we need to add content in between, which will be considered as 'children' prop

How can I write comments in JSX?

- The syntax {/* */} to wrap around the comment text.

What is <React.Fragment></React.Fragment> and <> </>?

- Allows us to group elements without a wrapper node
- If we don't need to add any props like 'key' then we can use the empty <></>
  Else we should use <React.Fragment></React.Fragment> or <Fragment></Fragment>

Virtual DOM?

- The virtual DOM (VDOM) is a programming concept where an ideal, or "virtual", representation of a UI is kept in memory and synced with the "real" DOM by a library such as ReactDOM. This process is called reconciliation.

What is Reconciliation in React?

- The process of syncing virtual DOM with the real. Which will use diffing algorithm to update the Real DOM by updating only the elements which got changed.
- Assumptions in change detection:
  - Two elements of different types will produce different trees.
  - The developer can hint at which child elements may be stable across different renders with a key prop.

What is React Fiber?

React Fiber is a concept of ReactJS that is used to render a system faster, smoother and smarter. The Fiber reconciler, which became the default reconciler for React 16 and above, is a complete rewrite of React's reconciliation algorithm to solve some long-standing issues in React. Because Fiber is asynchronous, React can:

- Pause and restart rendering work on components as new updates come in
- Reuse previously completed work and even abort it if not needed
- Split work into chunks and prioritize tasks based on importance

Why do we need keys in React?

- Keys are used in React to identify which items in the list got modified

Can we use index as keys in React?

Yes, we can use the index as keys, but it is not considered as a good practice to use them because there will be negative impact when the order got changed

LET'S GET HOOKED

- One of the best practices is to create separate files for separate components
- React folder structure is depended on developer. It can be based on feature or file type etc.
- File name of a component should be same as component name as per convention
- We can make the component name extensions as .js or jsx both are same. If we use typescript then we have to use .tsx
- File export can be done in two ways
    - Default Export
      If we have only one thing to export
      Eg: export default <name_of_thing>
      Import of default export: import <thing> from <path>

    - Named Export
      To export multiple things
      Eg: export const <thing1>
           export const <thing2>
      Import of named export: import { <thing1>, <thing2> } from <path>
- React is faster because of faster DOM manipulation as well

Event Listeners

- We will be able to use all default HTML event attributes
- In React we will use it as in camel case
  ```
  Ex: onClick={()=>{                    // Callback function which will be called on click
  Console.log('button clicked')         //
  }}
  ```
- We will be passing a callback to the event listeners to execute it

React Hooks

- JavaScript utility functions given to us by React
- Two important Hooks are
    - useState() – To create super powerful state variables in React
    - useEffect()

Variable using useState()

- We can create a local state variable inside a component using useState()
  ```
  Ex:
  Header = () => {
          Const [<variableName>, <setVariableName>] = useState(<defaultValue>)
  }
  ```

- The < setVariableName > will be a method used to update the variable, as we will not be able to update it like a normal variable
- If we don't want to update the variable then we don't need the <setVariableName>

- State variables will work same as normal variable but we won't be able to update it by variable assignment
- <setVariableName> - by convention we are naming it as a set prefix with variable name, but it can be any name
- State variables keep the UI layer in sync with the data layer
- *Whenever the state variable updates, React will trigger the reconciliation cycle (re-renders the component)*
- Used to create super powerful state variable, Its super powerful because React will keep track of state variables. Whenever the state variable updates React will trigger the diff algorithm and find the difference between Current Virtual DOM with Previous Virtual DOM and automatically updates the Actual DOM

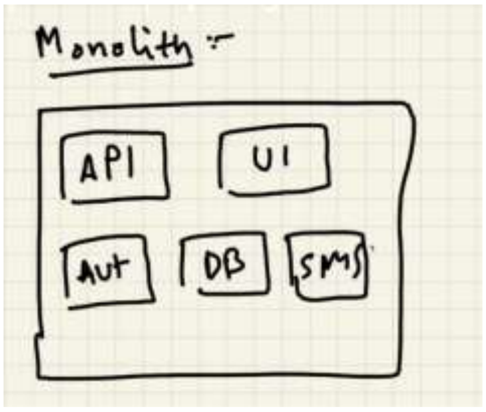
Reconciliation Algorithm (React Fiber)

- React Fiber - It got introduced in React 16
- It's a new way of finding the diff and updating the DOM
- Virtual DOM
    - Representation of Actual UI
    - Basically, a JavaScript Object
      Ex: when we console a react element/component we will get a JavaScript Object
- *React is faster because of efficient DOM manipulation using React Fiber (explain..)*
  As JavaScript Object comparison is faster than DOM tree comparison. Diff algorithm will make use of virtual DOM to compare and find the difference and then Updates the Actual DOM
- <setVariableName> is used to trigger the diff algorithm

<u>EXPLORING THE WORLD</u>

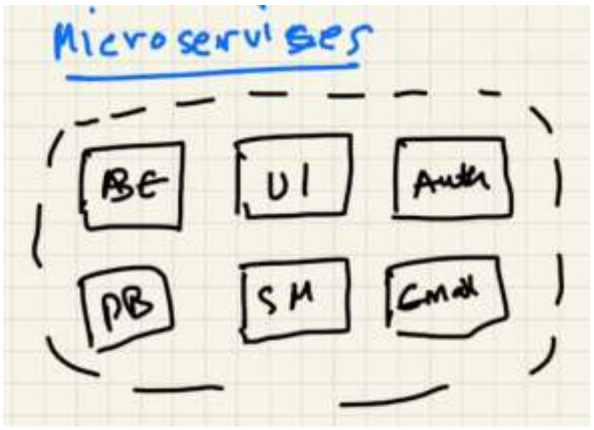
Monolith

- In monolith architecture everything will be in a single project



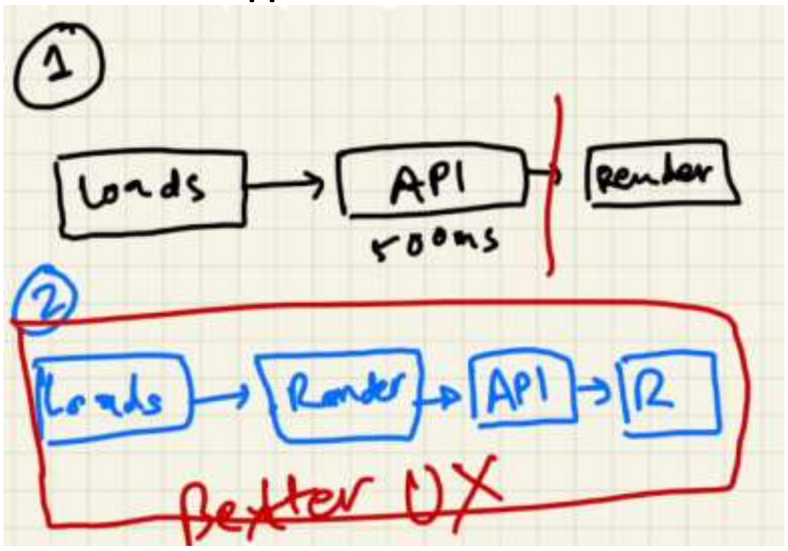
Eg: A single Java project which is having UI, API, Auth, DB etc, everything written in Java in a single repo. So, if we need to make any change in any single service (UI), we have to make the whole project deployment

