Section-6(Styling react components)

styled components:

It is a third party library, a very popular one.which allows us to set pre-styled components with their own scoped styles

inline styles:

```
<input type="text" onChange={goalInputChangeHandler} value={enteredValue}
style={{borderColor: !isValid ? 'red' : 'black',background: !isValid ? 'salmon' :
'transparent'}}  placeholder={isValid===false ? 'eneter valid text' : 'enter text'}/>
```

Dyanamically adding classes:

in css file:

```
.form-control.isValid input{
  background-color: rgb(233, 167, 167);
  border-color: red;
  }

.form-control.isValid label{
    color:red}
```

In js file:

```
<div className={`form-control ${!isValid ? 'isValid' : ''}`}>
<label>Course Goal</label>
<input type="text" onChange={goalInputChangeHandler} value={enteredValue}
placeholder={isValid===false ? 'eneter valid text' : 'enter text'}/>
</div>
```

1) Regular css will not have particular scope.they are accessible to all files.It might be problem when two files are using same class name.to avoid it there are two approaches.

-> use a package called styled components.

Styled components is a package that helps you build components which have certain styles attached to them.where the styles really only affect the components to which they were attached and not any other components.

```
import styled from 'styled-components';

const Button= styled.button``;
```

This is called attacked template literal.styled is object and button is method here.between back ticks we can add any number of styles we want.this will create a button with styles that we provided for it.

```
const Button= styled.button`

  font: inherit;
  padding: 0.5rem 1.5rem;
  border: 1px solid #8b005d;
  color: white;
  background: #8b005d;
  box-shadow: 0 0 4px rgba(0, 0, 0, 0.26);
  cursor: pointer;


&:focus {
  outline: none;
}

&:hover,
&:active {
  background: #ac0e77;
  border-color: #ac0e77;
  box-shadow: 0 0 8px rgba(0, 0, 0, 0.26);
}

`;
```

No need to use class names. For pseudo selectors ‘&’ is used.’&’ is like class name here.
psuedo selectors means :
.class_name:hover{
};

We can also pass props to dynamically change styles.

```
const FormControl=styled.div`
margin: 0.5rem 0;
& label {
  font-weight: bold;
  display: block;
  margin-bottom: 0.5rem;
  color:${props =>(props.Isvalid) ?'red' : 'black'};
}

& input {
  display: block;
  width: 100%;
  border: 1px solid ${props=>(props.Isvalid) ? 'red' : 'black'};
  background: ${props =>(props.Isvalid) ? 'rgb(233, 167, 167)': 'transparent'}
  font: inherit;
  line-height: 1.5rem;
  padding: 0 0.25rem;
}
```

```
 <form onSubmit={formSubmitHandler}>
      <FormControl Isvalid={!isValid}>
        <label>Course Goal</label>
        <input type="text" onChange={goalInputChangeHandler}
value={enteredValue} placeholder={isValid===false ? 'eneter valid text' : 'enter
text'}/>
      </FormControl>
      <Button type="submit">Add Goal</Button>
    </form>
```

We can add media query to fit button in all screens.

```
const Button= styled.button`
  width:100%;
  font: inherit;
```

```
  padding: 0.5rem 1.5rem;
  border: 1px solid #8b005d;
  color: white;
  background: #8b005d;
  box-shadow: 0 0 4px rgba(0, 0, 0, 0.26);
  cursor: pointer;

@media (min-width: 600px){
    width:auto;
  }
```

Here in @media (min-width:600px) width is default width.it will change width size when screen size is less than 600px. It use above style to apply. So it uses width:100% from above styles.

-> second approach is css modules.

```
import styles from './Button.module.css'

const Button = props => {
  return (
    <button type={props.type} className={styles} onClick={props.onClick}>
      {props.children}
    </button>
  );
};

export default Button;
```

Here style becomes an object. And can use properties like 'styles.button'.

css modules It takes css classes and css files and basically changes those classes names to be unique.

We can add media query like this:

```
@media (min-width:600px){
  .button{
    width:auto;
  }
}
```

Section-7(Debugging react apps)

->react components name always should start with capital letters.
<Expense>.
-> while returning jsx code in a component,it should be wrapped in a single parent component.
->when we get syntax error,it is visible in terminal and also visible in browser.so react carefully and try to correct it
-> when there is logical errors they are visible in console. We need to solve by following through chain.Identify where logic is wrong.
->use break points to understand the process of execution.

When you don't need the `else` branch, you can also use a shorter logical `&&` syntax:

```
<div>
  {isLoggedIn && <AdminPanel />}
</div>
```

Section-8

```
parseInt(event.target.value, 10)
```

For converting string to number.

(or)

```
+enteredAge
```

This also work.

Section-9

Limitations of JSX

Jsx is a code we return in our component which in end will be rendered to the real dom by a react.
->we cannot return more than one 'root' jsx element(we cannot also store more than one 'root' jsx element)

Solution:
  -Enclose jsx codes inside
  -enclose in an array.but if we use array then we need to add keys as well.

Fragments:

It's an empty wrapper component.it doesn't render any real html element to DOM.But it fulfills 'react/jsx' requirements.

```
return(
<React.Fragment>
<AddUser>
</AddUser>
</React.Fragment>
)
```

It helps to write cleaner code.It reduces unnecessary html elements on final page.simply it can be <> and </>

React Portals:

It also similar to react Fragment

Section-10

useEffect:

useEffect runs on every render. That means that when the count changes, a render happens, which then triggers another effect.(invokes useEfffect after return).

1. No dependency passed:

```
useEffect(() => {
  //Runs on every render

});
```

2. An empty array:

```
useEffect(() => {

  //Runs only on the first render

}, []);
```

If we don't specify dependencies then function will execute only once.

3. Props or state values:

```
useEffect(() => {

  //Runs on the first render

  //And any time any dependency value changes

}, [prop, state]);
```

When we specify dependencies ,like prop ,state function inside useEffect executes everytime prop and state changes.

Side effects:whenever we have an action that should be executed in response to some other action

useEffect Does Not Actively “Watch”

Some frameworks are *reactive*, meaning they automatically detect changes and update the UI when changes occur.

React does not do this – it will only re-render in response to state changes.

useEffect, too, does not actively “watch” for changes. When you call `useEffect` in your component, this is effectively queuing or scheduling an effect to *maybe* run, *after* the render is done.

After rendering finishes, `useEffect` will check the list of dependency values against the values from the last render, and will call your effect function if any one of them has changed.

Debouncing:

Debouncing user input.we want to make sure we are not doing something with it on every keystroke,but once the user made a pause during typing.

