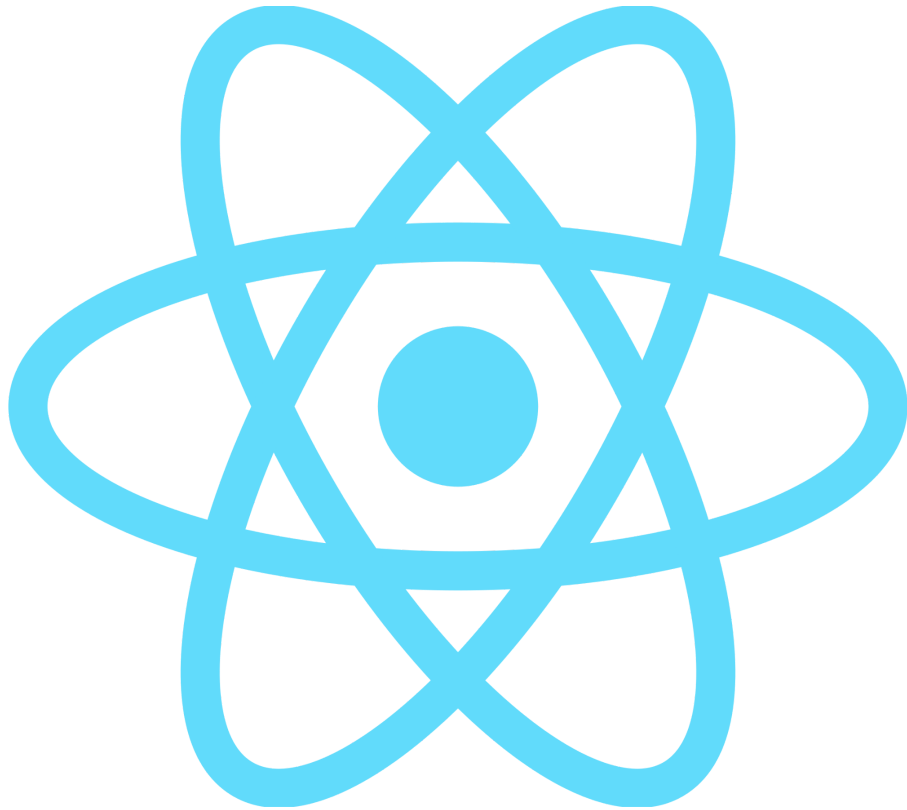


REACT JS

Notes



RUSHIL SHIVADE

All the JavaScript you need to know before learning ReactJS

1. Arrow Functions:

Arrow functions are a concise syntax for writing function expressions in JavaScript. They were introduced in ECMAScript 6 (ES6) and provide a more compact and expressive way to define functions compared to traditional function expressions.

So you should preferably use arrow functions cuz it'll be helpful in react.

```
<button
  onClick={() => {
    console.log("hello world");
  }}
></button>;
```

2. Ternary Operators

So in react, you have jsx files where u can simply write js code inside html so u obviously want your code to be short. The if else statement take up a lot of lines so alternative:

```
let age = 16;
let name = age > 10 ? "Pedro" : "Jack";
```

```
if(age>10) {
  Name = "Pedro"}
```

```
Else{
  Name = "Jack"}
```

This if else if converted to one single line.

```
let age = 16;  
let name = age > 10 && "Pedro";
```

if(age > 10){name = "Pedro"}

3. Objects

```
const person = {  
  name: "Pedro",  
  age: 20,  
  isMarried: false,  
};  
  
const person2 = { ...person, name: "Jack" };  
  
const names = ["Pedro", "Jack", "Jessica"];  
const names2 = [...names, "Joel"];
```

Const person represents an object that has key value pairs. Const person2 has a ...person - This is a spread operator which will copy the entire person object and then name: "Jack" will only modify the name in that object.

4. Map, filter and reduce methods

```
let names = ["Pedro", "Jessica", "Carol"];  
  
names.map((name) => {  
  return name + "1";  
});
```

Here, the names array will get modified to have names like Pedro1, Jessica1 and Carol1.

Can be used for manipulation and displaying.

```
let names = ["Pedro", "Jessica", "Carol", "Pedro", "Pedro"];

names.filter((name) => {
  return name !== "Pedro"
})
```

This will modify the array to have names except “Pedro”

1. **`map()` method**: The `map()` method creates a new array by applying a transformation function to each element of the original array. It iterates over each element and transforms it according to the logic provided in the callback function. The resulting array has the same length as the original array.

The `map()` method is commonly used when you want to perform a one-to-one transformation on each element of an array. It is useful when you want to derive a new array with modified values from the original array.

