React

It is a JS library that is specifically designed to create UI.

Benifits of using React:

* Development process is very fast using React.
* optimised code/app creation using React.
* React helps to write modular code.
* It has a component-based architecture.
* React uses Virtual DOM.
* Easy to use/debug.
* State management.
* Data communication using props.
* It provides hooks.

jsx: a code having html + js code.

Components:

It is just a fn. in React.

2 types:

- fn. based component
- class " (not used)

It is just a fn. that gives some html type code to render.

* In html, we used 'class = " "' keyword, whereas in jsx, we use 'className = " "'.

* Using react, we can create single-pg applications.

** img container:

In jsx file;

```
import mylmg from '/source'
<img src={mylmg} alt=''></img>
```

** To import css to a jsx:

In jsx file;

```
import "./source"
```

Props:

```
<usercard name="Aditya" />
<usercard name="verma" />
```

style="color:black"

In html, this is known as attribute and in jsx, it is called prop.

then; const usercard = (props) => {

```
return (
    <div classname='container'>
        <p>{props.name}</p>
    </div>
```

style={props.style}

eg)

Hook:

It is a method to use functionalities provided by React.

useState hook: helps to access the functionality provided by React, (State Management).

It provides 2 things:

→ State variable (Stores the state)

→ State function (fn. to change state)

var/state fn. initial state

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

```
return (
    <div>
        <btn onClick={() => {setCount(count+1)}}>
```

Passing Props as Fn. & children:

```
<card>
{ }  → text, element, nested elements ] children
</card>
```

{props.children} → to print all text, element etc.

```
<card children="text one">
</card>
```

// we can also pass a children like this.

How to pass a fn. from parent to child?

Let app : parent
button : child

In app;

```
const [count, setCount] = useState(0);
function handleClick (){
    setCount(count+1);
}
return (
    <div>
        <Button incrCount = {handleClick}/>
```

In button,

```
return (
    <button onClick = {props.incrCount}>
        Click Me
    </button>
```

State Lifting :

Case1) When we need to transfer data from child to parent.

Case2) 2 siblings want to access the same state.

In parent;
//create state app, manage, change state

```
const [name, setName] = useState(' ');
return (
  <div>
    <card name={name} setName={setN}/>
  </div>
```

In child;

```
return(
 <div>
 <input type="text" onchange={(e)=>
        props.setName(e.target.value)}/>
```

↑

Now whatever we change in this, it will also change in parent class.

Conditional rendering:

It is done using:

- → if - else
- → ternary operator
- → logical operator
- → early return (break)

Create 2 components, (children), one is loginbutton, and other is logoutbtn.

Now, in parent;

If-else

```
const [isLoggedIn, setLogIn] = useState(true);
if (isLoggedIn) {
    return (
        <LoggedOutBtn />
    )
}
else {
    return (
        <LoggedInBtn />
    )
}
```

Ternary

```
return (
    <div>
        {isLoggedIn ? <LoggedOutBtn /> : <LoginBtn />}
    </div>
)
```

Logical

```
return (
    <div>
        <div>
            {isLoggedIn && <LogoutBtn />}
        </div>
    </div>
```

Event handling in React:

In app.;

```
function handleClick(){
    alert("I am clicked");
}
return(                    (event)
  <button onClick={handleClick}>
        Click Me
  </button>
```

Q) What is immediate invocation?

```
<button onClick={alert("clicked")}>
    Click me
</button>
```

In this case, btn pe click krne se pehle hi alert aajayega.

To prevent this: use arrow fn.

II UseEffect Hook:

Used to generate side Effect.

eg) Event: (DOM content Load)
∴ Side Effect: (dB connection)

SYNTAX:

In app;

```
impore {useEffect} from 'React';
useEffect ( () => {
    first (sideEffect fn. / logic)
    return () => {
        second (clean-up fn.)
    }
}, [third])
```

↳ comma separated dependency list

Variation 1:

```
//runs on every render
useEffect ( () => {
    alert ("Runs on each render");
})
```

Variation 2:

```
//runs on only first render
useEffect ( () => {
    alert ("Runs on 1st render");
}, [])
```

Variation 3:

```
useEffect (() => {
    alert ("I will run every time when count updates");
}, [count])
```

Variation 4:

```
// multiple dependencies

useEffect (() => {
    alert ("Runs everytime when count or total is updated");
}, [count, total])
```

Variation 5:

```
// includes a cleanup fn:

useEffect (() => {
    alert ("count is updated");
    return () => {
        alert ("count is unmounted");
    }
}, [count])
```

III useContext hook:

Prop-drilling: it can be avoided by using useContext hook.