Cleanup function:

It does not run before very first side effect execution.It runs before every new side effect function execution.

cleanup function **saves applications from unwanted behaviors like memory leaks by cleaning up effects**.

setTimeout:

Executes after specified period of time.

```
useEffect(() => {
    const identifier = setTimeout(() => {
      console.log("useeffect");
      setFormIsValid(enteredEmail.includes("@") && enteredPassword.length > 6);
    }, 500);
    return () => {
      console.log("clean function");
      clearTimeout(identifier);
    };
  }, [enteredPassword, enteredEmail]);
```

->code that changes state is called as a side effect.

useReducer:

useReducer is used as a replacement for useState() if you need more complex state management.

The useReducer Hook returns the current state and a dispatch method.

Props is for configuration. react context is not optimized for high frequency changes.

Rules of Hooks :

1)Only call React Hooks in React Functions ->React Component Functions /Custom Hooks (covered later!)

2)Only call React Hooks at the Top Level ->Don’t call them in nested functions/ Don’t call them in any block statements + extra,

unofficial Rule for useEffect(): ALWAYS add everything you refer to inside of useEffect() as a dependency

```
 z-index: 10;
overflow: hidden;
object-fit: cover;
  transform: rotateZ(-5deg) translateY(-4rem) translateX(-1rem);
 position: fixed;
.bump {
    animation: bump 300ms ease-out;
  }


  @keyframes bump {
    0% {
      transform: scale(1);
    }
```

```
    10% {
      transform: scale(0.9);
    }
    30% {
      transform: scale(1.1);
    }
    50% {
      transform: scale(1.15);
    }
    100% {
      transform: scale(1);
    }
  }
```

z-index:

The z-index property accepts a numerical value, and the higher the value, the closer the element is to the top of the stacking order. Elements with a higher z-index value will appear in front of elements with a lower z-index value.

It's important to note that the z-index property only works on elements that have a position value of "relative", "absolute", or "fixed". Elements with a position value of "static" cannot have a z-index value.

Let's say we have two HTML elements on a web page, a button and a div container. The div container has a z-index of 10, and the button has a z-index of 20. This means that the button will appear in front of the div container because it has a higher z-index value.

Backdrop/overlay :

The backdrop is a simple element that is positioned behind the modal or dialog, typically covering the entire screen. It is usually semi-transparent, allowing the user to see the content behind it, but not interact with it.

Modal:

A modal is usually displayed on top of the rest of the page, with a semi-transparent overlay (known as a backdrop) covering the rest of the page to visually indicate that the user cannot interact with the underlying content while the modal is active.

Reduce:

Reduce is a method which transforms an array of data into a single value

```
reduce((currentValue,item)=>{
Return currentValue+item.value;
},0);
```

It takes two arguments

1.function-> takes two arguments

    1.for first time it will be initial value(0 in this example).in later iterations it will be result of previous execution.

    2.values of array.

2.initial value

useRef:

`useRef` is a React Hook that lets you reference a value that’s not needed for rendering.

```
const ref = useRef(initialValue)
```

`useRef` returns a ref object with a single `current` property initially set to the initial value you provided.

On the next renders, `useRef` will return the same object. You can change its `current` property to store information and read it later. This might remind you of state, but there is an important difference.

Changing a ref does not trigger a re-render. This means refs are perfect for storing information that doesn't affect the visual output of your component. For example, if you need to store an interval ID and retrieve it later, you can put it in a ref. To update the value inside the ref, you need to manually change its `current` property:

By using a ref, you ensure that:

- You can **store information** between re-renders (unlike regular variables, which reset on every render).
- Changing it **does not trigger a re-render** (unlike state variables, which trigger a re-render).
- The **information is local** to each copy of your component (unlike the variables outside, which are shared).

Changing a ref does not trigger a re-render, so refs are not appropriate for storing information you want to display on the screen. Use state for that instead.

Section-12:

->whenever state,props,context, of a component changes that component is re-evaluated by react.

->React DOM is interface to the web

->react cares about:

1.props(data from parent component)

2.state

3.context(component wide data)

Components using this concepts are updated by react.React checks whether this component now wants to draw something new onto the screen.React will let the react DOM know about that so,that react DOM brings that new screen

->Re-evaluating components!== Re-rendering the DOM:

1.components: re-evaluating whenever props,state,context changes.react executes component functions

2.RealDOM: changes to the real DOM are only made for differences between evaluations.new snapshots will be added to real DOM.rest will not re-render.

->when state changes entire component is re-evaluated , when props changes that particular component is also re-evaluated.

App.js:

return(<Button show={show}>)

Here show changes Button component is also re-evaluated.

If props doesnot change but parent component changes then also child component is re-evaluated.since it is part of parent component.parent component has child components and their functions.

->React.memo() helps to optimize functional components

It tells react that for this component which it gets as argument,react should look at props this component gets and check new value for all these props and compare to previous value those props got and only if value of a prop changed,the component should be re-evaluated and re-executed.if parent component is changed and if props are not changed then child component will not re-execute

->React.memo() works for props(premitive values)like boolean.

React.memo actually compares with previous values so,premitives can be comparable like

*props.show===previous.prop.show

But functions are not like that.

Suppose : <Button show={show}> (boolean memo works here)

: <button onClick={clickHandler}>(function memo doesn't work here)

useCallback:

It allows us to store function across component executes.so it allow us to tell react that we want to save a function and this function should not recreated with every execution.

In other words, `useCallback` caches a function between re-renders until its dependencies change.

useCallback(()=>{},[]);

To cache a function between re-renders of your component, wrap its definition into the

`useCallback` Hook:

//

Const result=useCallback(()=>{

if(allowToggle){

SetShowParagraph((prevShowPara)=>!prevShowPara);

}

},[allowToggle]);

//

In the above example based on allowToggle value it should be recreated when allowToggle changes.if not allowToggle values will be same value when function created for first time.it will not take new snapshot of allowToggle value.

Usememo:

Usememo is just like a usecallback.useCallback is for functions where as usememo is for values.

During initial rendering, useMemo(compute, dependencies) **invokes compute , memoizes the calculation result, and returns it to the component**. If the dependencies don't change during the next renderings, then useMemo() doesn't invoke compute , but returns the memoized value

Class based components:

We use render method in class based components.

Class Product extends Component{

render(){

Return <h2>A product</h2>

}

}

->In class based components state is always an object.object should have state as a property.

->updating states in class based components merges but in function based components it overrides.

->error boundaries are used to avoid crashing of application just like try,catch.