2. **`filter()` method**: The `filter()` method creates a new array containing elements from the original array that satisfy a specified condition. It iterates over each element and tests it against a provided condition in the callback function. The resulting array may have fewer elements than the original array or may be empty.

The `filter()` method is useful when you want to selectively extract elements from an array based on a specific condition. It creates a new array with elements that pass the condition.

In summary, `map()` is used for transforming each element of an array, while `filter()` is used for selecting elements from an array based on a condition. Both methods return a new array, leaving the original array unchanged.

The `reduce()` method is an array method in JavaScript that iterates over the elements of an array and accumulates a single value based on a reducer function. It is used to reduce an array to a single value by performing a specified operation on each element.

REACT JS

Starting out:

Creating a react app

`npx create-react-app .`

Will create a boiler plate for you with all the file structure.

To run the react app: `npm start`

So in the public folder, there's an index.html file that contains a single div of id = root.
So in src/app.js, we write all of our code and condense it in an app and then append it to the div so you're loading the entire website in one single file so no reloads.

Module - 2

```
function App() {  
  const name = "Rushil"  
  return <div className="App">{name}</div>;  
}
```

Everytime you want to show something on the page, like h1 in html or stuff like that, you use jsx, create a function and return the html code in it. Has something to do with components (will be explained later)

```
function App() {
  const name = <h1>Rushil</h1>
  return <div className="App">{name}</div>;
}
```

Since we're dealing with JSX, we can define variables that are html elements.

```
function App() {
  const name = <h1>Pedro</h1>;
  const age = <h2> 21</h2>;
  const email = <h2> pedro@pedro.com</h2>;
  const user = (
    <div>
      {name}
      {age}
      {email}
    </div>
  );
  return (
    <div className="App">
      {user}
      {user}
      {user}
    </div>
  );
}
```

As you can see, here is the code that makes it easy to use over and over again. You can define a UI inside of a variable.

COMPONENT: A Javascript function that returns some sort of UI. Like the function App() in the above image.

```
const GetName = () => {
  return "Pedro";
};

const GetNameComponent = () => {
  return <h1>Pedro</h1>;
};
```

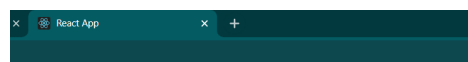
1st one is a js function whereas 2nd one is a component as it returns a UI (HTML element)

- Every component needs to start with a capital letter.

```
function App() {
  // const name = <h1>Rushil</h1>
  return (
    <User />
  );
}

function User() {
  return(
    <div className="App">
      <h1>Rushil</h1>
      <h2>9420282547</h2>
      <h2>rshivade02@gmail.com</h2>
    </div>
  )
}
```

We can define a component and then use it by writing <User /> in the App() because the App() component is the first component loaded onto the page.



Rushil

9420282547

rshivade02@gmail.com

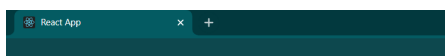
Makes the code look better.

In normal js functions, we pass in certain arguments right but in react components, you can't just pass in arguments. There's a thing called props.

Function Job(props){}

```
function App() {  
  // const name = <h1>Rushil</h1>  
  return (  
    <div className="App">  
      <Job salary={90000} position="Senior SDE" company="Amazon"/>  
      <Job salary={60000} position="Junior SDE" company="Amazon"/>  
      <Job salary={40000} position="Intern" company="Amazon"/>  
    </div>  
  );  
}  
  
function Job(props) {  
  return(  
    <div className="App">  
      <h1>{props.salary}</h1>  
      <h2>{props.position}</h2>  
      <h2>{props.company}</h2>  
    </div>  
  )  
}  
  
export default App;
```

In the App component, I pass in parameters like salary, position and company. These can then be used as props.salary later. We do this and replace arguments.



