npx create-react-app to create a react project.

JSX - JavascriptXML that means html code in javascript.

You can create a component and import it in App.js then you can use it in App.js like <App/>

If you have multiple lines of html code then you should bind it inside the div tag

like

return (

<div>

<h1>Hello</h1>

<button>Click me</button>

</div>

);

like this you should be wrap all the multiple lines html in div tag.

dynamic data binding

Text

Description automatically generated

This is how dynamic data binding can be achieved in react JS.

**PROPS**

Props is nothing but sending data from one component to another component and using it in another component.

Lets see the example below



in this image as we can see we are passing some data from App.js component to ExpenseItem component there we can access the data and use it in out own component like the below image

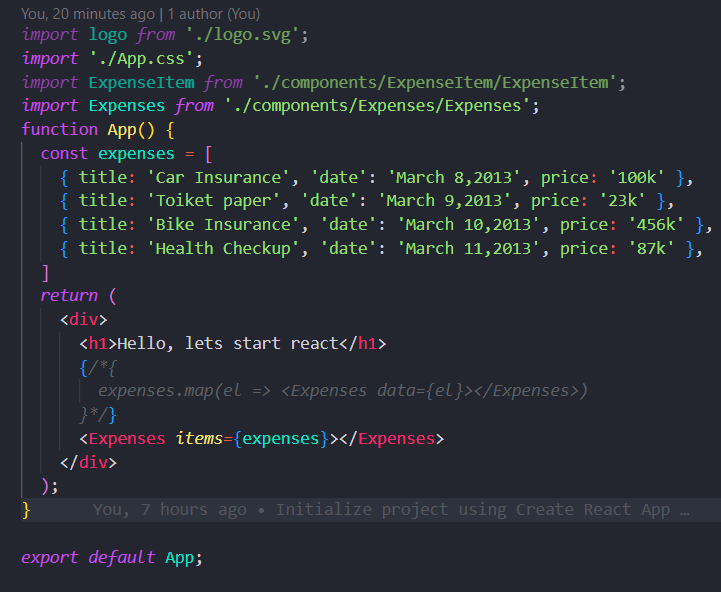


This is what the concept of props is all about



This way also you can follow by iterating through object and send data to component.

Lets see the concept of using nested components



Here in the above picture we can see that we have some data in App.js which we are passing it as props to Expenses component.

Graphical user interface

Description automatically generated

Here in Expenses.js we are receiving data from App.js through props and from here we are sending data to ExpenseItem.js component

Text

Description automatically generated

Here we are using that data from App to Expense to ExpenseItem

We are making the components readable and simpler.

When you are sending data to another component and you’ve some child data in the tag

Lets say

<Expense>

<div>

<h1>Hello</h1>

</div>

</Expense>

Then in the Expense component you can view the inside data using

{props.children}

**STATES**

onClick

to change anything in react we have to use states concept.

useState

a react hook to update the state of an variable.