Section-14:

->don’t connect react directly to database.It might be not secure.we can view code through developer tools.

->Instead use a backend server side language and this backend application is talking to database.react then communicate with backend API.backend code runs on different server so this code is not visible to users.

->use fetch API that is build into browsers it allows us to fetch data and also send data

fetch('URL'{});

-> url is url of API(to get data) and {} object is to add extra body,headers and methods.if dont specify that object then by default it is get method.

->json() is to convert json object into javascript object.

->JSON.stringify() is to convert javascript object to json.this is converted because to backend API data need to be send in that format.

->if we use .then() then we use catch().if we use asyn then we use try and catch.

->fetch() will not treat errors(200,404 ect) as a real errors.we need to write code for that.axios package treats those as a real errors.

Section-15

->React hooks are used in functional components and in custom hooks.

-> **If you have one or multiple React hooks that will be used at multiple locations in a code**,

-> Custom hooks are nothing but a javascript function. When this function is used in different components gets a different unique state to each component. only logic is shared between components, not the state.

-> Custom hooks start with 'use' just like built-in hooks.

-> Custom hooks can return anything.

-> If custom hooks use states and if they are updated then the components which use this custom hook are implicitly using those states and this component re-renders

->if we want to update state used in custom hook then define a function in that custom hook and invoke that function in component where this hook is used.we can just invoke it through function pointer.

SECTION-16

-> When a form is submitted, it automatically sends HTTP request to the server.during development server that we use only serves html,js pages. It also restarts the entire application.We would loose our state.to prevent that default behavior we use

event.preventDefault();

->onBlur **triggers when focus is lost from the input element in context**.

SECTION-17

->useEffect should return cleanup function and should not return a promise.so use async inside useEffect.

->fetching data from db is asynchronous task.it might take some time to fetch data.a function always returns a promise.

**********Promises(JS)**********

Promise:

A promise is a special JavaScript object that links the “producing code” and the “consuming code” together.

```
let promise = new Promise(function(resolve, reject) {
```

```
// executor (the producing code, "singer")

});
```

The function passed to `new Promise` is called the *executor*. When `new Promise` is created, the executor runs automatically. It contains the producing code(code which usually takes time) which should eventually produce the result.

the executor should perform a job (usually something that takes time) and then call `resolve` or `reject` to change the state of the corresponding promise object.

Its arguments `resolve` and `reject` are callbacks provided by JavaScript itself. Our code is only inside the executor.

When the executor obtains the result, be it soon or late, doesn't matter, it should call one of these callbacks:

- `resolve(value)` — if the job is finished successfully, with result `value`.
- `reject(error)` — if an error has occurred, `error` is the error object.

So to summarize: the executor runs automatically and attempts to perform a job. When it is finished with the attempt, it calls `resolve` if it was successful or `reject` if there was an error.

The `promise` object returned by the `new Promise` constructor has these internal properties:

- `state` — initially `"pending"`, then changes to either `"fulfilled"` when `resolve` is called or `"rejected"` when `reject` is called.
- `result` — initially `undefined`, then changes to `value` when `resolve(value)` is called or `error` when `reject(error)` is called.

The properties `state` and `result` of the Promise object are internal. We can't directly access them. We can use the methods `.then`/`.catch`/`.finally` for that. They are described below.

The first argument of `.then` is a function that runs when the promise is resolved and receives the result.

The second argument of `.then` is a function that runs when the promise is rejected and receives the error.

For instance, here's a reaction to a successfully resolved promise:

```
let promise = new Promise(function(resolve, reject) {
  setTimeout(() => resolve("done!"), 1000);
});

// resolve runs the first function in .then
promise.then(
  result => alert(result), // shows "done!" after 1 second
  error => alert(error) // doesn't run
);
```

If we're interested only in successful completions, then we can provide only one function argument to `.then`:

```
let promise = new Promise(resolve => {

  setTimeout(() => resolve("done!"), 1000);

});



promise.then(alert); // shows "done!" after 1 second
```

If we're interested only in errors, then we can use `null` as the first argument: `.then(null, errorHandlingFunction)`. Or we can use `.catch(errorHandlingFunction)`, which is exactly the same:

```
let promise = new Promise((resolve, reject) => {

  setTimeout(() => reject(new Error("Whoops!")), 1000);

});



// .catch(f) is the same as promise.then(null, f)


promise.catch(alert); // shows "Error: Whoops!" after 1 second
```

The call `.catch(f)` is a complete analog of `.then(null, f)`, it's just a shorthand.

Async/await:

The `async` keyword before a function has two effects:

1. Makes it always return a promise.
2. Allows `await` to be used in it.

The `await` keyword before a promise makes JavaScript wait until that promise settles, and then:

1. If it's an error, an exception is generated — same as if `throw error` were called at that very place.
2. Otherwise, it returns the result.

Together they provide a great framework to write asynchronous code that is easy to both read and write.

With `async/await` we rarely need to write `promise.then/catch`, but we still shouldn't forget that they are based on promises, because sometimes (e.g. in the outermost scope) we have to use these methods. Also `Promise.all` is nice when we are waiting for many tasks simultaneously.

SECTION -18

->Redux: A state management system for cross component or app-wide state.

->Redux is not react specific .It can be used in any javascript project.

-> react context and react redux are similar but some disadvantages are there for react context.

->react context might be complex for large applications.

->performance.its performance is not good if data changes a lot.

->Working of Redux:

It has one central data store in our application.only one data store for all state for entire application so that we can use it from inside of our components.components setup subscriptions to our central store.whenever data changes store notifies components.so components can get data they need and can use it.components cannot manipulate datainside store.they can only get data from store.to manipulate data in store redux use

reducer function(example: function that reduces list of numbers into sum).components and dispatcher communicates like -> components dispatch actions(components trigger certain actions).action is nothing but simple javascript object,which describes kind of operation reducer should perform.therefore redux forward actions to reducer and operation is performed by reducer.reducer replace existing state with new state in central store after updation, components get notified and components update their UI.

-> Execute command 'npm init -y' to get package.json file.this file is to install third party packages.

->Automatically Generated Files: node_modules and package-lock. json. When you first install a package to a Node. js project, **npm automatically creates the node_modules folder to store the modules needed for your project and the package-lock**.

->inorder to use redux we need to import it.since we are using nodejs,import statement look different for importing third party package.

Const redux=request('redux')

This is a default syntax of nodejs to install third party packages.

->reducer function always take two arguments,one is previous state and another is action.it always outputs a new state object.it is a pure function,there is no side effects.same input leads to same output.