90000

Senior SDE

Amazon

60000

Junior SDE

Amazon

40000

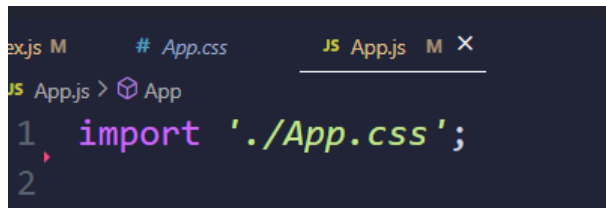
Intern

Amazon

MODULE - 3

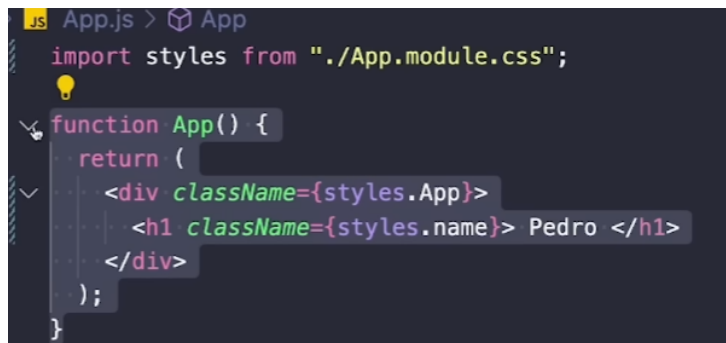
Ternary Operators, Lists and CSS

When dealing with CSS, you import in the app.js like this



```
ex.js M # App.css JS App.js M X
JS App.js > App
1 import './App.css';
2
```

Another way to do this:

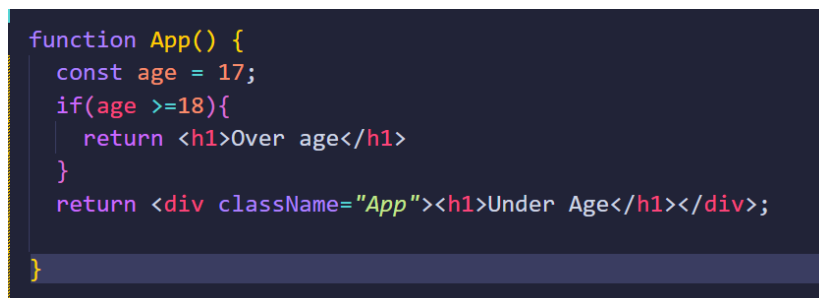


```
JS App.js > App
import styles from './App.module.css';

function App() {
  return (
    <div className={styles.App}>
      <h1 className={styles.name}> Pedro </h1>
    </div>
  );
}
```

You don't have to write in strings.

But the 1st one is mostly used.



```
function App() {
  const age = 17;
  if(age >=18){
    return <h1>Over age</h1>
  }
  return <div className="App"><h1>Under Age</h1></div>;
}
```

Replacing by ternary operator

```
function App() {  
  const age = 19;  
  
  return (  
    <div className="App">  
      {age >= 18 ? <h1>Over age</h1> : <h1>Under age</h1>}  
    </div>);  
}
```

Including CSS

```
function App() {  
  const age = 17;  
  const isGreen = false;  
  
  return (  
    <div className="App">  
      {age >= 18 ? <h1> OVER AGE</h1> : <h1> UNDER AGE</h1>}  
      <h1 style={{ color: isGreen ? "green" : "red" }}> THIS HAS COLOR </h1>  
  
      {isGreen && <button> THIS IS A BUTTON</button>}  
    </div>  
  );  
}
```

```
function App() {  
  const age = 19;  
  const shouldDisplay = true;  
  return (  
    <div className="App">  
      {age >= 18 ? <h1>Over age</h1> : <h1>Under age</h1>}  
      {shouldDisplay && <h1 style = {{color: "green"}}>Green is true</h1>}  
    </div>);  
}
```

Here, you are basically using JS code inside {} and display HTML elements, inside of which you can put in styles. Note: Any JS code that you have to write within HTML

tags must be written inside of curly braces.

- **Lists**

```
function App() {
  const names = ["Rushil", "Rutuja", "Ross", "Rachel"];

  return (
    <div className = "App">
      {names.map((name, key) =>{
        return <h1 key= {key}>{name}</h1>
      })}
    </div>
  )
}
```

Any JS that is written inside of an HTML element is written inside {}.

```
import './App.css';
import {Users} from './Planets';
```

```
function App() {
  const planets = [
    {name: "Mars", isGasPlanet: false},
    {name: "Earth", isGasPlanet: false},
    {name: "Jupiter", isGasPlanet: true},
    {name: "Venus", isGasPlanet: false},
    {name: "Neptune", isGasPlanet: true},
    {name: "Uranus", isGasPlanet: true},
  ];

  return (
    <div>
      {planets.map((planets) => {
        return <Users name = {planets.name} isGasPlanet = {planets.isGasPlanet}/>
      })}
    </div>
  );
}
```

```

1 export const Users = (props) => {
2   return(
3     <div>
4       <h2>{props.name}</h2> <h3>{props.isGasPlanet ? <h3>This is a gas planet</h3> : <h3>This is not a gas planet</h3>}
5       </h3>
6     </div>
7     { /* <h2>{props.isGasPlanet && <h3>{props.name}</h3></h2> */}
8   )
9 }
10
11

```

So the exercise was to loop through the planets object and display their names and if it's a gas planet, display that it is and likewise.

