SECTION-28

React Hooks:

React Hooks are simple JavaScript functions that we can use to isolate the reusable part from a functional component.These hooks can only used in functional components and in custom hooks.they start with ‘use’.

useState():

It always returns an array which contains, current state snapshot and function to update that current state.when state is changed,component re-renders.state services between re-renders.

-> setTimeout() gives you a reference, a pointer at that timer.that pointer is currently active timer.

useReducer():

->we write reducer function(which updates state based on type) outside of the component,bcz we don’t want to recreate function everytime component renders.reducer function takes two arguments such as current state and action(object to update state)
-> useReducer() takes a reducer function and initial state as arguments and returns an array that contains state and dispatch function.
-> when working with useReducer() , React will re-render the component whenever reducer returns new state.
-> dispatch function invokes reducer function to update state.

useContext():

->in a context-api file , a context(store) is created with createContext.this should be outside of react component.
-> write a react component that returns <store_name.Provider>{props.children</store_name.Provider>.this component defines state and functions contained in store.

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

export const AuthContext = React.createContext({
  isAuth: false,
  login: () => {}
});

const AuthContextProvider = props => {
  const [isAuthenticated, setIsAuthenticated] = useState(false);

  const loginHandler = () => {
    setIsAuthenticated(true);
  };

  return (
    <AuthContext.Provider
      value={{ login: loginHandler, isAuth: isAuthenticated }}
    >
      {props.children}
    </AuthContext.Provider>
  );
};

export default AuthContextProvider;
```

->when using store, useContext() is used.To provide data to all components component is wrapped at top level.In this case,`AuthContextProvider` is wrapped at top level.

-> **useCallBack():**
useCallBack() hook is to avoid recreation of function.It catches function between re-renders.

-> **useMemo():**
useMemo() hook is is to avoid recreation of values.It catches values between re-renders.

SECTION-29

->Creating react app 2 ways
1) npx create-react-app App_name
2) Npm create vite
   - -npm install
   - -npm run dev

-> code written in react is not browser friendly.but the project setup created with npx create-react-app or npm create vite transforms react code to browser friendly.
-> package.json file contains all the third party packages used in our application
-> outputting dynamic values with help of '{}'
-> Event listeners start with 'on'.

->functions are objects.

-> event.target.value => event.target means elements like input,textarea ,etc.event.target.value is to access value of that element

-> when using useState, then component re-rendrs.In this case,DOM is updated with only changes occurred ,instead of entire dom recreation.React compares different snapshots and then update when changes occurred.

-> when a component is used as wrapper to other components, It will have children props.

Like Modal.js:

```
const Modal=(props)=>{
  return<>
  <div/>
  <dialog>{props.children}</dialog>
  </>
}
```

poster.js

```
const poster=()=>{
  return
  <Modal>
    <poster/>

  </Modal>
}
```

We need to specify props.children to show where to use components that are wrapped around Modal component.
-> Inorder to use react router, need to install "react-router-dom".

->**Layout routes**
  If anything need to render in all paths then we use layout routes.Like some paths are nested in a single path using children property.

```
const route=createBrowserRouter([
  {path:'/',
   element:<RootLayout/>,
   children:[
    {path:'/',element:<something/>},
    {path:'/products',element:<something/>},

   ]
}
])
```

```
const RooteLayout=()=>{
  return<>
  <Header/>
  <Outlet/>
  </>
}
```

<Outlet> component tells where to render child paths.

-> If we use <a> then it sends request to backend, to serve this entire react app again.To avoid that use,<link> and <Nav>.Which prevents browser's default bhaviour to send request to backend.

-> when there are more than one <Form> then we can use 'method' property.with this we can know which <Form> invoked same action in same route.
-> 'redirect' is used in loaders and actions to return result of calling action and loader functions.redirect generates a response object which in the end returned by the action.redirect is to simply redirect to specified path after action method execution.like redirect('/').

-> 'gatsby.com' for react another library for react.
-> 'react native' for mobile applications.
-> 'next js' for server side rendering which is framework for react.It also improves search engine optimization.
-> 'Advanced redux' section.

Testing:

-> To test react apps we can use react testing library.but drawback of it is that it cannot able to test child components.for that we have 'enzyme'.

-> screen is the way to interact with the component we render
-> there will be more methods on screen like getByText,queryAllByText,findByText etc.90% we will be using 'get' instead of 'find' and query.
-> we have to mimic the interaction with user as much as possible inside of our tests.
-> in order to use Element.not.toBeInTheDocument(),queryByText() works.getByText() doesnot work.

->router version:
-> react testing library version:
-> fit-content.

Typescript:

->props:

```
<greetings name="Devi"/>
```

```
const greetings=(props:{name:string})=>{

  return <h1>hi {props.name}</h1>
}
```

(or)

```
<greetings name="Devi" value={10}/>
```

```
Type greetings={
Name:string,
Value:number
}

const greetings=(props:greetings)=>{
  return <h1>hi {props.name}</h1>
}
```

(or)

```
<greetings name="Devi" value={10}/>
```

```
const greetings:React.FC<{name:string}>=(props)=>{

  return <h1>hi {props.name}</h1>
}
```

```
-> string => name:string
   Boolean => val:boolean
   Object => obj={FirstName:string,LastName:string}
   Number => val:number
   Array of objects => names:{FirstName:string,LastName:string}[]
   Array of number => values:number[]
   Array of strings => names:string[]
   Array of boolean => values:boolean[]
```

-> children prop:

```
  type oscar={name:string,
   children:string}
```

-> If the type of children is component(component inside another component):

```
  type oscar={
        children:React.ReactNode
        }
```

```
->  const oscar={
          name:string,
          messageCount:number,
          }
```

There are some situations where don’t pass values to components,in that case we get error.like

```
  <greetings name="hi"/>
```

Here only name is sent,messageCount is not sent.so we will get error.To avoid that use '?'.

```
const oscar={
          name:string,
          messageCount?:number,
            }
```

To give default values, in component destructure messageCount and assign defaultvalue.

```
const {messageCount=0} = props;
```

-> Event props:

```
   type ButtonElement={
OnClickHandler:(event:React.MouseEvent<HTMLButtonElement>)=>void
}

 type InputElement={
OnChangeHandler:(event:React.ChangeEvent<HTMLInputElement>)=>void
}
```

->style props

```
  type style={
    style :CSSProperties
}
```

-> when there are lot more types then that can be kept in a separate file and export it.import in required file.
-> try to reuse the types.
-> useState() hook
   -> type inference means javascript takes type based on default values provided to useState().
useState(false)
   ->

```
 type AuthUser={
          Name:string,
          Email:string
          }
      const [user,SetUser]=useState<AuthUser | null>(null);
```

-> useState type assertion:

```
 const [user,SetUser]=useState<AuthUser>({} as AuthUser);
```

-> useReducer() hook:

-> useReducer is used when next state is dependent on previous state.used for complex state management

```
   -> const StateStruc={ count:number}
      const actionStruc={type:string,payLoad:number}
      function reducerFunc(state:StateStruc,action:actionStruc){}
```

->If we only want to accept two strings then we can give like “increment”|”decrement”.

-> suppose “increment” and “dectrement” requires payload and “reset” doesnot require payload then we can define.

```
type UpdateState={
     type:”increment” | “decrement”,
     Payload:number
    }

type ResetState={
       type:”Reset”
        }

type counterAction=UpdateState | ResetState;
```

counterAction is used at action.