Usage:

```
const redux = require("redux");

const CounterReducer = (state = { counter: 0 }, actions) => {
```

```
  if (actions.type === "increment") {
    return {
      counter: state.counter + 1,
    };
  }
  if (actions.type === "decrement") {
    return {
      counter: state.counter - 1,
    };
  }
  return state;
};

const store = redux.createStore(CounterReducer);
console.log(store.getState());

const CounterSubscriber = () => {
  const latest_data = store.getState();
```

```
  console.log(store.getState());

};



store.subscribe(CounterSubscriber);



store.dispatch({ type: "increment" });

store.dispatch({ type: "decrement" });
```

In the above example,createStore is to create store.The state inside store is updated by reducer function.we give this reducer function as argument to createStore.dispatcher() invokes reducer function with a an action.whenever state inside store updates,the subscriber function gets updated state.the store and subscriber are connected through **store.subscriber(Name_of_subscriberFunction)**

->React Redux is the official React binding for Redux. It **allows React components to read data from a Redux Store, and dispatch Actions to the Store to update data**.

->In React usage:

Store>index.js:

```
import { createStore } from "redux";
const counterReducer = (state = { counter: 0 }, action) => {
    if (action.type === "increment") {
        return {
```

```
          counter: state.counter + 1,
            };
 }

  if (action.type === "decrement") {

    return {

      counter: state.counter - 1,

    };

  }

  return state;

};
const store = createStore(counterReducer);
export default store;
```

We can provide this store to required components.we need to import provider component from ‘react-redux’.provider component has store props.so that provider component knows where the store is.the component wrapped in <Provider> component and all child components of that component can use that store.

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

import store from './store/index';
```

```
const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<Provider store={store}><App /></Provider>);
```

->to use data of store in components, we need to import useStore/useSelector.

->useStore provides direct access to data,useSelector is more convenient than useStore because it allows us to automatically select a part from our state managed by store.

->when we use useSelector,redux will automatically set up a subscription.whenever state changes,the component using useSelector,re-executes and always gets latest state.

->`const counter=useSelector(state=>state.counter);`

->function inside useSelector is managed by react-redux,and returns required part.

->Inorder to dispatch actions we need useDispatch() hook.this hook will return a dispatch function,which we call that will dispatch an action against redux-store.

->Class based components:

React-redux also exports connect method.that connects class based components to redux.

->connect is higher order component.connect() returns a function.

->Redux always overrides existing state with latest state.It will not merge latest state with existing state.

->never update/mutate existing state.Instead return new state.
like: state.counter++;
Instead do like: return{ counter:state.counter+1}

->Bigger the project complex to use redux.solution for this is to use redux toolkit.

->Redux toolkit provides easier and convienient way to use redux.Redux toolkit simplifies a couple of aspects of working with redux

->A "slice" is **a collection of Redux reducer logic and actions for a single feature in your app, typically defined together in a single file**. The name comes from splitting up the root Redux state object into multiple "slices" of state.

->A function that accepts an initial state, an object of reducer functions, and a "slice name", and automatically generates action creators and action types that correspond to the reducers and state. This API is the standard approach for writing Redux logic.

```
->createSlice({
    name: "counter",
initialState: InitialState,
reducers: {
```

```
increment(state) {
State.counter++;
},
decrement(state) {
State.counter--;},
increase(state,action) {
State.counter=state.counter+action.amount},
toggle(state) {
state.ShowCounter=!state.ShowCounter},
},
});
```

Reducer functions doesnot need actions,since they are called based on which action is triggered.
We cannot accidentally manipulate existing state when using redux toolkit createSlice.because redux toolkit internally uses another package,called imgur which will detect the code and automatically clone existing state ,create a new state object ,keep all state which are not editing and override the state which we are editing in an immutable way.

->we can pass only single reducer to createStore which creates a store.If we have multiple reducers then createStore doesn't work.for this we need configureStore.configureStore merges multiple reducers into single and creates a store.we pass object as argument to configureStore.'reducer' is expected property in that object

->export const counterActions=counterSlice.actions;//actions is an object with reducer methods as keys.

->createSlice creates a piece of state from global state.

SECTION -20

->When building complex user interfaces,we typically build single page applications(SPAs).

->Only one initial html request and response.
->We add client side react code that basically watches currently active URL and that triggers whenever URL changes,and then leads to different content being displayed on screen when url changes.So instead of loading new html pages from backend,we could add some client side code that simply watches the URL and loads different react component when URL changes.

-> we install package,because this routing functionality,watching url and loading content is not build into react.
->npm install react-router-dom.
->this package belongs to react router tool.

->Adding routing to application is multistep process.
->first step is to define routes we want to support.must define which urls we want to support.
->Second step is to activate our router.Load the route definitions that we defined in first step.
->Third step is to make sure that we all have these components that we want to load

->install: npm install react-router-dom --save

->step1:

```
import {createBrowserRouter} from 'react-router-dom';
```

`createBrowserRouter` is a function provided by 'react-router-dom' package.which allows us to define our routes that we wanma support in application.this function takes array of objects where each object is a route.for this route objects important

properties are path,element(which component should be loaded when that route is active).<RouterProvider> component is imported to tell react to use particular router.It has router prop,the value passed to router prop must be value created using createBrowserRouter.

```
Const router=createBrowserRouter([
{path:'/',element:<home/>}
]);
<RouterProvider router={router}/>
```

-> <Link/> :

It does render an anchor element,but it basically listens for clicks on that element,prevents browser default of sending HTTP request if the link is clicked,and instead simply takes a look at router definitions to update the page accordingly and load the appropriate content.It will also change url but without sending a new HTTP request.this link works only inside <RouterProvider> component.

```
 <Link to='edit'>Edit</Link>
```

->Root.js

```
function Root() {
    return (
          <>
        <MainNavigation />

      <Outlet />
```

```
    </>

  );
```

App.js

```
const router = createBrowserRouter([
{

    path:'/',

    element:<Root/>,

    children:[

      { path: "/", element: <HomePage /> },

      { path: "/products", element: <ProductsPage /> },

    ]




  }



]);
function App() {
return <RouterProvider router={router} />;
}
```

<RouteLayout> is the parent route of remaining routes and it acts as wrapper to these components.
<Outlet> component tells where the child route elements should be rendered.<Root> acts as wrapper to <HomePage> and <Products>.

->when there is an error we can render the component we want.this can be done through adding `errorElement` property to our route definition to define which page should be loaded if an error is created.

-> `<NavLink>` :
To support links that should show us whether they led to the currently active page or not,react-router-dom has an alternative lo Link component.
<NavLink> is used in place of <Link>.<NavLink> has one special behaviour.It has className prop,which is a function that returns css classname that should be added to anchor tag.This function automatically receives an object from which we can destructure isActive property.isActive is provided by 'react-router-dom' as a boolean.if we give '/' to 'to' props in <NavLink> then for all its child paths it treats '/' is in active.To avoid that there is a props called 'end'.