We can either create a component inside of the App.js or what's generally followed is to create another .js file and export it like done above. Over there, you use props. And in App.js, you return <Users /> inside of a map function so as to iterate through the object.

MODULE - 4

States in React, useState Hook

React only renders the component once so like if you set a variable age=0 and even if you have a button and the logic to increase the age variable, it won't do so cuz the rendering part has been done once.

So in react, we-

Import (useState) from "react";

A hook, as of now it is anything that begins with the word 'use'. This is a useState hook.

```

function App() {
  const [age, setAge] = useState(0);
}

```

This is how you declare a variable. Age is the name of the var. setAge is going to be a function that has the logic to modify the var and useState(0) -> 0 is the initial value of the var. Also, the function has to be named: setAge. set and then the name of the var.

```
function App() {
  const [age, setAge] = useState(0);

  const increaseAge = () =>{
    setAge(age+1);
  }

  return (
    <div className="App">
      {age}
      <button onClick={increaseAge}>Increase Age</button>
    </div>
  );
}
```

Parameter that you pass in to setAge is what will be changed. With this, react will know that the value has been changed so it'll re-render the page.

- An event in JS is used to grab information about the input.

```
function App() {
  const [text, setText] = useState("");
  const alterText = (event) =>{
    setText(event.target.value);
  }

  return (
    <div className="App">
      <input type="text" onChange={alterText}/>
      <br></br>
      {text}
    </div>
  );
}
```

More examples on States

```
function App() {
  const [showText, setShowText] = useState(true);
  const alterText = () =>{
    setShowText(!showText);
  }

  return (
    <div className="App">
      <button onClick={alterText}>Show/Hide text</button>
      {showText && <h1>My name is Rushil</h1>}
    </div>
  );
}

export default App;
```

Will show/hide the text as you click

the button.

```
function App() {
  const [changeColor, setChangeColor] = useState("black");
  const alterColor = () =>{
    setChangeColor(changeColor === "black" ? "red" : "black");
  }

  return (
    <div className="App">
      <button onClick={alterColor}>Change color of text</button>
      { <h1 style={{color: changeColor}}>My name is Rushil</h1>}
    </div>
  );
}
```

See how this alters the color between red and black as you click the button.

MODULE - 5

CRUD To-Do List

When you're using states in react, you can't just modify the value of a variable by writing say,

newTask = "Go to gym". You need to pass the new value in the setNewTask function.

```
div className="list" >
  {todoList.map((task) =>{
    return <div>
      <h1>{task}</h1>
      <button onClick = {() => deleteTask(task)}>X</button>
    </div>
  })}
```

Whenever you want to pass in arguments inside a function like above, you gotta use another function like done above to make things work.

Look at the TodoList code for understanding purposes.

MODULE - 6

Component Lifecycle, useEffect Hook

What exactly is a component lifecycle?

It's basically the events that happen through the period right from the birth of a component to the death of the component, from start to the end. Three stages of lifecycle:

1. Mounting: Mounting is basically when the component UI gets rendered or appears on the project.
2. Updating: Updating some stuff like props getting changed.
3. Unmounting: When the component UI exits the project and is no longer being displayed.

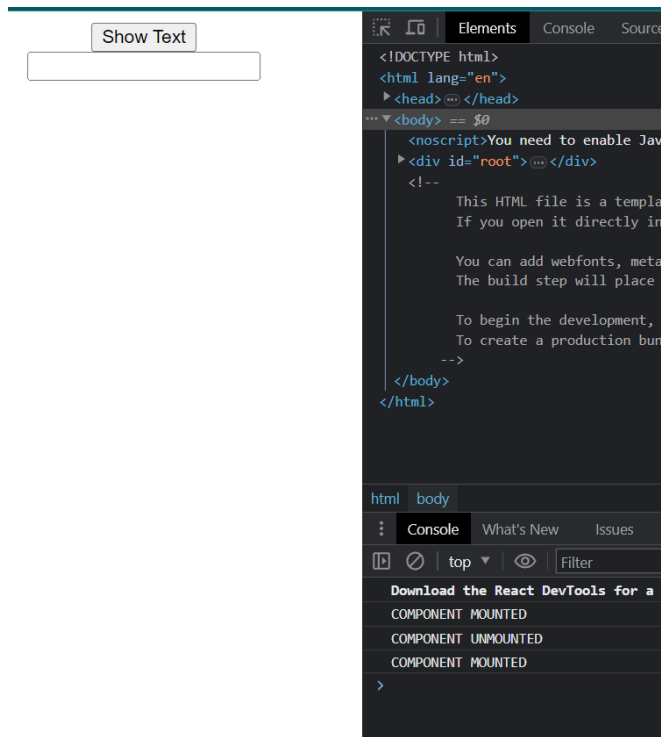
UseEffect allows us to do things in specific stages of the component lifecycle. For example if I want to make an API call after a link has been clicked so I can apply a useEffect to that link to load the API, like in the mounting stage. And you want to continue sending API requests throughout and then after the component disappears, you want to stop that request. So this can be done by UseEffect().