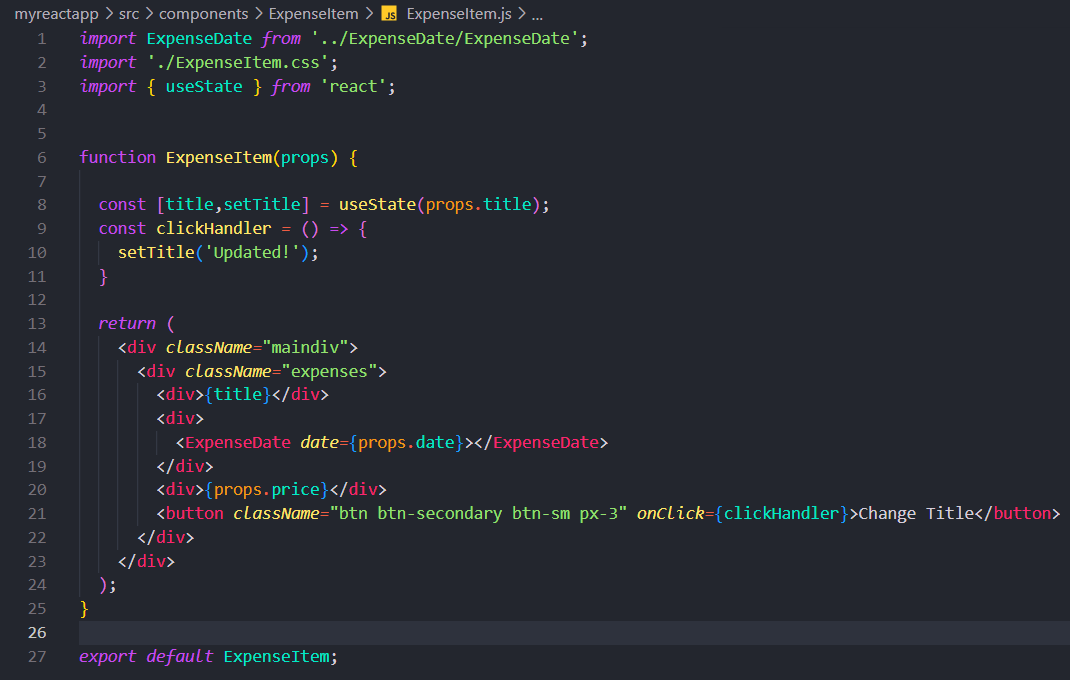
What is the use of useState()

For example we have some title in our UI and we want to change it once a button was

Clicked, once the button got clicked the value should be updated and should be reflected in UI as well, okay how to do that, how to re render a react component in browser, even if you update the value of a variable that will be updated but not reflected because the component is not re rendered again in browser so to make the component re render when any changes happened we can make use of one thing which is react useState hook which helps us to detect changes and re render the component so that the changes will be reflected in browser, lets see how to use useState hook.

The updated value which will be stored in the first argument and the second argument will be the function to be executed to update the value.

<https://www.geeksforgeeks.org/what-is-usestate-in-react/>



Here in the above image we can see that we are rendering a title in browser, which we want to change it when we clicked on a button,

Now we are using useState() hook, it accepts 2 arguments first one is value, and 2nd the one is the function which we can use to update the value or something that we want to make changes to the variables

Now we have to use title,instead of props.title because whatever the changes happened which were being stored in the title variable in line 8.

**onChange event handler**

­­­this event handler is used to grab the data entered by the user from the user form fields.

You can have as many states as you have in your application.

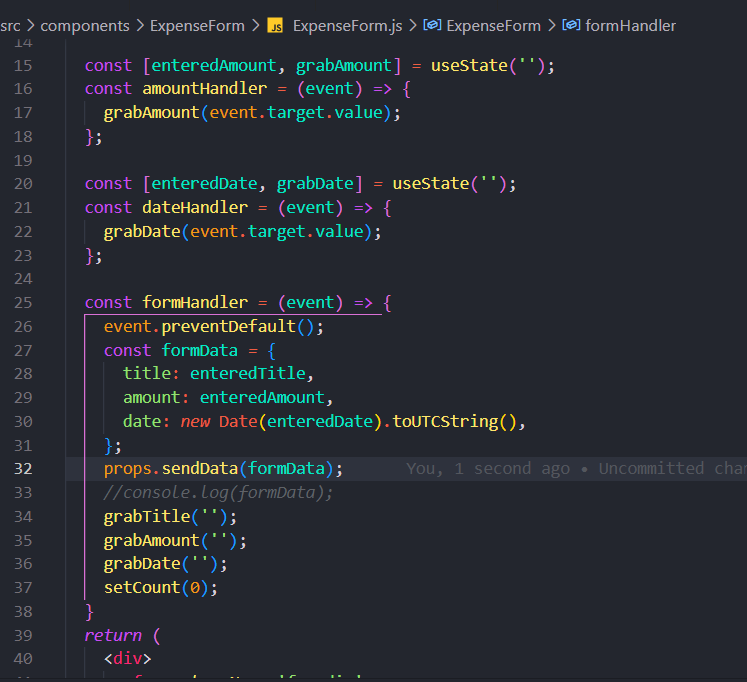
You can have many useState() in your react app.