Microservices



- Separation of concerns
- Single responsibility principle – Each and every service has its own job
- Here we can have separate repo for each service
- We will be able to use different tech stack for different service like React for UI, Python for backend etc. which we will choose by understanding which one is best for a specific service

There are two approaches to interact with backend API



The useEffect() Hook

- Its like a normal function
- Syntax: useEffect(<callback()>, <dependency array>)
- The useEffect callback() method will be called after the component rendering
- It will be helpful to execute the code post rendering of our component

CORS

- Before the CORS policy, browsers won't allow to communicate between cross origin

Which means
https://abc.com will not be able to communicate with
https://def.com – different domain
https://api.abc.com – different sub domain
https://abc.com:5000 – different port
http://abc.com – without SSL

- As CORS is a web standard now, whenever our web app needs communicate with another one in different server, first it will send a preflight/options request and get a response. If the response header will have 'Access-Control-Allow-Origin' header as '*' or our origin name then it can make the actual call else it will throw CORS error

Shimmer UI

- It makes better user experience by displaying fake UI instead of a loading spinner

Conditional Rendering

- Rendering content inside the Component based on certain condition

Promise APIs

- Following APIs are used to handle multiple promises
- Promise.all()
    - In Success scenario it will wait till all the promises to get resolved and return the result
    - In failure scenario, if any of the promise got rejected then it will return the result as rejected at that time itself
      Ex: To handle multiple API calls
      ```
      Promise.all([p1, p2, p3])
              .then(result => {})
              .catch(error => {})
      ```
- Promise.allSettled()
    - Irrespective of resolve or reject it will wait till all the promises to get **settled**
    - It will return an array of objects with result and status
      ```
      Promise. allSettled ([p1, p2, p3])
              .then(result => {})
              .catch(error => {})
      ```
- Promise.race()
    - It will consider only the first promise which got settled
      ```
      Promise. race([p1, p2, p3])
              .then(result => {})
              .catch(error => {})
      ```
- Promise.any()
    - It will wait to get at least one promise to get resolved,
    - If everything fails it will return Aggregated error

      ```
      Promise. any([p1, p2, p3])
              .then(result => {})
              .catch(error => {})
      ```

FINDING THE PATH

useEffect

- The callback method argument is mandatory for useEffect(<callback>,<dependency array>)
- If no dependency array => useEffect is called on every component render

```
useEffect(() => console.log("header use effect"));
```

- If we have a dependency array, even its empty then the useEffect callback is called only once on its initial render

```
useEffect(() => console.log("header use effect"), []);
```

- If we have anything inside dependency array then the useEffect callback is called everytime when the dependency changes, along with the initial render call

```
const [btnName, setBtnName] = useState("Login");

useEffect(() => console.log("Use effect called"), [btnName]);
```

useState

- Never create a state variable out of our component body, as it causes error
- Used to create local state variables inside the functional components
- We should call it on the top i.e. the start of the function
- It's not recommended to create state variables inside our if conditions as it causes inconsistency to our program
- It's also not recommended to create state variables inside for loop and functions

React router DOM

- For routing in react
- We will be adding routing config in App.js. The main file

createBrowserRouter

- Used to create routing configuration
- Routing configuration – Which will define when we access a path what should happen
- We will be defining routes as an array of objects, in which each object will represents a route. Which will contain a path and element

```
const appRouter = createBrowserRouter([
  {
    path: "/",
    element: <AppLayout/>
  },
  {
    path: "/about",
    element: <About />
  }
]);
```

- We should provide the defined routes to render. For that we will be making use of RouterProvider component

```
root.render(<RouterProvider router={appRouter} />);
```

- If we simply type 'rafce' in VS Code. It will create a react component

Error Handling

- If the user enters a path which is not defined then react-router-dom will redirect to its error page. If we need to display our custom error page, then we should define it in route object on errorElement property as a component
- We will be defining the errorElement property in the root/home route object

```
  {
    path: "/",
    element: <AppLayout/>,
    errorElement: <Error />
  },
```

- We can identify hooks by checking the 'use' prefix
- useRouteError – Hook provided by react-router-dom to get the error details when there is an error with route, like not found

```
const Error = () => {
  const err = useRouteError();
  return (
    <div>
      <h1>Oops!!!</h1>
      <h2>Something went wrong!!</h2>
      <h3>
        {err.status}: {err.statusText}
      </h3>
    </div>
  );
};
```

Child Routes

- There are scenarios when a component should be fixed and others are dynamic like Header is fixed and body can change according to routes. In such scenarios we can make use of child routes
- The child routes are defined inside the route object under 'children' property as an array of route objects

```
{
    path: "/",
    element: <AppLayout />,
    errorElement: <Error />,
    children: [
      {
        path: "/",
        element: <Body />,
      },
      {
        path: "/about",
        element: <About />,
      },
      {
        path: "/contact",
        element: <Contact />,
      },
    ],
  }
```

- We should mark place in the parent component where the child routes need to be displayed according to the path by using ‘Outlet’ component of react-router-dom

```
<div className="app">
      <Header />
      <Outlet />
    </div>
```

Link Component

- Link component of react-router-dom - used to navigate between components.
- If we try to use href it will reload the entire application.
- Link component is similar to anchor tag, the difference is, it will use ‘to’ instead of ‘href’

```
<Link to="/about">About Us</Link>
```

- The routes are helping us to achieve the single page application concept

Routing in web apps

- Client Side Routing
  - Without making a network call and navigating between pages
  - Single page applications
- Server Side Routing
  - Making a network call and navigating between pages
  - Legacy web apps

Dynamic Routes

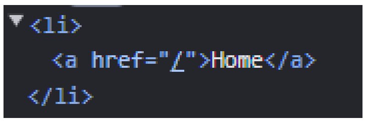
- We can define a dynamic route in route object by making the value of path as a string having colon prefix variable name after the slash

```
{
        path: "/restaurant/:resId",
        element: <RestaurantMenu />,
}
```

- We will be using ‘useParams’ hook from react-router-dom for getting dynamic route parameters

```
const params = useParams();
```

- Link component is a wrapper over anchor tag
  It will be rendered as an anchor tag in UI



```
▼<li>
   <a href="/">Home</a>
 </li>
```

Add Images into our App

- Importing from local directory

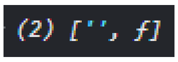
```
import logo from "./../reactLogo.jpg";

<img src={logo} />
```

- Externally hosted image

```
<img src="https://logos-world.net/wp-content/uploads/2023/08/React-
Logo-500x281.png" />
```

useState console

```
console.log(useState(""));
```



```
(2) ['', f]
```

<u>LET's GET CLASSY</u>

Class based components

- Javascript Class which extends React.Component to make it as component
- Which will have a render method that returns JSX
- *Javascript Class which extends React.Component and have render method which will return piece of JSX code*
- Once we created the class we should export, import and implement it as we do the functional component

```
class UserClass extends React.Component {
  render() {
    return (div className="user-card">div>);
  }
}
```

Passing Props into class-based components

- We can pass it in the same way as in functional components

```
<UserClass name="Mikhil Class" location="Calicut class" />
```