In React, the `useEffect` hook is used to perform side effects in functional components. It allows you to run code in response to certain events or conditions, such as when the component mounts, when specific props or state values change, or when the component unmounts.

```
export const Text = () => {  
  const [text, setText] = useState("");  
  
  useEffect(() => {  
    console.log("COMPONENT MOUNTED");  
  
    return () => {  
      console.log("COMPONENT UNMOUNTED");  
    };  
  }, []);  
}
```


If nothing is used and plain `useEffect()` is used, it will by default execute at the mounting stage, if you want in the updating stage, add `[]`. If you want in the unmounting stage, return it.

- React Strict mode - It was made so that we write better code. Basically if you enable it and `console.log("Mounted")`, it will log twice.



What happens is, it mounts the object, unmounts it and then mounts it again to see if the `useEffect()` hook has been used correctly or not.

- In `useEffect()`, you can specify a variable in the `[]` so everytime the variable changes, everything inside of the `useEffect()` will re-render.

MODULE - 7

Fetching data from APIs

```
import './App.css';

function App() {
  fetch("https://catfact.ninja/fact")
    .then((res) => res.json())
    .then((data) => {
      console.log(data);
    });
  return (
    <div className="App">
      <button> Generate Cat Fact</button>
      <p> </p>
    </div>
  );
}
```

Using Fetch API. But using the Axios library is preferable.

- If you want to get a request only once, use the `useEffect()` hook and like this, `useEffect(()=>{}, [])` so that it will only get executed at the mounting stage.

```
function App() {
  const [catFact, setCatFact] = useState('');

  useEffect(() => {
    Axios.get('https://catfact.ninja/fact').then((response) => {
      setCatFact(response.data.fact);
    })
  }, [])

  return (
    <div className='App'>
      <button>Generate cat fact</button>
      <p>{catFact}</p>
    </div>
  )
}
```

As of now, we'll turn off the react strict mode so it does not load the fact twice because you see, with react strict mode, the component first mounts, unmounts and then mounts again so the fact is fetched twice.

```
const func = (exc) => {
  Axios.get(`https://excuser-three.vercel.app/v1/excuse/${exc}`).then((res) => {
    setExcuse(res.data[0].excuse);
  })
}

// useEffect(()=>{
//   family();
//   office();
//   party();
// }, [0])

return (
  <div className='App'>
    <h1>Generate an excuse</h1>
    <button onClick={() => { func("office") }}>Office</button>
    <button onClick={() => { func("party") }}>Party</button>
    <button onClick={() => { func("family") }}>Family</button>
    <p>{excuse}</p>
  </div>
)
```

Exercise solution.

MODULE - 8

REACT ROUTER DOM

Used to navigate between multiple pages in a website. Basically uses routes to define which component to render when going to a particular route.

```

function App() {
  return(
    <div className='App'>
      <Router>
        <Navbar />
        <Routes>
          <Route path='/' element={<Home />}/>
          <Route path='/menu' element={<Menu />}/>
          <Route path='/contact' element={<Contact />}/>
          <Route path='*' element={
            <div>
              <h1> ERROR 404</h1>
              <br></br>
              <h1> PAGE NOT FOUND</h1>
            </div>
          }
        />
      </Routes>
    </Router>
  </div>
)
}

```

MODULE - 9

State Management, useContext Hook

Prop drilling: You pass in a prop in a function but don't use it, you simply just pass it on to some other function.

```
const TopComponent = () => {
  const [state, setState] = useState();
  return (
    <div>
      <MiddleComponent state={state} />
    </div>
  );
};

const MiddleComponent = (state) => {
  return (
    <div>
      <BottomComponent state={state} />
    </div>
  );
};
```

this which is by using some sort of
State Management solution

But this can have several issues so we have to avoid this.

State management solution is used to prevent prop drilling.