```
<NavLink to='/' className={({isActive})=>
isActive? Classes.active:undefined}
end> Products </NavLink>
```

->`Navigating programatically` :

```
import { useNavigate } from "react-router-dom";
```

```
function HomePage() {

  const navigate=useNavigate();

  const navigateHandler=()=>{

    navigate('/products')

  }

  return (

    <>

      <h1>Welcome To Home Page</h1>

      <button onClick={navigateHandler}>navigate</button>

    </>

  );

}



export default HomePage;
```

useNavigate() hook provides function that can be used for navigating.

->Dyanamic path segments or path parameters:

Suppose there are list of products,if each product needs to render 'productDetails' component with separate data for each component.

{path:'/products/:productsId' , element:<productsDetails/>}

```
<ul>{

  PRODUCTS.map((product)=>(<Link to={`/products/${product.id}`}><li
key={product.id}>{product.title}</li></Link>))

   }

  </ul>
```

':' is important here.productsId is dynamic here,/products/:anyvalue. Anyvalue is used as actual value for this productId placeholder.In place of productId we can use any path it renders <productDetails> component.

->`useParams()` hook return object,this object is nothing but javascript object which contains every dynamic path segment we defined in our route definition as a property

```
const params=useParams();
<p>{params.productsId}</p>
```

Difference between Link and NavLink:

The NavLink is used when you want to highlight a link as active.So, on every routing to a page, the link is highlighted according to the activeClassName.Link is for links that need no highlighting.we can add className to anchor tag with help of <NavLink>.

Absolute and relative paths:

-> '/'

'/products'

These are absolute paths,because they all start with slash.they always seen after domain name

->''

'product'

'products/productDetails'

Because it is not starting with '/'.means the child paths are appended to path of wrapper route.appended at currently active path.

->'..' is relative path which specifies to go one level up.

-> <Link> component has 'relative' prop.It can take two values.

1)path

2)route

By default it takes route.if we absolute path it is appended at end of domain.if we use relative path then it is appended at end of active url,then we use 'relative' prop to control this kind of behaviour.when '..' is used it will remove segments to reach parent route.when we give relative as path in <Link> component it will remove only one segment.

->

```
const router = createBrowserRouter([{
    path: "/",
    element: <Root />,
    errorElement: <Error />,
    children: [
      { path: "/", element: <HomePage /> },
      { path: "/products", element: <ProductsPage /> },
      { path:"/products/:productId",element:<ProductDetails/>}
      ],
```

```
},
]);
```

Here /products ,/products/:productId are direct children to <Root> path./products ,/products/:productId are siblings to each other.so when '..' is used it will remove segments to reach parent path.but if we specify relative prop as ' path' in <Link> component it will remove only one segment.

->special index property:
The route which use this index property is called as index route.It is a default route that should be loaded if the parent's route path is currently active.

->We write fetching data from database inside useEffect() hook.so it will execute after the component is rendered.so component renders before fetching data from database and then data fetching happens.It would be nice if data is fetched first and then component renders with fetched data.React router helps us to do such thing.To route definitions we can additional property called as loader.loader is a property that wants function as value.This function is executed by react whenever we are about to visit that route.So just before route gets rendered,loader function gets triggered and executed by react router.In this function we can load and fetch the data.The value returned by loader function is available to the component that is assigned to a route and all components that are used inside that components.This data is not available to components that are higher level to this component.`useLoaderData()` hook which is executed to get access to closest loader data.fetching data

actually return a promise,but react router checks if a promise is returned and automatically get the resolved data from a promise for you.In loader functions we cannot use hooks,but can use any default browser features(localStorage….. etc)

```
const fetchedEvents = useLoaderData();
return (
     <>
     <EventsList events={fetchedEvents} />
     </>
);
```

->React route provides `useNavigation()` special hook which we can use to check current route transitions state.It return an object.that contains state(Loading,idle,submitting) as property.

```
 const loading = useNavigation();

  const state = loading.state === "loading";

  return (

    <>

      <MainNavigation />

      <div className={classes.text}>

        <Outlet />

        {state && <p>Loading.........</p>}
```

```
      </div>

    </>

  );

}
```

->when there is an error in any route related code including loaders ,react simply renders the closest error element.Error element is rendered not only for 404(not found) error but also to remaining kind of errors. Throw errors using Response();

->`useRouteError()` hook gives error object.

->
```
export async function loader(request,params){
fetch('http://localhost:8080/events/');
}
```

This function accepts route parameters.because react router which calls this function for us actually passes object to this loader function.It has two parameters.'request' and 'param'.we can get route parameter value through params propetry(like get them with useParams)

->`useRouteLoaderData()` hook is similar to useLoaderData() hook ,but it takes 'id' as argument.we can get access to higher level loader from a route that doesn't have loader.we use useRouteLoaderData() instead of useLoaderData().

->

```
  {         path: "events",
```

```
          element: <RootLayout />,
          children: [
            {
              index: true,
              element: <EventsPage />,
              loader: eventsLoader,
            },
            {
              path: ":Id",
              element: <EventDetailPage />,
              loader: eventDetailLoader,
            },
            { path: "new", element: <NewEventPage /> },
            { path: ":Id/edit", element: <EditEventPage /> },
          ],
```

If we use navigate(..) in <EditEventPage/> component,it will go back from 'id/edit' to 'events',but will not go to 'id'.In order to do that it should be parent like

```
path: ":Id",

              loader: eventDetailLoader,

              id:'event-id',

              children: [

                { path: "edit", element: <EditEventPage /> },

                {

                  index: true,

                  element: <EventDetailPage />,

                },

              ],

            },
```

->actions() to send data to backend.Like loader it will also takes function as a argument.use <Form> component provided by 'react-router-dom'.sending request to server is omitted by <Form> tag.But it will take that request that would have been sent and give it to currently active route action.This request contain all data that was submitted as part of the form.we use method property to 'post'.when sending data to api,it is converted to json string(JSON.stringify).