- But while receiving it we will be getting it inside the constructor

```
class UserClass extends React.Component {
  constructor(props) {
    super(props);
  }

  render() {}
}
```

- Here we should use super(props) method to call the parent class constructor with the props
- Using 'this' keyword we can access the props anywhere in the class as we are passing props to parent via constructor

```
render() {
    const { name, location } = this.props;
}
```

Creating State variable in class based component

- The class based components were the first version of components and at that time we didn't had functional components and hooks. So, we created the state variables in a different way
- The 'state' is a keyword in class based components
- State is created whenever a class instance got created
- Constructor is the best place to create state variable as it will get executed on instantiating the class
- We will be creating a state variable by assigning an object which contains state variables to this.state

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

```
<h1>Count: {this.state.count}</h1>
```

- Multiple state variable in class based components

```
this.state = {
   count: 0,
   count2: 1,
};
```

- Never update state variables directly
- The setState() method is used to update the state variables. Where we can pass our updated state variable inside an object

```
this.setState({
   count: this.state.count + 1,
});
```

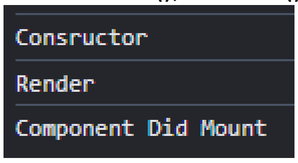
- We will be able to update a single state variable, Even if our state object have lot of state variables using setState() method

Component life cycle

- Import React from “react”
  Class x extends React.Component => which is same as

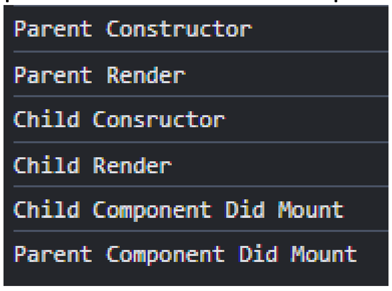
  import { Component } from “react”
  Class x extends Component

- Order of execution of class-based component:
  constructor(), render(), componentDidMount()



```
Consructor
Render
Component Did Mount
```

- If we have a child component inside the parent then the componentDidMount() of parent is called after completing the child component life cycle methods

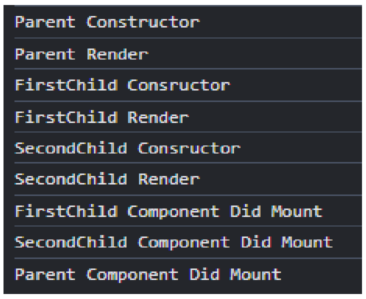


```
Parent Constructor
Parent Render
Child Consructor
Child Render
Child Component Did Mount
Parent Component Did Mount
```

componentDidMount()

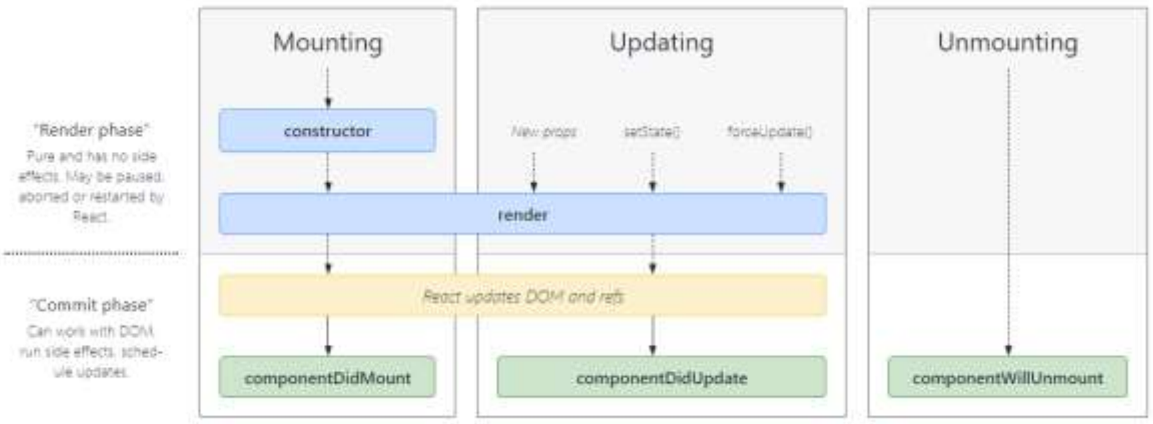
- To execute some code(make API calls) once we mounted the component successfully
- Is used to make API calls
- Similar to useEffect(<cb>, []) callback with empty dependency array
- It basically helps us to gives a better user experience by render the component quickly then makes the API call and fill the data

Multiple Child



```
Parent Constructor
Parent Render
FirstChild Consructor
FirstChild Render
SecondChild Consructor
SecondChild Render
FirstChild Component Did Mount
SecondChild Component Did Mount
Parent Component Did Mount
```

Mounting Cycle



- Mounting happens in two phases
  - Render Phase
  - Commit Phase

Render Phase

- Combine all child components in current parent to make it a batch rendering.
- React is doing like this for the performance enhancements as updating the DOM is a costly process. React will batch it
- Virtual DOM Update

Commit Phase

- Batch all the child components and do the commit
- Real DOM Update

Making API calls in Class based components

- We will be making API calls in componentDidMount()

```
async componentDidMount() {
    const data = await fetch("https://api.github.com/users/mikhil-m");
    const json = await data.json();
    this.setState({
      userInfo: json,
    });
  }
```

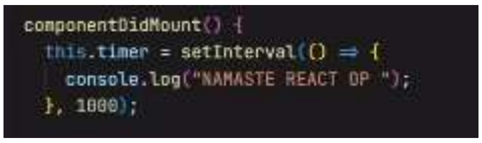
Update Cycle

- It will triggered on updating the state with setState()
- Which will also call the render() method to update the virtualDOM
- After the render() method it will update the Real DOM and call componentDidUpdate

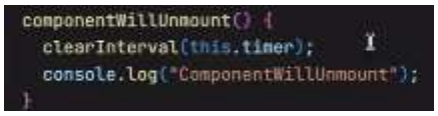
Unmounting Cycle

- It will happen when the component will disappear from the HTML

- Here the componentWillUnmount() method is called just before unmounting our component
- It helps us to cleanup setIntervals() or similar stuffs on navigating to different component
  Ex:



```
componentDidMount() {
  this.timer = setInterval(() => {
    console.log("NAMASTE REACT OP ");
  }, 1000);
```

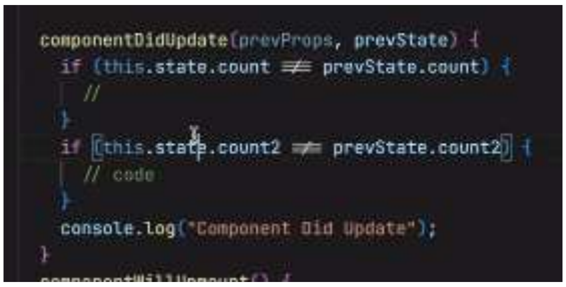


```
componentWillUnmount() {
  clearInterval(this.timer);
  console.log("ComponentWillUnmount");
}
```

```
/**
 * -----MOUNTING----
 *
 *  Constructor(dummy)
 *  Render(dummy)
 *  <HTML Dummy >
 *  componentDidMount(dummy)
 *    <API Call>
 *    <this.setState> -> Update state variable
 *
 * -----UPDATING----
 *
 *  render(API Data)
 *  <HTML API Data>
 *  componentDidUpdate(API Data)
 *
 * -----UNMOUNTING----
 *
 *  componentWillUnmount()
 */
```