If the button of type=’submit’ will be clicked inside an form element then It will emit an event of type submit. It will emit the submit event.

**How to send data from child to parent component ?**

Sending data from child to parent

Lets understand with some examples as below



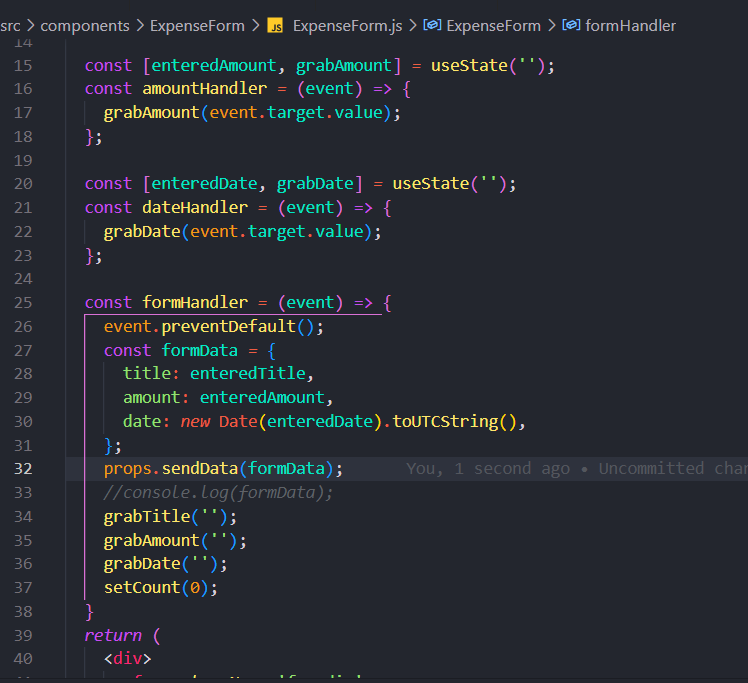
I’m having this collected form data in the component and I want to send it to another component so the steps here are,

1.In the newExpense component iam defining a function(prop) which is having one parameter and then it spreads the data,and adds a new param id and logs the data using console.log

2. Now in the <ExpenseForm> tag add an handler of any name of your choice and pass the defined function to it like we did in the screenshot below.



Now go to the ExpenseForm component and receive the handler through props.



Now as we can see here in line 32 we are passing the formdata to the function defined in NewExpense component through props. As props.sendData(formData) here sendData is the handler attached to the ExpenseForm tag in NewExpense component.

**useEffect**

useEffect will be executed whenever something was rendered on browser.

It will have two parameters first one is a function and second argument is an array, In the array we can specify our functions so that whenever the functions present in the array has any change then the useEffect hook will be executed instead of executing everytime.

Whenever the values specified in the array changes then the useEffect will be executed.

The **useEffect** hook in React will be executed immediately after the component has been rendered on the page (mounting). The exact number of times that it will be executed depends on the inputs you provide as arguments to the hook.

If you pass an empty array (**[]**) as the second argument to **useEffect**, it will only be executed once on initial render, similar to **componentDidMount** in class components.

If you pass an array with values as the second argument, the **useEffect** hook will be executed whenever one of those values changes (it acts as a dependency array). For example, if you pass **[count]** as the second argument and **count** is a state variable, the **useEffect** will be executed whenever **count** changes.

If you don't provide a second argument, the **useEffect** hook will run on every render, including on initial render, which can lead to performance problems and is generally discouraged.

Let’s consider one of the example of useEffect, let’s say we have a login component where correct login details are entered we will be navigated to home page, but here the thing is if reload happens then the app is going to login page again even if we didn’t logged out.

So we want to store the login information in local storage so that we will not be automatically logged out on page reload.

But if we check

*if*(localStorage*.*getItem('isLogin') *==* 'true'){

      setIsLoggedIn(true);

}

If we do this then we fell into a infinite loop because the component is initialized and we’re checking the login present or not if present we are changing the state, so state changed so component will be re-rendered like this we fell into a infinite loop,

So to avoid this we can put the same code in useEffect and the code in useEffect will be initially executed and after that the component will be re-rendered only if any changes happened to the dependencies provided as array to useEffect hook, and if not useEffect will not be triggered unnecessarily.