```
export async function action({ request, params }) {
```

```
  const data = await request.formData();

  console.log(data);

  const eventData = {

    title: data.get('title'),

    image: data.get('image'),

    date: data.get('date'),

    description: data.get('description'),

  };

  console.log(eventData);

  const response=await fetch('http://localhost:8080/events',{

    method:'POST',

    body:JSON.stringify(eventData),

    headers:{

      'Content-type':'application/json'

    }

  });

  if(!response.ok)

  {
```

```
    return json({message:'could not save'},{status:500});

  }

  return redirect('/events');

}
```

Request contain `formData()` as method that gets data send by Form component.Data can be accessed using get method.It should return redirect,upon saving it will go to '/events'.It should return something.

->If we want to send data to action on other route definitions,we could point to that action by simply setting action prop to Form component.
action:"/any-other-path"

```
<Form method="POST" action:"any-other-path">
```

->Triggering an action programatically
useSubmit() hook,provides a function.

```
Const submit = useSubmit();
submit({},{method:'delete',action:'/events'});
```

This function takes two arguments.first one is data we want to submit and that data be automatically wrapped in form data object,which we could then extract with this spectial formData() we saw when creating a new event(const data=request.formData();).second argument basically allows us to set same values we could set on a form for example method which we can set to delete.no need to keep action if it is on same route.Here action belongs to '/events' will be triggered

->`useActionData()` to get data returned by action method.

->`useFetcher()` hook.This hook when executed gives you an object.This object contains bunch of useful properties.for example it provides another Form component.it contains methods like
Form,submit,data,formAction,formData,formEnCType,formMethod,load,state.
<fetcher.Form> still trigger an action but it will not initialize route transition.Fetcher basically used whenever we want to trigger an action or loader without actually navigating to the page to which loader or action belongs to.
->useFetcher hook is basically used if you want to interact with some action or loader without transitioning.send request without triggering route changes.

->defer():
We can render a component even though data is not full there yet.defer() is a function,that accepts object as a argument.the object will have key that hold a promise that will eventually resolve.there will be <Await> component which has resolve as a prop that takes promise.between opening and closing tags of<Await> there will be a function that will be executed by react router once data is there,so once promise resolved

-> <suspence> component is used to show a fall back while we are waiting for other data to arrive.it has fallback as a prop.

SECTION-21

-> To access data from backend user must be authenticated.user sends request to backend server,backend server then sends yes/no response.but this response is not safe,we can add that response in other requests.for that there are two solutions

1) server-side sessions :

Store unique identifier on server ,send same identifier to client.

Client sends request along with identifier to protected resource

2) Authentication tokens :

Create permission token on server(not store) ,send token to client.

Client sends token along with request to protected resource

-> The advantage of query parameters would be we can directly link to the page in sign up or login page so that we can directly link user to sign up page even though it is one at the same page just with a different UI being rendered.

Query parameters are a defined set of parameters attached to the end of a url. More simply, they are key=value pairs we can attach to a url, used as one of many ways to pass data to an application.

For example, If the page renders create user and login on same url, we cannot send login data and create user data to backend using same url(api).So, query parameters are added.It differentiate login and create user.

-> useSearchParams() hook is to get access to currently set query parameters(search parameters).This hook actually returns an array.This array has two elements, First element is object that gives us access to currently set query parameters.Second element is a function that updates currently set query parameters.

First element is object,that object can have get, append, delete, entries, forEach, getAll, has, keys,set,sort,toString,values.

-> we can get data submitted by <Form> through request object with this object we can call formData() method to get data object.

-> we cannot use hooks inside actions or loader function.hooks can be used only in components.to get currently set query parameters useSearchParams() hook is used.but in actions method to get currently set query parameters we use URL() constructor.pass ‘request.url’ to URL() constructor.and get searchParams object from that.

-> Suppose path like: ‘/events/new ‘ should be accessible only user is authenticated then we can restrict from goint to that path.
We can add loader(which redirect to ‘/auth’) to that particular route so,loaders execute before component renders,loader changes path to ‘auth’.

```
export function loader(){
    const token=getAuthToken();
    if(!token){
        return redirect('/auth');
    }
    return null;
}
```

```
 {
    path: '/',
    element: <RootLayout />,
    errorElement: <ErrorPage />,
    id:'root-id',
    loader:tokenLoader,
```

```
    children: [
```

->Logout Automatically:

After creation of token,It will be stored in localstorage,and will not be removed from localstorage until logout action is triggered.logout action will be triggered when we click on logout button.So user will be logged in until he clicks on logout button which is not realistic. So, user need to logout automatically after one hour.for that we can use ‘useEffect’.

```
const token = useLoaderData();

const submit = useSubmit();

  useEffect(() => {

    if (!token) {

      return;

    }

    setTimeout(() => {

      submit(null, { action: "/logout", method: "post" });

    }, 1*60*60 * 1000);

  }, [token, submit]);
```

useEffect will be executed after component renders or dependencies changes.here useSubmit() hook is used to programmatically trigger logout action.but this approach is not yet safe,because if user refresh the application then useEffect will execute again and timer will be set again.so user will never get logout.

```
const expiration=new Date();

  expiration.setHours(expiration.getHours()+1)
```

The code you provided creates a new `Date` object called `expiration`, which represents the current date and time.The `setHours()` method is then used to modify the hours component of the `expiration` date. In this case, `expiration.getHours()` retrieves the current hour value from the `expiration` date, and `expiration.getHours() + 1` adds one hour to that value.

> By adding one hour to the current hour value, the `setHours()` method updates the `expiration` date object to reflect the new hour value.
>
> In other words, the code sets the expiration time to be one hour ahead of the current time.

SECTION - 22

->steps to deploy a react application :

1) writing code
2) test the code.It should not have any errors
3) optimise the code(look into concept lazy loading)
4) Build app for production

Shipping less code will load app faster.we are automatically able to generate optimised,minified code with a certain script.once we get optimised package which is ready for deployment,we need to deploy the package.

5) upload production code to server.
6) configure server

-> step 3)

Lazy loading :

Certain pieces of code is loaded only when it is needed to load.

For example import statements need to be resolved before application renders.If the application

is complex then it takes more time to load initial page.To avoid that Lazy loading loads when it is required.

->import() can also be called as a function,which will import something dynamically only when its needed.we need to pass path as an argument.this import() is an asynchronous task so,it returns a promise

```
{ index: true, element: <BlogPage />, loader:
()=>import('./pages/Blog.js').then(module=>module.loader()) },
```

Before going to render <BlogPage> function of loader executes.