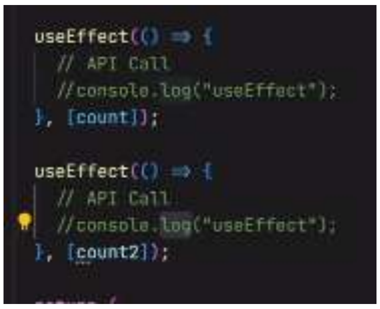
useEffect vs Class based component lifecycle method

- useEffect have the capability to do more functionalities than one lifecycle method
- If useEffect is without dependency array it will act as both …DidMount() and …DidUpdate()
- useEffect callback can be called based in dependency array but in class based. We have to add the conditions manually in componentDidMount() by comparing previous and current state
- If we have to do different set of logic based on different dependency variables then we can make use of different useEffect methods for each but in class based, we have to add more conditions
  Ex: Class Based



```
componentDidUpdate(prevProps, prevState) {
  if (this.state.count !== prevState.count) {
    //
  }
  if (this.state.count2 !== prevState.count2) {
    // code
  }
  console.log("Component Did Update");
}
componentWillUnmount() {
```

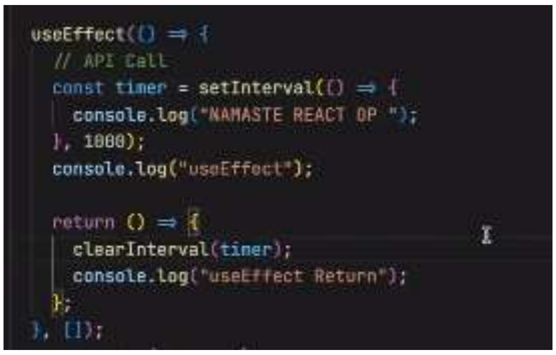
Ex Functional Based



```
useEffect(() => {
  // API Call
  //console.log("useEffect");
}, [count]);

useEffect(() => {
  // API Call
  //console.log("useEffect");
}, [count2]);
```

Alternative for componentWillUnmount in Functional components

- If we want to clear our set intervals in the useEffect() functional components then we can return a callback from functional components where we should define the clear interval and will get called automatically on component unmount



```
useEffect(() => {
  // API Call
  const timer = setInterval(() => {
    console.log("NAMASTE REACT OP ");
  }, 1000);
  console.log("useEffect");

  return () => {
    clearInterval(timer);
    console.log("useEffect Return");
  };
}, []);
```

Why async callbacks cannot be used in useEffect

- The useEffect hook expects its effect function to return either a cleanup function or nothing at all

<u>OPTIMIZING OUR APP</u>

Custom Hooks

- Hooks are just utility functions
- Custom hooks will make our code more modular, readable, reusable and testable
- As per convention hooks are created inside util folder and for each hook we will use separate file with same name as the hook
- Hook name will start with 'use' so that react can understand but it's not mandatory
  Custom hook usage ex:

```
const resInfo = useRestaurantMenu(params.resId);
```

  Hook definition ex:

```
import { useEffect, useState } from "react";
import { MENU_URL } from "./constants";

const useRestaurantMenu = (resId) => {
  const [resInfo, setResInfo] = useState(null);
  useEffect(() => {
    fetchMenu();
  }, []);
  const fetchMenu = async () => {
    const data = await fetch(MENU_URL + resId);
    const json = await data.json();
    setResInfo(json.data);
  };
  return resInfo;
};

export default useRestaurantMenu
```

- Custom hooks will have its own state, useEffect and lifecycle

Chunking

- Splitting our app into smaller logical chunks
- Alternate names
  - Code splitting
  - Dynamic Bundling
  - Lazy loading
  - On demand loading
  - Dynamic import

Lazy Loading

- To implement lazy loading we will be using lazy() method of react library
- The components to be lazy loaded are imported using lazy method
  Ex:

```
const Grocery = lazy(() => import("./components/Grocery"));
```

- We should wrap our lazy loaded components using <Suspense/> component else it will throw an error as it is not immediately available while navigating
- We should pass a content to the fallback of Suspense to display something while it tries to load the lazy loaded component as it requires some time to load the lazy loaded component
- Fallback is not mandatory
  Ex:

```
{
        path: "/grocery",
        element: (
          <Suspense fallback={<h1>Loading...</h1>}>
            <Grocery />,
          </Suspense>
        ),
},
```

JO DIKHTA HAI, VO BIKTA HAI

Sass/Scss

- CSS with superpowers
- It's not a best practice for larger apps

Libraries and Frameworks for Styling Components

- Styled components
- Material UI
- Chakra UI
- Bootstrap
- Ant design
- Tailwind CSS

Tailwind CSS

- Generic CSS framework which we can use with any frameworks such as React, Angular etc.
- We should use the configuration for parcel as we use parcel here
- Commands to install

```
npm install -D tailwindcss postcss
```

- Here we will install postcss package as well. As it is used to transform CSS with JavaScript. Which will be used internally by tailwind CSS
- To configure Tailwind CSS with our app we have to execute the following command. Which will generate a Tailwind config file for our application

```
npx tailwindcss init
```

- We have to create .postcssrc file as a config for postcss and add the following code in it

```
{
  "plugins": {
    "tailwindcss": {}
  }
}
```

- Parcel will use .postcssrc to understand the Tailwind in our app

Tailwind Config File

- Under 'content' property in Tailwind config file we will mention the files where we can use the Tailwind CSS
  Ex:

```
content: ["./src/**/*.{html,js}"],
```

- Tailwind CSS IntelliSense – VS Code extension for better developer experience

Disadvantages of Tailwind CSS

- If we want to apply more styles then we might need to apply that many classes which will make our code looks ugly

Advantages

- It is light weight
- While making the bundle the style bundle will be small as it takes only the styles of the classes which we used even though Tailwind CSS have lot of classes

<u>DATA IS THE NEW OIL</u>

Higher Order Components

- Function that takes a component as an argument and returns back it an enhanced version of it as a new component
  Ex:

  Usage

```
import RestaurantCard, { withHighlyRatedLabel } from
"./RestaurantCard";

const RestaurantCardHighlyRated = withHighlyRatedLabel(RestaurantCard);
```

```
<RestaurantCardHighlyRated resData={restaurant} />
```

Defenition

```
export const withHighlyRatedLabel = (Component) => {
  return (props) => {
    return (
      <div>
        <label className="absolute p-2 m-2 text-white rounded-full bg-
lime-500">
          Highly Rated
        </label>
        <Component {...props} />
      </div>
    );
  };
};
```

Controlled And Uncontrolled Components

- Controlled Components – If the parent component controls the child component
  Ex:
  Parent Component

```
const [showIndex, setShowIndex] = useState(null);
```

```
{categories.map((category, index) => (
        <RestaurantCategory
          key={category?.card?.card?.title}
          category={category?.card?.card}
          showItems={index == showIndex}
          setShowIndex={() => setShowIndex(index)}
        />
      ))}
```