In the example presented, we passed in the setUsername function to the profile.js as a prop and then it wasn't used rather just passed onto changeProfile.js so that when you write something in the text box, the username will get changed entirely. This is prop drilling because we're only passing in arguments just to pass it onto other files.

So for this, we use Context API which allows us to include certain things inside of it so all the components included in that main component can have access to it without the use of props.

```

import { Profile } from "../Pages/Profile";
import { Navbar } from "../Navbar";
import { createContext } from 'react';

export const AppContext = createContext();

function App() {
  const [username, setUsername] = useState("Rushil");

  return (
    <div className='App'>
      <AppContext.Provider value={{username, setUsername}}>
        <Router>
          <Navbar />
          <Routes>
            <Route path="/" element={<Home username={username} />} />
            <Route path="/profile" element={<Profile username={username} setUsername={setUsername} />} />
            <Route path="/contact" element={<Contact />} />
            <Route path="*" element={
              <div>
                <h1> ERROR 404</h1>
                <br></br>
                <h1> PAGE NOT FOUND</h1>
              </div>
            } />
          </Routes>
        </Router>
      </AppContext.Provider>
    </div>
  );
}

```

```

import { useState } from "react"
import { useContext } from "react"
import { AppContext } from "../App"

export const ChangeProfile = () => {
  const {setUsername} = useContext(AppContext);

  const [newUserName, setNewUserName] = useState("");

  return (
    <div>
      <input onChange={(event) => {
        setNewUserName(event.target.value);
      }} />

      <button onClick={() => {
        setUsername(newUserName);
      }}> Change Profile</button>
    </div>
  )
}

```

MODULE - 10

React Query

React Query is a [ReactJS](#) pre configured data management library which gives you power and control over server-side state management, fetching, and caching of data, and error handling in a simple and declarative way without affecting the global state of your application.

For fetching data from APIs, we used Fetch or Axios library right but in latest react versions, the `useEffect()` is rendered twice for checking purposes implemented due to the react strict mode so we had to turn it off but it's not really a good practice to just get rid of strict mode so a better way to fetch data from API is by using react query.

Now, react query should be implemented at the highest level of components, basically the one which includes all the other components for ex the `App()` component.

```
import { QueryClient, QueryClientProvider } from
  '@tanstack/react-query';
```

```

import {useQuery} from "@tanstack/react-query";
import Axios from "axios";

export const Home = () => {
  const {data, isLoading, isError, refetch} = useQuery(["cat"], () => {
    return Axios.get("https://catfact.ninja/fact").then((response) => response.data);
  });

  if(isError) {
    return <h1>Error 404</h1>
  }

  if(isLoading) {
    return <h2>Loading...</h2>
  }

  return(
    <div>
      <h1>This is the Home Page</h1>
      <p>{data?.fact}</p>
      <button onClick={refetch}>Re Fetch</button>
    </div>
  )
}

```

Here is an example of using react query. "Cat" is the id that needs to be passed.

```

import { QueryClient, QueryClientProvider } from '@tanstack/react-query';

function App() {
  //const [username, setUsername] = useState("Rushil");

  const client = new QueryClient({defaultOptions:{
    queries:{
      refetchOnWindowFocus: true,
    },
  }},
  });
}

```

refetchOnWindowFocus. When set to true, it will refetch the data everytime you say change tabs and then come back, when set to false, won't update the data.

You don't have to use the useEffect() now, react query takes care of all of it.

MODULE - 11

Forms in React

To implement forms in react, we'll use two libraries:

1. React hook forms: Basic form functionalities
2. Yup: For input validation purposes.

```
import './App.css';
import {useForm} from 'react-hook-form';
export const Form = () => {
  const {register, handleSubmit} = useForm();

  const onSubmit = () => {
    console.log("Form submitted");
  }
  return (
    <form
      className="form-container"
      onSubmit={handleSubmit(onSubmit)}
    >
      <input type="text" placeholder="Full name" className="form-input" />
      <input type="text" placeholder="Email" className="form-input" />
      <input type="text" placeholder="Age" className="form-input" />
      <input type="password" placeholder="Password" className="form-input" />
      <input type="password" placeholder="Confirm Password" className="form-input" />
      <input type="submit" className="form-button" value="Submit" />
    </form>
  );
};
```

Here, we're using the useForm hook from the react-hook-form library. Now you have to declare two functions, register and handleSubmit. handleSubmit has to be executed whenever we click the submit button. We pass in the actual submit function inside of the handleSubmit. handleSubmit basically does the work before submitting the form.

Register function is used in each of these inputs to tell react hook form which inputs should part of the validation.