```
const blogPage=lazy(()=>import('./pages/Blog'));
```

Here without lazy function import() returns promise.but blogPage need to be component(jsx code).so lazy function is used.

-> <Suspense> component is used by other components to wait for content to be loaded,before actually rendering content.It has fallback prop which shows something on screen until content is loaded.

-> Step 4):
jsx code is not supported in browsers.It must be transformed before uploading into servers,that serve it to end users.During development we preview application in browser which is transformed version of code.the development server which we start with ‘npm start’ transforms the code.
In this step we quit development server and executing different script(build script).
When we run “npm run build” we execute that script ,this will produce the code bundle with highly transformed and optimized code which is ready to be uploaded.after execution,It will create a build folder that should be deployed to a server.in static folder we have optimized js files with dynamically loading junks for lazy loading,but also with main chunk that downloaded initially and it contains all the we wrote and all third party packages we are using including react library.

-> step 5):

If we are building react single page applications(static website) which contains only html,css and js files and does not contain code to be executed on server,instead browser parses code and executed in visitors computer.therefore we only need static site host.we dont need any hosting provider that executes code on server. we don’t execute any server side code.we need static site host.

-> To host your site with Firebase Hosting, you need the Firebase CLI (a command line tool)

Command: “ npm install -g firebase-tools”

-> run “firebase login” inorder to login.

Run “firebase init “ to turn local project to a project connected to firebase.

Questions:

1)? Which Firebase features do you want to set up for this directory? Press Space to select features, then Enter to confirm your choices.

Hosting: Configure files for Firebase Hosting and (optionally) set up GitHub Action deploys

2)? Please select an option:

Use an existing project (project in firebase)

3)? Select a default Firebase project for this directory:

react-deployment-f3256 (React-deployment)

4) ? What do you want to use as your public directory?

Build

5)? Configure as a single-page app (rewrite all urls to /index.html)?

Yes

6) ? Set up automatic builds and deploys with GitHub?

No

7)? File build/index.html already exists. Overwrite?

No

Initialised this project as a firebase project and connected it to the backend.It give us new files which basically establish connection to backend and contain some extra settings for hosting

-> run command: ‘firebase deploy’ to upload files to firebase. It will give you url to visit application.

-> inorder to make website inaccessible
Command: “firebase hosting:disable”

->? Configure as a single-page app (rewrite all urls to /index.html)? => yes

By answering this question with yes, firebase set up some configuration which is respected by firebase server , which basically tells firebase to always return index.html no matter which kind of path we are requesting , firebase always return index.html which then request same javascript files. So this make sure that client side routing is used instead of server side routing

SECTION-25

->context API is good for low frequency updates(like authentication) not for high frequency updates.

->the way context API works is that whenever something changes in context,every component that uses this context is re-build or re-render

-> with just react hooks to create store:

If the components uses custom hook,and if custom hook uses useState then all components which uses custom hook re-renders.

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

```
let globalState={};

let listeners=[];

let actions={};



const useStore=()=>{

    const setState = useState(globalState)[1];

    listeners.push(setState);



}
```

In the above code variables are used outside the custom hook bcz, that should be creacted only once,not for every component that uses it.previously custom hooks are used for logic sharing.now it is using for data sharing.something changes in State then components using this hook re-renders

->

```
const setState = useState(globalState)[1];
```

Only getting function to update state .

-> every component using this custom hook gets their own state and that state is pushed to listeners array.

SECTION-26

->automated testing:

Writing code for testing code.Its not replacement to manual testing.it is addition to manual testing.Manual testing is error prone ,hard to test all possible scenarios and combinations.with automated testing we write extra code that runs and test our code(application code)

-> types of automated testing :

1) unit testing : test the individual building blocks(functions,components)in isolation. Projects typically contain dozens or hundreds of unit tests.

2) Integration testing : test the combination of multiple building blocks .project typically contain

a couple of integration test

3) end–to-end testing : test complete scenarios in your app as the user would experience them.projects typically contain only a few e2e

-> Required tools and setup:

We need a tool for running our tests and asserting the results(jest)+we need a tool for “simulatin”(rendering)our react app /components(React testing library).
These two tools are setup when using create react app.

-> In project:

-> setupTests.js file: does some setup work.we don’t need to anything in this file.
-> App.test.js file: this file contains som testing code

-> App.test.js:

```
import { render, screen } from '@testing-library/react';
```

```
import App from './App';



test('renders learn react link', () => {
  render(<App />);
  const linkElement = screen.getByText(/learn react/i);
  expect(linkElement).toBeInTheDocument();
});
```

Test function(globally available,no need to import) takes two arguments ,first argument iis description of test (it helps you identify this test in testing output,its specially helpful if you have more than one test).Second argument is a function which contains actual testing code.
To run automated test command: *npm test*

After executing “npm test” it does not run the tests right now,instead we need to select ‘a’ to run all test cases.It will automatically look for files that end with dot test dot js and then run all tests that are defined in there with that test function

-> Writing test with three A’s:

1)Arrange : setup the test data, test conditions and test environment.

2)Act : Run logic that should be tested(ex: execute functions)

3) Assert: compare execution results with expected results.(we use screen to get access to virtual dom for elements.expect() which is global function.we pass testing result value that can be anything like string,number,etc.

->Grouping tests together with test suites:

```
 describe("Greetings component", () => {
test("renders Hello world as a test", () => {
    //Arrange
    render(<Greetings />);

    //Act

    //Assert
    const HelloWorldElement = screen.getByText("Hello world", { exact: false });
    expect(HelloWorldElement).toBeInTheDocument();
  });
});
```

Available globally.takes two arguments.first describes name for group of tests.second argument is test.then above is one suite with one test.

->userEvent is an object that helps us trigger userEvents in this virtual screen

`userEvent.click(screen.getByRole('button')`

It needs element on which we simulate a click.

->queryByText () returns null if the element is not found.
->getByRole() fail if we have more than one item with specified role.
->getByAllRole() for more than one item with specified role.(It will instantly look for elements in screen, If we use fetching data then data is not immediately present on screen.) so, use findAllByRole() this will wait for http request succeeded.

-> during testing we don't need to send requests to servers that start changing things there. We have to send to fake server(testing server) or don't send request to server.
->we don't need to test the code that is not written by us.for example we don't test fetch function,since it is not written by us,it is build into browser.instead test the component how it works for different outcomes of sending a request.so replace fetch function with mock function(dummy function) that overrides the build in function.This dummy function what we require but not send real request to server.jest is globally available which runs our test.it has method called FN(it creates a mock function).mockResolvedValueOnce() It allows us to set values this fetch function should resolve to when it's being called

SECTION-27

-> React + typescript:
Adding type safety to react apps.

->typescript is superset to javascript.It is a programming language that extends javascript.It adds more features to js syntax.It is not a library for javascript like react.so,it doesn't use javascript features ,it extends javascript syntax.

->typeScript adds static typing to javascript.javascript is dynamically typed.

->To install Typescript : *npm install typescript*(for specific project)

npm install typescript -g(for globally available)

-> TS doesnot run in browsers.we need a compiler to compile TS to JS.In order to convert TS code to JS code run cmd

npx tsc file-name

-> `//Primitives : number , string , boolean.`

```
//More complex types : arrays , objects.