- Child Component

```
const RestaurantCategory = ({ category, showItems, setShowIndex }) => {
  return (
    <div
      className="bg-gray-100 shadow-lg w-6/12 mx-auto my-4 p-2 cursor-
pointer"
      onClick={() => setShowIndex()}
    >
      <div className="flex justify-between">
        <span className="font-bold text-lg">
          {category?.title}({category?.itemCards?.length})
        </span>
        {showItems ? <span>⬇</span> : <span>⬆</span>}
      </div>
      {showItems && <ItemList itemCards={category?.itemCards} />}
    </div>
  );
};
```

- Uncontrolled Components – If it controls its own
  Ex: The given RestaurantCategory component controls it own, basically its actions are handled by itself. Based on showItems its getting hide and show the section

```
import { useState } from "react";
import ItemList from "./ItemList";

const RestaurantCategory = ({ category }) => {
  const [showItems, setShowItems] = useState(false);
  return (
    <div
      className="bg-gray-100 shadow-lg w-6/12 mx-auto my-4 p-2 cursor-
pointer"
      onClick={() => setShowItems(!showItems)}
    >
      <div className="flex justify-between">
        <span className="font-bold text-lg">
          {category?.title}({category?.itemCards?.length})
        </span>
        {showItems ? <span>⬇</span> : <span>⬆</span>}
      </div>
      {showItems && <ItemList itemCards={category?.itemCards} />}
    </div>
  );
};

export default RestaurantCategory;
```

Lifting The State Up/Sharing State Between Components

- If we want the state of two components to always change together. To do it, remove state from both of them, move it to their closest common parent, and then pass it down to them via props. This is known as lifting state up

Props Drilling

- In React there is only one-way data flow
- If we want to pass data from top level parent to 8th level child then we have to pass it as props throughout the child components till it reaches the 8th level child that's Props Drilling and its not a recommended way if there are more levels

React Context

- Helps us to store data in one place that anybody can access.
- One of the ways to avoid Props Drilling is by using React Context.
- Best place to create the Context is inside the utils folder

Context Creation

- We will be able to create any number of Context in our application
- createContext() method of react library is used to create context, we will be passing an object to createContext() as argument to set the default value
  Ex:

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

const UserContext = createContext({
  loggedInUser: "Default User",
});

export default UserContext;
```

- We can access the Context in two different ways
  - useContext() hook
  - <CreatedContext.Consumer> as a component

useContext

- useContext() is the react hook used to get access the context by passing the Context which we needed to get data from as argument
- We will be able to access the Context even in the lazy loaded component
  Ex:

```
import UserContext from "../Utils/UserContext";
```

```
const { loggedInUser } = useContext(UserContext);
```

```
<li className="px-2 py-2 font-bold">{loggedInUser}</li>
```

Accessing Contexts in Class Based Components

- We will be making use of <CreatedContext.Consumer> as a component
- Within its opening and closing tag we will get it as an argument inside callback method
  Ex:

```
import UserContext from "../Utils/UserContext";
```

```
<UserContext.Consumer>
   {({ loggedInUser }) => (
       <h2 className="font-bold">User: {loggedInUser}</h2>
   )}
</UserContext.Consumer>
```

Context Provider

- Context Provider is used to set/override the context values and we can add it as a component at root level by wrapping whole app within it to override the context in whole app

Ex:

```
<UserContext.Provider value={{ loggedInUser: userName, setUserName }}>
    <div className="app">
      <Header />
      <Outlet />
    </div>
</UserContext.Provider>
```

- If we want to override the context only to a specific component then we can wrap that component using Context provider
  Ex:

```
<div className="app">
      <UserContext.Provider value={{ loggedInUser: "TEST" }}>
        <Header />
      </UserContext.Provider>
      <Outlet />
</div>
```

In Summary - Context

- Context is a global space; we can provide it to whole app or just a small portion of our app. We can create multiple contexts. We can create a new context for specific component. We will be able to override it anywhere we want
- If we tie the context with state variable. Whenever the state variable changes our context will also get updated
- We can use Context for data management in small and medium applications but if it's too big we can make use of external state/data management libraries such as Redux

LET's BUILD OUR STORE

- Redux is not mandatory
- We will be able to create small and mid-sized applications without Redux
- Redux will come into picture on large scale application where data is heavily used
- Redux and React are different libraries
- Redux is not the only library for state management, there are other libraries like Zustand

Advantages

- Data/State management for large scale applications
- Redux offers easy debugging

Redux

- A predictable state container for JavaScript applications
- There are other variants such as Redux Toolkit and React Redux

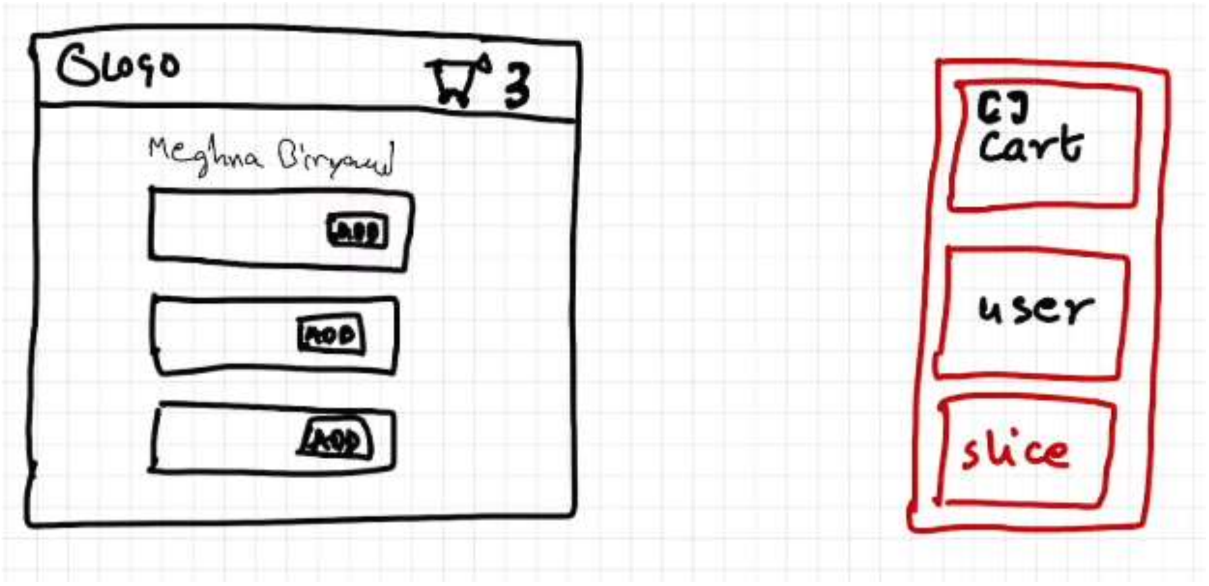
- Redux Toolkit is the latest redux library or we can say it is the modern way for writing Redux
- React Redux will act as a bridge between React and Redux
- Vanilla Redux – Older way of writing Redux

Redux Toolkit (RTK)

- The Redux Toolkit package is intended to be the standard way to write Redux logic
- It was originally created to help address three common concerns about Redux:
  - "Configuring a Redux store is too complicated"
  - "I have to add a lot of packages to get Redux to do anything useful"
  - "Redux requires too much boilerplate code"

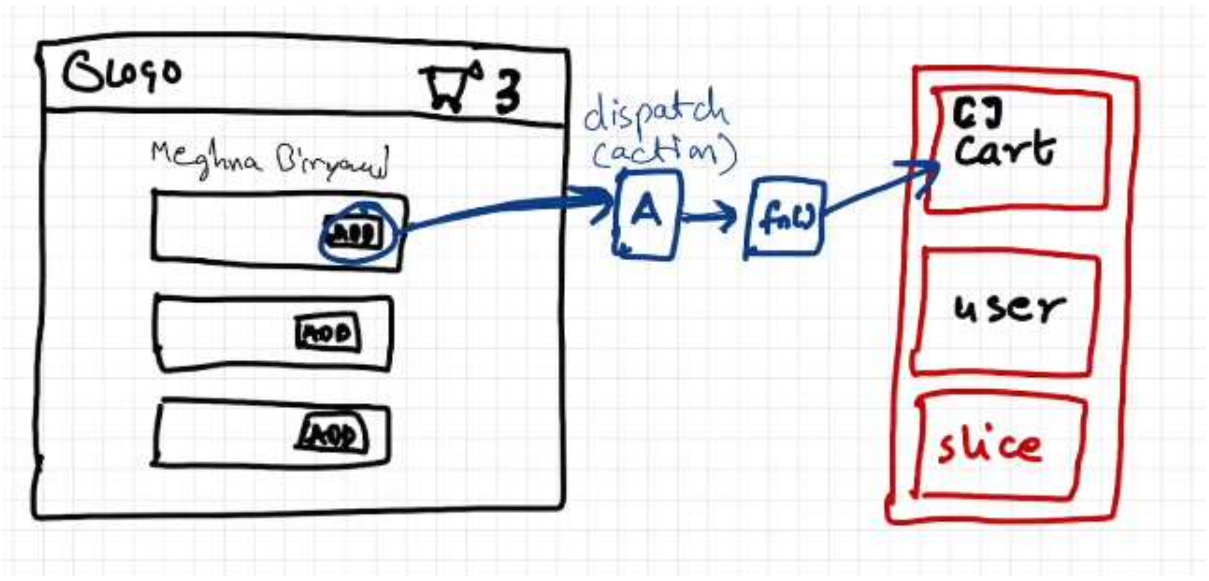
Redux Store

- We can consider it as a big whole object with lot of data inside of it and kept in a central global space
- Any component in our app can access it
- Slices – Used to make our Redux store not clumsy, it's like a small portion of the Redux store to make logical separations
  Ex:
  Cart Slice - To store cart related info
  User Slice - To store logged in user related info
  Theme Slice – Theme related data



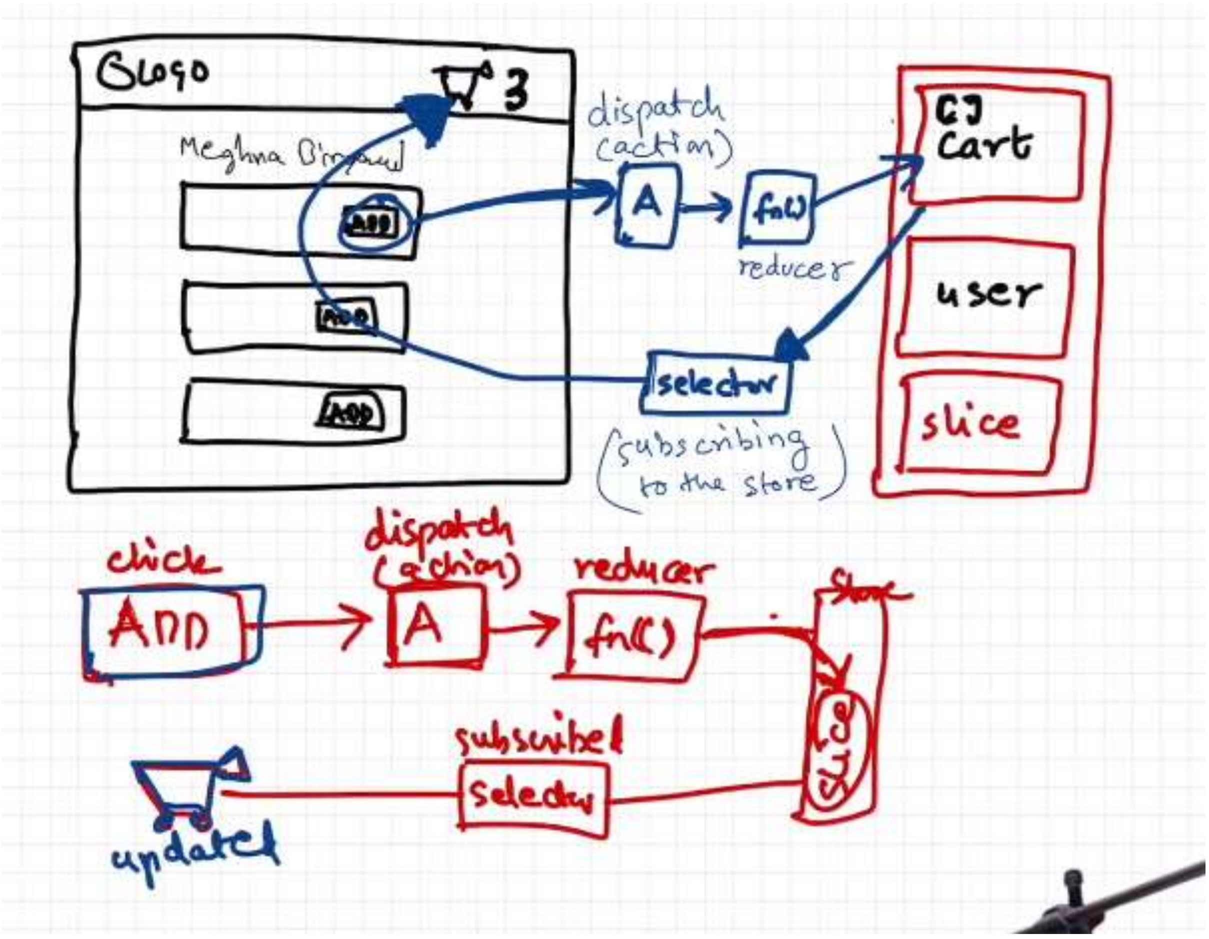
Add/Update Redux Store

- We will not be able to update the data in the store directly
- For Example:
  If we need to add items into cart slice while clicking on add
  First, we will be dispatching an action, it will then call the reducer function which modifies the slice of the redux store (cart)



Read From Redux Store

- Selector - To read data from Redux store as a subscription so that whenever the data in the store updates, it will automatically update the component



Redux Toolkit Code Setup

- Install @reduxjs/toolkit and react-redux
- Build our store
- Connect our store to our app
- Slice (cartSlice)
- Dispatch (action)
- Selector

Build our store

- We will be using configureStore() method of Redux Toolkit to create our store
- The best place to create store is inside our Utils folder
  Ex:

```
import { configureStore } from "@reduxjs/toolkit";

const appStore = configureStore();

export default appStore;
```

Connect our store to our app

- Provider component from react-redux is used to provide our store to our application, basically connecting store with our application
- Provider will act as a component. We will be wrapping our whole application inside the opening and closing tag of 'Provider' to provide the store to the whole application and it will have a 'store' props to which we will be passing 'appStore' which we created
  Ex:

```
import { Provider } from "react-redux";
import appStore from "./Utils/appStore";
```

```
<Provider store={appStore}>
   <UserContext.Provider value={{ loggedInUser: userName, setUserName
}}>
     <div className="app">
       <Header />
       <Outlet />
     </div>
   </UserContext.Provider>
</Provider>
```

- If we don't want to have redux for the whole application then we just have to wrap the small portion which needs redux using 'Provider'

Slice (cartSlice)

- We can keep all Redux related files in a separate folder if we need

- We will be creating a JS file for slice as well in Utils folder (cartSlice.js)
- createSlice() method of Redux Toolkit is used to create a slice
- Which will take a configuration object as an argument to create slice
- The configuration object will have the 'name', 'initialState' and 'reducers'
- initialState will be an object which contains the initial state of the slice
- Reducers are like APIs to interact with the Redux store
- Reducers object will have action and reducer method as key value pairs, basically reducer functions corresponding to actions
- Once we have our slice, we should export the reducer and actions
  Ex:

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

const cartSlice = createSlice({
  name: "cart",
  initialState: {
    items: [],
  },
  reducers: {
    addItem: (state, action) => {
      // mutating the state
      state.items.push(action.payload);
    },
    removeItem: (state, action) => {
      state.items.pop();
    },
    clearCart: (state) => {
      state.items.length = 0;
    },
  },
});

export const { addItem, removeItem, clearCart } = cartSlice.actions;
export default cartSlice.reducer;
```

Adding Slice to our store

- The configureStore() method used while creating a store will accepts a configuration object which will be having all reducers used in our app
- If we need to update the main store, we need a reducer for that as well
- We will be adding Slices inside the main reducer object
  Ex:

```
import cartReducer from "./cartSlice";
const appStore = configureStore({
  reducer: {
    cart: cartReducer,
  },
});
```

Accessing/Read data from the Store

- We will be subscribing to the state using selector
- useSelector is the hook from react-redux
- Which will give us access to the whole store, but we will be taking only the required section
  Ex:

```
const cart = useSelector((store) => store.cart.items);
```

Adding items to the Store

- The dispatch() method of useDispatch hook from react-redux is used to dispatch actions
  Ex:

```
import { addItem } from "../Utils/cartSlice"; -> action
import { useDispatch } from "react-redux";
```

```
const dispatch = useDispatch();
const handleAddItem = (item) => {
  dispatch(addItem(item));
};
```

```
<button onClick={() => handleAddItem(item)}>
    ADD
</button>
```

Best way to subscribe from store

- If we subscribe only on the small portion of the store where we have our data instead of the whole store is the best way
  Ex:

```
const cartItems = useSelector((store) => store.cart.items);
```

- If we subscribe to the whole store, which will cause a performance issue as we will get informed whenever any portion of the store gets updated even though we don't need it as it's a subscription
  Ex: (Inneficient)

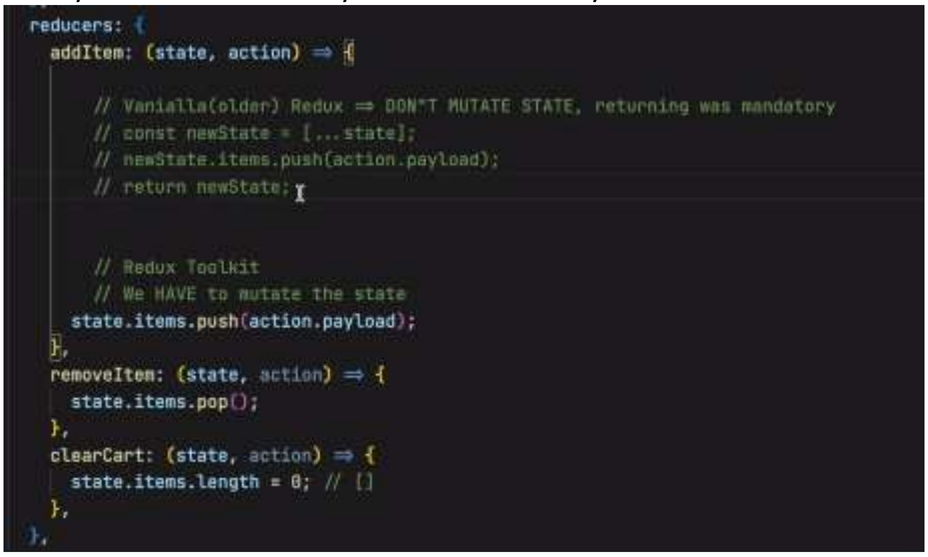
```
const store = useSelector((store) => store);
const cartItems = store.cart.items;
```

Reducer

- Reducer is a combination of small reducers
- In app store we will be having only 'reducer' key while in slices we will have 'reducers' as a key since here we define all reducer functions

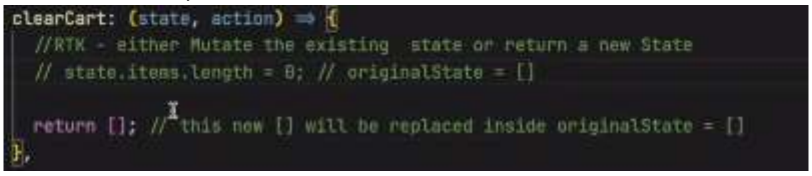
Handling state in Vanilla Redux vs Redux Toolkit

- In vanilla (older) Redux it was mandatory to not mutate the state and we should return the modified state, but in Redux Toolkit we can mutate the state as it has the Immer library which will internally make it immutably



```
reducers: {
  addItem: (state, action) => {

      // Vanialla(older) Redux => DON"T MUTATE STATE, returning was mandatory
      // const newState = [...state];
      // newState.items.push(action.payload);
      // return newState;


      // Redux Toolkit
      // We HAVE to mutate the state
    state.items.push(action.payload);
  },
  removeItem: (state, action) => {
    state.items.pop();
  },
  clearCart: (state, action) => {
    state.items.length = 0; // []
  },
},
```

- We will not be able to see proper object if we console the state, so we should use current() method of redux toolkit
  Ex:
  Console.log(current(state))
- RTK – Mutate / New State



```
clearCart: (state, action) => {
  //RTK - either Mutate the existing  state or return a new State
  // state.items.length = 0; // originalState = []

  return []; // this new [] will be replaced inside originalState = []
},
```

Redux Debugging

- Redux Dev Tools – Helps in debugging

RTK Query

- In older Redux we used Middle wares and Thunk for API calls, but in Redux Toolkit we will be using RTK Query

<u>TIME FOR TEST</u>

Importance of Testing

- If we are trying to add a new feature or even if we change a single line of code which might impact the entire application. So, we should make sure it's not impacting anything else with proper testing

Types of Testing as a Developer

- Unit Testing - Test our React component in isolation

- Integration Testing – Will validate multiple components are able to communicate with each other to make a flow
- End to End Testing – e2e testing – Will test from the point where the user started interacting with our application till the user leaves, and we will be testing all flows
  Tools required to do e2e tests
    - Cypress
    - Puppeteer
    - Selenium