So with useEffect we are avoiding unnecessary code execution means we are avoiding the side effects.

**What to add & Not to add as Dependencies**

In the previous lecture, we explored useEffect() dependencies.

You learned, that you should add "everything" you use in the effect function as a dependency - i.e. all state variables and functions you use in there.

That is correct, but there are a **few exceptions** you should be aware of:

* You **DON'T need to add state updating functions** (as we did in the last lecture with setFormIsValid): React guarantees that those functions never change, hence you don't need to add them as dependencies (you could though)
* You also **DON'T need to add "built-in" APIs or functions** like fetch(), localStorage etc (functions and features built-into the browser and hence available globally): These browser APIs / global functions are not related to the React component render cycle and they also never change
* You also **DON'T need to add variables or functions** you might've **defined OUTSIDE of your components** (e.g. if you create a new helper function in a separate file): Such functions or variables also are not created inside of a component function and hence changing them won't affect your components (components won't be re-evaluated if such variables or functions change and vice-versa)

So long story short: You must add all "things" you use in your effect function **if those "things" could change because your component (or some parent component) re-rendered.** That's why variables or state defined in component functions, props or functions defined in component functions have to be added as dependencies!

Here's a made-up dummy example to further clarify the above-mentioned scenarios:

1. import { useEffect, useState } from 'react';
3. let myTimer;
5. const MyComponent = (props) => {
6. const [timerIsActive, setTimerIsActive] = useState(false);
8. const { timerDuration } = props; // using destructuring to pull out specific props values
10. useEffect(() => {
11. if (!timerIsActive) {
12. setTimerIsActive(true);
13. myTimer = setTimeout(() => {
14. setTimerIsActive(false);
15. }, timerDuration);
16. }
17. }, [timerIsActive, timerDuration]);
18. };

In this example:

* timerIsActive is **added as a dependency** because it's component state that may change when the component changes (e.g. because the state was updated)
* timerDuration is **added as a dependency** because it's a prop value of that component - so it may change if a parent component changes that value (causing this MyComponent component to re-render as well)
* setTimerIsActive is **NOT added as a dependency** because it's that **exception**: State updating functions could be added but don't have to be added since React guarantees that the functions themselves never change
* myTimer is **NOT added as a dependency** because it's **not a component-internal variable** (i.e. not some state or a prop value) - it's defined outside of the component and changing it (no matter where) **wouldn't cause the component to be re-evaluated**
* setTimeout is **NOT added as a dependency** because it's **a built-in API** (built-into the browser) - it's independent from React and your components, it doesn't change

Normally we are updating the formstate whenever the email is changes or password is changed