So basically we use it to assign a name to an input field so that the data submitted in the form is an object and it looks like:

```
{  
  
  Name: Rushil  
  
  Age: 21  
  
}
```

Name and Age are defined by this register function.

```
import './App.css';  
import {useForm} from 'react-hook-form';  
export const Form = () => {  
  const {register, handleSubmit} = useForm();  
  
  const onSubmit = (data) => {  
    console.log("Form submitted");  
    console.log(data);  
  }  
  return (  
    <form  
      className="form-container"  
      onSubmit={handleSubmit(onSubmit)}  
    >  
      <input type="text" placeholder="Full name" className="form-input" {...register("fullName")} />  
      <input type="text" placeholder="Email" className="form-input" {...register("email")} />  
      <input type="text" placeholder="Age" className="form-input" {...register("age")} />  
      <input type="password" placeholder="Password" className="form-input" {...register("password")} />  
      <input type="password" placeholder="Confirm Password" className="form-input" {...register("confirmPassword")} />  
      <input type="submit" className="form-button" value="Submit" />  
    </form>  
  );  
};
```

Using the register function and also, when you use handleSubmit, you can simply get the data in the onSubmit function like done above.

Form validation:

```
import './App.css';
import {useForm} from 'react-hook-form';
import {yupResolver} from "@hookform/resolvers/yup";
import * as yup from "yup";

export const Form = () => {

  const schema = yup.object().shape({
    fullName: yup.string().required(),
    email: yup.string().email().required(),
    age: yup.number().positive().integer().min(18).required(),
    password: yup.string().min(4).max(20).required(),
    confirmPassword: yup.string().oneOf([yup.ref("password"), null]).required() //null used to specify that this is the only
    value we'll need. yup.ref -> Basically lets you grab the input of one the input fields.
  })

  const {register, handleSubmit} = useForm({
    resolver: yupResolver(schema), //Integrating yup with react hook form. yupResolver is a package we import from the hookform
    resolvers.
  });
```

See how easy it is to validate stuff with yup. Also, look how the integration of the schema is done with the actual form.

CODE:

```
import './App.css';

import {useForm} from 'react-hook-form';

import {yupResolver} from "@hookform/resolvers/yup";

import * as yup from "yup";

export const Form = () => {

  const schema = yup.object().shape({

    fullName: yup.string().required("Your full name is required!"),

    email: yup.string().email().required("Email is required!"),

    age: yup.number().positive().integer().min(18).required("Please specify your age"),

    password: yup.string().min(4).max(20).required(),

    confirmPassword: yup.string().oneOf([yup.ref("password"), null], "Passwords do not
    match!").required() //null used to specify that this is the only value we'll need. yup.ref ->
```

Basically lets you grab the input of one the input fields.

```
    ))

    const {register, handleSubmit, formState: {errors}} = useForm({ //Errors are used to print
the error message.

    resolver: yupResolver(schema), //Integrating yup with react hook form. yupResolver is a
package we import from the hookform resolvers.

    });

    const onSubmit = (data) => {

        console.log("Form submitted");

        console.log(data);

    }

    return (

        <form

            className="form-container"

            onSubmit={handleSubmit(onSubmit)}

            >

                <input type="text" placeholder="Full name" className="form-input"
{...register("fullName")} />

                {errors.fullName && <p id = "errorMsg">{errors.fullName.message}</p>}

                <input type="text" placeholder="Email" className="form-input" {...register("email")} />

                {errors.email && <p id = "errorMsg">{errors.email.message}</p>}

                <input type="number" placeholder="Age" className="form-input" {...register("age")} />

                {errors.age && <p id = "errorMsg">{errors.age.message}</p>}

                <input type="password" placeholder="Password" className="form-input"
{...register("password")} />

                {errors.password && <p id = "errorMsg">{errors.password.message}</p>}

                <input type="password" placeholder="Confirm Password" className="form-input"
{...register("confirmPassword")} />
```

```
    {errors.confirmPassword && <p id = "errorMsg">{errors.confirmPassword.message}</p>}  
  
    <input type="submit" className="form-button" value="Submit" />  
  
  </form>  
  
  );  
  
};
```

MODULE - 12

Custom Hooks

You can create custom hooks to suit your needs by say making a hook for a functionality that you know you'll be making use of quite frequently. Three rules to make custom hooks:

1. Should start with 'use'.
2. Has to be inside of the highest level of the component.
3. Has to be inside of a component overall, not inside a function.

Hooks return the logic, not the jsx UI that is normally rendered by any react component that we make. Basic difference between custom hook and component.