//Function types , parameters.



//primitives:



let age: number;

age = 21;



let userName: string;

userName = "Max";



let IsInstructor: boolean;

IsInstructor = true;



//more complex types
```

```
let names: string[];

names = ["max", "kelvin"];



let person: {

  name: string;

  age: number;

};

person = {

  name: "max",

  age: 21,

};



let people: {

  name: string;

  age: number;

}[];   //array of objects
```

->type inference:

Once one variable is assigned with string,It cannot be assigned with number.By default it tries to know which type we have defined.

```
let happyname="max";
happyname=12;
```

->Union types:

Union type is atype definition that allows more than one type.

```
let course: string | number = "cse";
course = 12;
```

-> Type aliases(typescript feature and is not considered when code is compiled to javascript):

Type aliases is defined with type keyword.

```
type person1= {
    name: string;
    age: number;
  };


let person1;


let people1:person1[];
```

-> Function and types(we can give types to function parameters and return types)

```
function Add(a:number,b:number):number{
 return (a+b);
}
function PrintOutput(value:number){  //return type is of void
    console.log(value);
}
```

->Generics :

```
function insertAtTheBegining(array:any[],value:any){

    const newArray=[value,...array];

    return newArray;

}



const oldArray=[1,2,3];

const output=insertAtTheBegining(oldArray,-1);

console.log(output);
```

From the above code output is of any type.it is not number type even the input array is of number. So,we can achieve this through generics.

```
function insertAtTheBegining<T>(array:T[],value:T){

    const newArray=[value,...array];

    return newArray;

}



const oldArray=[1,2,3];

const output=insertAtTheBegining(oldArray,-1);

console.log(output);
```

->now output is of number type.If we pass strings then it become string type.

-> To create react app with typescript configuration.

npx create-react-app my-app --template typescript

-> '.tsx' extension is used to allow jsx code.

-> typescript files are automatically compiled to javascript files.it would also happen if we build our code for production using
npm run build

->`const todo:React.FC<{items:string[]}> =(props)=>{`
`Return`
`}`

`From above code FC is functional component.'React.FC' will recomend children props.'React.FC<{item:string[]}>' recommend custom props.reason for using typescript is that it shows errors if we don't use components currectly. Errors can be prevented during development instead of at runtime when we test the app this is benefit of typescript.{items:string[]} is always an object.It is merged with base props.`

->In that case, if the `ToDo` component receives an array of objects as a prop and passes each object as props to `ToDoCom`, then the correct type for the `ToDoCom` component would be `{ items: Item }` rather than `{ items: Item[] }`.

```
interface Item {
  id: number; // Adjust the type of `id` accordingly
  // Add other properties if necessary
}


const ToDoCom: React.FC<{ items: Item }> = (props) => {
  console.log(props.items);
  return <li>{props.items.id}</li>;
```

};

export default ToDoCom;

In this updated code, we define the `Item` interface that represents the structure of each object(not the type.it describes how it should be) in the array. Then, in the `ToDoCom` component definition, we specify that `items` should be of type `Item`, indicating that it receives a single object as a prop. Inside the component, we can access the `id` property directly from `props.items`.

-> React.FormEvenvt

Special type provided by react package, which is event object type which automatically get when listening to form submission.

```
Function sunbmitHAndler(event: React.FormEvent){
.....
}
```

There will be other event types like MouseEvent,....etc.

->useRef()

If we are using useRef for paragraph element then it should be 'HtmlParagraphElement'.If it is input then 'HtmlInputElement'.ref is connected to paragraph element

```
Const refobj=useRef<HtmlParagraphElement>(null);
```

Starting value should be null,otherwise it will be connected to someother element by default.

->If there is a value for sure and we are 100% sure that there will be value instead of null then we can use '!' instead of '?'.

-> bind() is default method in Javascript,which allows us to pre-configure a function for execution.

->**Context API in TypeScript with react**

First of all create a folder called ‘store’.create a file context in that folder.

-> In context.tsx file create a context using “ createContext()” method

```
const ToDoContext=React.createContext({

 items:[],

  AddToDo:()=>{},

  RemoveToDo:()=>{}

});
```

Since it is typescript we need to specify the type of context.

```
const ToDoContext=React.createContext<{

  items:ToDo[],

  AddToDo:()=>void,

  RemoveToDo:(id:string)=>void

}>

({

 items:[],

  AddToDo:()=>{},

  RemoveToDo:()=>{}

});
```

Structure of items,structure of AddToDo function,structure of RemoveToDo is defined as type of context.

->next step is create context provider component which is responsible for managing that context state.

```
const ToDosContextProvider:React.FC=(props)=>{
  return<ToDoContext.Provider>{props.children}</ToDoContext.Provider>
}
```

ToDosContextProvider is used to wrap around components,so that they have access to context store.

```
const ToDosContextProvider: React.FC = (props) => {
  const [Items, SetItems] = useState<ToDo[]>([]);


  function AddToDoItem(item: string) {
    const newToDo = new ToDo(item);
    SetItems((prev) => {
      return prev.concat(newToDo);
    });
  }
  function DeleteItem(Id: string) {
    SetItems((prev) => {
      const Up = prev.filter((obj) => obj.id !== Id);
      return Up;
    });
  }
```

```
  const contextValue = {
    items: Items,
    AddToDo: AddToDoItem,
    RemoveToDo: DeleteItem,
  };

  return (
    <ToDoContext.Provider value={contextValue}>
      {props.children}
    </ToDoContext.Provider>
  );
};
```

-> If we want to manage state then we have to define it in component.funnctions , useState all will be declared here.then create a contextValue and pass it to <ToDoContext.provider> component so,that all components that are warped inside this component can have access to those values.
->export ToDosContextProvider. Export ToDosContext itself to use useContext() hook later.

->In context api,context created using createContenxt is used as useContext in components.ToDosContextProvider is used for wrapping around components.

-> 'tscongfig.json' file configures compilation from ts to js.