So we can do that in useEffect as follows

useEffect(() => {

    setFormIsValid(

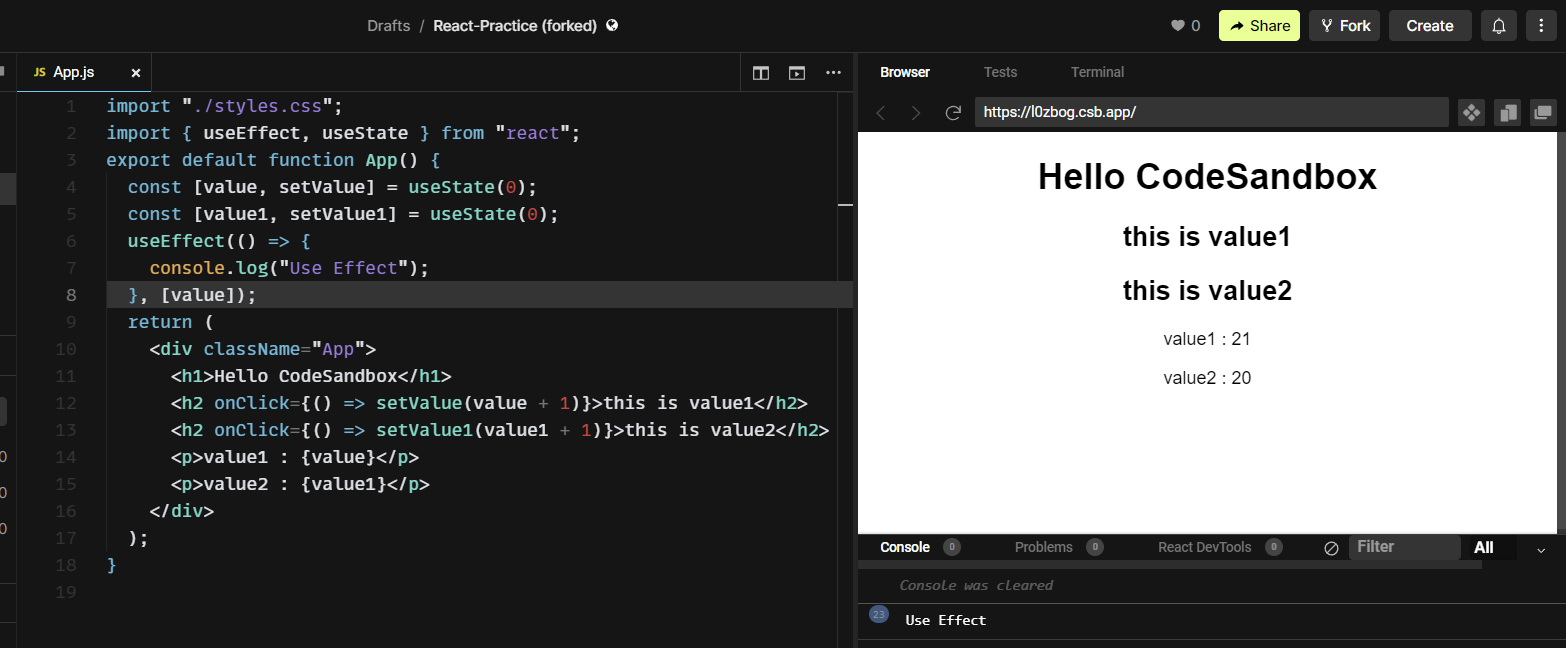
      enteredEmail*.*includes('@') *&&* enteredPassword*.*trim()*.*length *>* 6

    );

  },[enteredEmail,enteredPassword]);

Here we can see that the component is re-rendered whenever the email or password is changed since we have added the enteredEmail,enteredPassword to the dependent array so whenever these values changes then the useEffect will be executed so the component will be re-rendered.

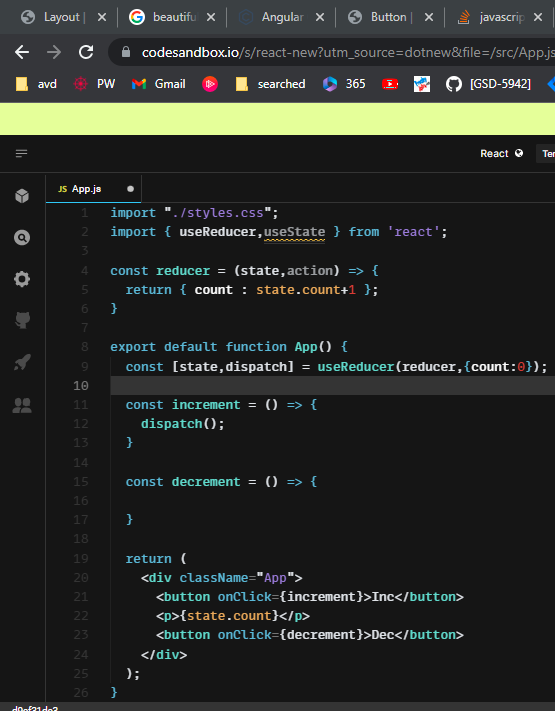
The cleanup function executes before the actual function executed in useEFfect.