So whenever you have a logic that you know will be reused again and again so rather than increasing the lines of code unnecessarily, create a hook for it.

```

import {useToggle} from "../useToggle";

function App() {
  const [isVisible, toggle] = useToggle();
  const [isVisible2, toggle2] = useToggle();

  return(
    <div className="App">
      <button onClick={toggle}>{isVisible?"Hide" : "Show"}</button>
      {isVisible && <h1>Hello</h1>}

      <button onClick={toggle2}>{isVisible2?"Hide" : "Show"}</button>
      {isVisible2 && <h1>Hello</h1>}
    </div>
  );
}

```

No need to write logic again and again.

MODULE - 13

TypeSafety, Typescript

```

7      <h1>Email: {props.email}</h1>
8      <h1>Age: {props.age}</h1>
9      <h1>This person {props.isMarried ? "is" : "is not"} MAR
10     {props.friends.map((friend) => (
11       <h1>{friend}</h1>
12     ))}
13   </div>
14 );
15 };
16
17 Person.propTypes = {
18   name: PropTypes.string,
19   email: PropTypes.string,
20   age: PropTypes.number,
21   isMarried: PropTypes.bool,
22   friends: PropTypes.arrayOf(PropTypes.string),
23 };
24

```

So when you pass in some parameters as props to some component, you obviously want to get the correct type of input to be passed only. So for this you use the prop-types library. You define the proptypes like done above and then the app won't break as it would if it wasn't used. It will still work but will throw an error in the console.

But you don't really need to do this if you're using typescript instead of javascript.

REDUX TOOLKIT

Redux and redux toolkit is used to handle state management easily.

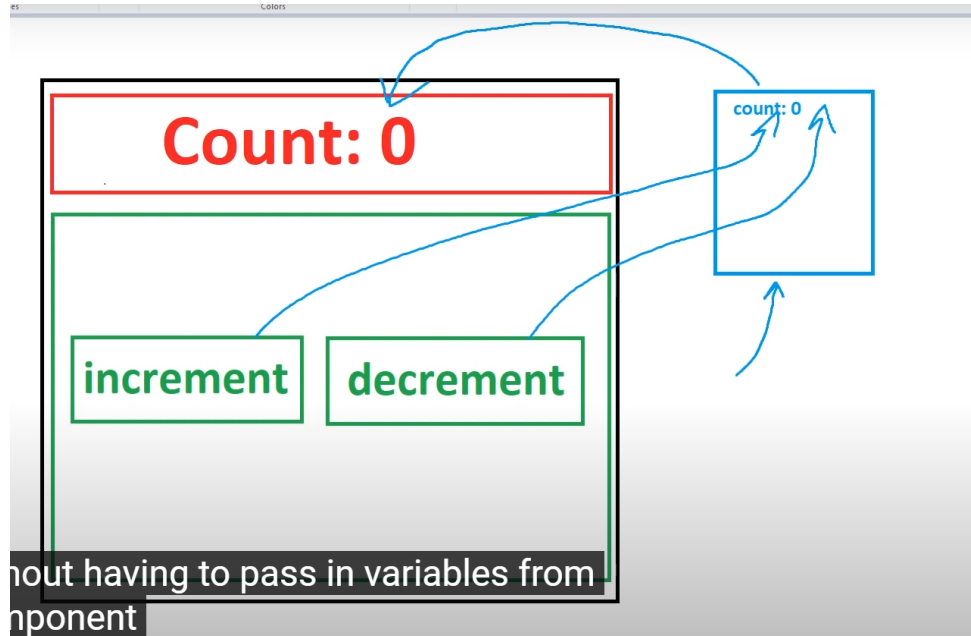
Redux is a popular state management library used with React.js applications. It helps you manage the state of your application in a predictable and efficient manner. Redux follows a unidirectional data flow pattern and is inspired by the Flux architecture.

At its core, Redux maintains the state of your application in a single JavaScript object called the "store." The store is responsible for holding the complete state tree of your application. Instead of directly modifying the state, you dispatch actions to the store. An action is a plain JavaScript object that describes a change in your application. It typically has a "type" property that specifies the type of action being performed.

Reducers are functions in Redux that specify how the state should be updated in response to an action. They take in the current state and an action, and return a new state. Reducers are pure functions, which means they do not modify the original state but produce a new state based on the input.

React components can access the state from the Redux store and update it by dispatching actions. Redux provides a mechanism called "connect" that allows components to connect to the store and receive the necessary state and dispatch functions as props.

By centralizing the state management in a single store and enforcing a strict unidirectional flow of data, Redux helps in writing predictable and maintainable code, especially for larger applications where managing state can become complex.



<https://redux-toolkit.js.org/>