Setup Testing Config

- Testing Config Steps:
    - Install React Testing Library
    - Install Jest
    - Install Babel Dependencies
    - Configure Babel
    - Configure Parcel config file to disable default Babel transpilation
    - Jest – npx jest –init
    - Install jsdom library
    - Install @babel/preset-react – to make the JSX work in test cases
    - Include @babel/preset-react inside the Babel config
    - npm i -D @testing-library/jest-dom
- React Testing Library – Used for testing in React
- It is built on top of DOM Testing Library
- Create-react-app have this testing library already inside of it
- It will use Jest – Jest is a delightful JavaScript Testing Framework with a focus on simplicity
- Command to install React Testing Library
  npm i -D @testing-library/react

- Command to install Jest
  npm i -D jest

- Babel dependencies required to work Jest (https://jestjs.io/docs/getting-started)
  npm install --save-dev babel-jest @babel/core @babel/preset-env

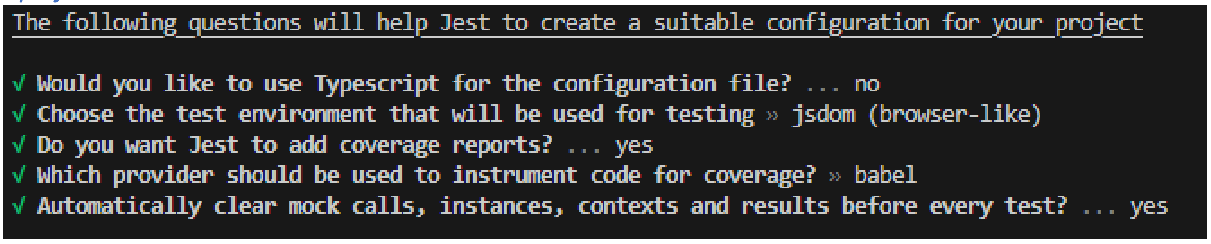
- We will create babel.config.js file at the root level for babel configuration and will add the following code

```
module.exports = {
  presets: [['@babel/preset-env', {targets: {node: 'current'}}]],
};
```

- As Parcel already have Babel transpiler. We should disable it to use the config we added to avoid the conflict, by adding a .parcelrc file with below configuration
  https://parceljs.org/languages/javascript/#usage-with-other-tools

```
{
  "extends": "@parcel/config-default",
  "transformers": {
    "*.{js,mjs,jsx,cjs,ts,tsx}": [
      "@parcel/transformer-js",
      "@parcel/transformer-react-refresh-wrap"
    ]
  }
}
```

- Jest configuration
  Command to generate jest config file – jest.config.js
  npx jest –init



```
The following questions will help Jest to create a suitable configuration for your project

√ Would you like to use Typescript for the configuration file? ... no
√ Choose the test environment that will be used for testing » jsdom (browser-like)
√ Do you want Jest to add coverage reports? ... yes
√ Which provider should be used to instrument code for coverage? » babel
√ Automatically clear mock calls, instances, contexts and results before every test? ... yes
```

- Jsdom
  Environment to run test cases. Like browser but not browser

- As we are using React Testing Library with jest having version greater than 28. We should install jest-environment-jsdom separately
  npm install --save-dev jest-environment-jsdom

Where to write test cases while using Jest

- We should create a folder named '__tests__' to keep all our test files with extension as .js or .ts so that the Jest will consider it as a test file
- Another way is to make file name ends with either of the one in the list .test.js, .test.ts, .spec.js, .spec.ts so that the Jest will consider it as a test file
- Double underscore in the front and back (__test__) is known as dunder

Testing a basic JavaScript function

- Function to be tested sum.js

```
export const sum = (a, b) => {
  return a + b;
};
```

- Test file name starts with actual filename followed by test extension
  Ex:
  Filename – sum.js, Test Filename – sum.test.js

- The test(<String-desc>,<Callback>) method is used for testing which is having two arguments a string and a callback function
  - String - Which will have the description
  - Callback – Implementation of the test case

- In test case implementation we will be doing the assertions, but it is not mandatory
- If we write an empty test case or without assertion, it will always pass
- It's a best practice to write test cases with assertions
  Ex:

```
import { sum } from "../sum";
test("Sum function sould calculate the sum of two numbers", () => {
  const result = sum(2, 5);
  expect(result).toBe(7);
});
```

- We will be using *npm run test* command to run test

Unit Test - Test to check weather a component is loaded on to the DOM or not

- To unit test a component, as a first step we should render it into the jsdom
- We will be using the render() method of React Testing Library to render a component to the jsdom
- The 'screen' object of React Testing Library is used to get access to the component we rendered into the jsdom
- To get elements based on role in the jsdom using getByRole() method in jsdom
  Ex: const heading = screen.getByRole("heading")

- For assertion we will be checking the heading is in the DOM by using toBeInTheDocument() method
  Ex: expect(heading).toBeInTheDocument();

- To use JSX inside testing file we should install @babel/preset-react by using the command
  npm i -D @babel/preset-react

- Also, we should update the babel config file with the following code

```
module.exports = {
  presets: [
    ["@babel/preset-env", { targets: { node: "current" } }],
    ["@babel/preset-react", { runtime: "automatic" }],
  ],
};
```

- We should install @testing-library/jest-dom to use toBeInTheDocument() method
  npm i -D @testing-library/jest-dom
  Post installation we should import it as import @testing-library/jest-dom

```
import { render, screen } from "@testing-library/react";
import Contact from "../Contact";
import "@testing-library/jest-dom";
test("Should load contact us component", () => {
  render(<Contact />);
  const heading = screen.getByRole("heading");
  expect(heading).toBeInTheDocument();
});
```

- screen.getBy… method will return only one result, If we need to get all elements, we should use screen.getAllBy…
- The screen.get… methods will return JSX / React Elements basically JavaScript object / Virtual DOM / React fiber node
- We can negate the test case by using 'not' after expect()
  Ex:

```
expect(inputBoxes.length).not.toBe(3);
```

Grouping Test Cases

- The describe() method is used to group the test cases into a single block
- We can have multiple describe methods, It will also allow nesting ie: describe() inside describe()
  Ex:

```
describe("Contact us Page Test Cases", () => {
  test("Should load contact us component", () => {
    // test 1
  });

  test("Should load button inside Contact component", () => {
    // test 2
  });

  test("Should load input name inside Contact component", () => {
    //test 3
  });

  test("Should load 2 input boxes inside Contact component", () => {
    //test 4
  });
});
```

- The test() function is same as it() function
- Three common steps for unit testing
  - Render
  - Querying
  - Assertion

Unit Test – Having Redux and react-router-dom

- To unit test a component having Redux and Link (react-router-dom) component,
  - We should provide the store while rendering the component using Provider component to work Redux in unit testing

Ex:
<code with provider in render>

- We should provide the browser router while rendering the component using BrowserRouter component to work Link (react-router-dom) in unit testing

  Ex:
  <code with router provider in render>

- The screen.getByRole() is the best way to query elements, if we can’t do it with screen.getByRole() then only will use screen.getByText()
- If we are having multiple elements with same role, then we can pass additional config with .getByRole() method

  Ex:
  <code with config in getByRole()>

- If we want to match a portion of string in the DOM, then we can use getByText() with regex

  Ex:
  <getByText(/Cart/)>