We can understand useEffect by the above example, useEffect is being rendered only when this is value1 is clicked because we specified value1 to the dependency array in useEffect so that whenever the value1 state changed then the useEffect will be executed.

**useReducer**

it can be used for advanced state management in react.



If we see the example we can find that we have defined a reducer

Const [state,dispatch] = useReducer(reducer, { count : 0 });

* + reducer is a function which is defined externally,as we can see above.
  + Whenever we call the dispatch function it will internally calls the reducer function and then it updates the values accordingly.
  + The dispatch function pass the { count : 0 } object to the state variable in reducer function with that it can access the values and make changes to it.
  + And whatever you pass in the dispatch function as an argument while calling it will be stored in action variable in reducer function so with that we can necessarily make modifications as required.

**import "./styles.css";**

**import { useReducer,useState } from 'react';**

**const reducer = (state,action) => {**

**if(action.type == ACTIONS.INCREMENT){**

**return { count : state.count + 1 };**

**}**

**return { count : state.count - 1 };**

**}**

**const ACTIONS = {**

**INCREMENT: 'INCREMENT',**

**DECREMENT: 'DECREMENT'**

**}**

**export default function App() {**

**const [state,dispatch] = useReducer(reducer,{count:0});**

**const increment = () => {**

**dispatch({ type : ACTIONS.INCREMENT });**

**}**

**const decrement = () => {**

**dispatch({ type : ACTIONS.DECREMENT });**

**}**

**return (**

**<div className="App">**

**<button onClick={increment}>Inc</button>**

**<p>{state.count}</p>**

**<button onClick={decrement}>Dec</button>**

**</div>**

**);**

**}**

Here we can see that first we have defined a useReducer like

Const [state,dispatch] = useReducer(reducer, { count : 0 });

Here { count : 0 } is the initial value that we are passing.

And **dispatch** is the function that we will call and it will call another function internally.

So if we do like dispatch( { type : ‘INCREMENT’ } ) then the dispatch function of useReducer will be called and it will call the reducer() function and in that reducer function the state param stores the { count : 0 } and the action param stores the { type : ‘INCREMENT’ } so we can use that action param to decide which has to be returned and which operation has to be performed.

Well this is an example todo app

import "./styles.css";

import { useReducer,useState } from 'react';

const TODO\_STATE = {

ADD\_TODO:'add'

}

const reducer = (todos,action) => {

switch(action.type){

case TODO\_STATE.ADD\_TODO:

return [...todos,addTodo(action.payload.name)]

}

}

const addTodo = (name) => {

return { name : name, id: Math.random()\*10 };

}

export default function App() {

const [todos,dispatch] = useReducer(reducer,[]);

const [name,setName] = useState('');

const submitHandler = (e) => {

e.preventDefault();

dispatch({ type : TODO\_STATE.ADD\_TODO ,payload : { name : name }});

setName('');

}

console.log(todos);

return (

<div className="App">

<form>

<input type='text' value={name} onChange={e => setName(e.target.value)}/>

<button type='submit' onClick={submitHandler}>Submit</button>

</form>

{

todos.map(item => {

return <li>{item.name}</li>

})

}

</div>

);

}

**So this is a TODO app using useReducer.**

If you don’t provide any dependencies array in useEffect and not even the empty array, then the useEffect will be executed for every re-render every component updates that will make unnecessary network calls so if you don’t have any dependencies then atleast provide the empty array.

If you didn’t provide any dependencies atleast the empty dependency array then the useEffect will be executed for every other state changes, if you provide the empty dependency array then the useEffect will be executed for only when the component runs.

import "./styles.css";

import { useEffect, useState } from "react";

export default function App() {

useEffect(() => {

console.log("useEffect");

}, []);

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

return (

<div className="App">

<p>use effect</p>

<p onClick={() => setCount(count + 1)}>use state</p>

<p>{count}</p>

</div>

);

}

Here in useEffect we have provided an empty array as dependency, so other state changes like when count has changes it does not executes the useEffect where as if you don’t provide the empty array then for every other state updates useEffect will be re-rendered. Could be a bad practice.