- create a context.
- provide
- consume

App
↳ childA
↳ childB
↳ childC

In app;

Step I const UserContext = createContext();

* create 3 childs linked with each other

Step II Wrap all the child inside a provider

```
return (
  <>
    <UserContext.Provider>
      <ChildA />
    </UserContext.Provider>
```

↳ (childB, childC too)

Step III Create and pass the value :

```
const [user, setUser] = useState ({name: "abc"});
<UserContext.Provider value={user}>
export {UserContext};
```

Step IV: Consume the value in the consumer:

In childC (let);

```
import {UserContext} from '../app'
const user = useContext (UserContext);
return (
  <div>
    user.name;
  </div>
```

* copy the react router DOM command

React - routing :

Route is simply a url / path.

I import { createBrowserRouter, RouterProvider } from "react-router-DOM";

Creation

II create router = createBrowserRouter(

```
[
  {
    path : "/",
    element : <Home />,
  },
  {
    path : "/about",
    element : <About />
  },
]);
```

this creates a route, that leads to home pg.

III In "app fn.";

```
return(
  <div>
    <RouterProvider router={router} />
  </div>
)
```

IV Now, you can create a component navbar.jsx, to access these links.

V In navbar.jsx;

```
<div>
  <ul>
    <li> Link to='/'> Home </Link>
    </li>
```

Similarly, create more.

Here, we don't use anchor tag, we use 'Link' or 'NavLink' tag.

* `<NavLink to='/' className={({isActive}) => isActive ? "active-link" : ""}>`
  `Home`
  `</NavLink>`

↳ Now, we can do easy CSS styling of 'active-link'.

VI How to handle parameters: useParam Hook

www.codehelp.in/id-"....."

parameter

↓ query parameter

Create:

```
{
  path: "/student/:id",
  element:
    <div>
      <Navbar />
      <ParamComp />
    </div>
}
```

V] Create new component : ParamComp.jsx;

```
const ParamComp = () => {
    const {id} = useParams();
    return (
        <div>
            Params : {id};
        </div>
    )}
```

VI] useNavigate hook :

In any component;

```
const navigate = useNavigate();
fn. handleClick () => {
        navigate ('/about');
}
return (
  <div>
     Home Pg.
     <button onClick={handleClick}>
        Move to about pg
     </button>
  </div>
)
```

VII Nested Routing:

We use the 'children' field.

```
{
  path: "/dashboard",
  element:
    <div>
      <Navbar />
      <Dashboard />
    </div>,
  children: [
    {
      path: '/courses',
      element: <Courses/>
    },
      create more
  ]
};
```

children path me '/' nhi lgayege!

Nested route

Now in dashboard component, in

```
return (
    <Outlet />  // just add this for children to run.
)
```

VIII To handle route error:

```
{
  path: "*",
  element: <NotFound />
}
```

Use react-hook-form package.

React hook form:

I const {
register,
handleSubmit,
watch,
formState: { errors }, isSubmitting }
} = useForm();

II Link an input field with form:-

return(
<form>

<label> First name: </label>
<input {...register('firstN')} />

III To submit a data:

return(
<form onSubmit = {handleSubmit((data) =>
console.log(data))}>

IV To check validation of input field:

<input {...register('firstN'}, {required: true,
minLength: 3 })} />

<..... minLength: {value: 3, message: "Min Len is 3"},
{errors.firstName && {errors.firstName.
message} }

* You can also use patterns in input field.

* To simulate any API call while submitting:

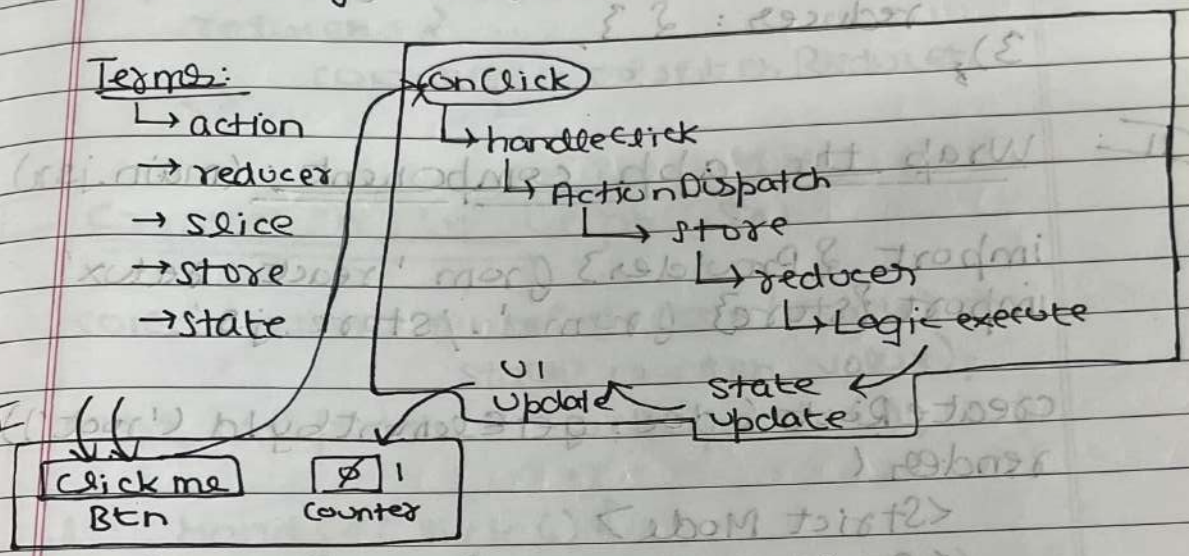
```
await new Promise (resolve) =>
                    setTimeout(resolve, 3000));
```

* To prevent multiple submitting of same form;

```
<input type='submit' disabled={isSubm-
                                itting} />
  value={isSubmitting? "Submitting" : "Submit"}
```

React - Redux Toolkit:

It helps us to finish the problem of prop drilling, by creating a centralised entity called 'store', where we manage the state. (data).



- **Action**: it is the wrap (object) of an event or (event + additional info.)
  ↳ payload (value).

- **Slice**: it contains the logic to maintain a feature of a state. (eg. initial state, reducer fn. etc) is a feature

- **Reducer**: it is a fn. that contains the entire logic to change / update the state.

- **Store**: it is a single source of truth that contains the state.

* Import redux module.

I Create a store: (store.js)

```
import {configureStore} from '@reduxjs/
                            toolkit',
export const store = configureStore({
    reducer: {},
});
```

II Wrap the app component: (main.jsx)

```
import {Provider} from 'react-redux'
import {store} from './store.js'

createRoot(doc.getElementById('root')).
render(
    <StrictMode>
    <Provider store={store}>
        <App />
    </Provider>
    </StrictMode>,
)
```

III Create a slice (for counter app): (counterSlice.jsx)

```
export const counterSlice = createSlice({
  name: 'counter',
  initialState: {value: 0},
  reducers: {
    incrementByAmount: (state, action) => {
      state.value += action.payload
    }
```

```
export const { incrementByAmount } =
                counterSlice.actions
export default counterSlice.reducer
```

IV Register the reducer to store: (store.js)

```
import ....
  reducer: {
      counter: counterReducer
  }
```

V Create a UI (app.jsx)

```
const count = useSelector((state) =>
                state.counter.value);
const dispatch = useDispatch();

fn handleClick() {
    dispatch(incrementByAmount());
}

return (
    Create UI
)
```

→ State var → change → re-render
→ useRef var → change → no re-render

Page:

useRef hook:

It is used in 2 cases:

case1) When a variable persists it's value across re-renders.

```
                          ↑initial value
let val = useRef(1);
                          ↳object value
fn. handleClick(){
    val.current = val.current + 1;
```

case2) It helps us to directly change any element from DOM.

```
// let btnRef = useRef();

fn. changeColor(){
// btnRef.current.style.backg-clr = "red";
}
return (
<div>
// <button ref={btnRef} onClick>
        Button 1
  </button>

  <button onClick={changeColor}>
      Change clr of 1st button
  </button>
</div>
```

useMemo hook:

useMemo
React hook ↵ ↳ Memorization

This hook prevents the un-necessary reuse of expensive tasks/operations/calculations.

useMemo (() ⇒ calcFunction, [dependencies]);

* It only memorizes the last value it stores.

---

useCallback hook:

(memorize)
It lets you cache a fn. definition b/w re-renders.

↳ un-necessary re-render of child component is prevented.

↓

In this case;

In childComponent.jsx;

```
const ChildComponent = React.memo(
        //entire component
)
```

↳ this will only re-render when the props change.

* React.memo has a limitation, it only avoids re-renders when a variable prop is passed. If a fⁿ. prop is passed, then it cannot avoid re-renders.

∴ Solution :

In app.jsx;

```
const handleClick = useCallback(() => {
    setCount(count + 1);
}, [count]);
```

↳ dependencies (if any)

* useCallback hook also helps to handle expensive operations.