Create the reat app with the two ways:

>> npx create-react-app react\_app\_name

– Using the Package Manager like Vite or Parcel

Vite is the preferable way to create the react app it is faster process of creating the reac app.

>> npm create vite@latest

>> npm install

**Create the Custome Element using the Java script**

const pelement = React.createElement(

'p',

{name: "para", id: "para"},

"Click to open the para",

)

const element = React.createElement(

'a',

{href: "https://www.google.com", alt: "hello"},

"Click to open the Google",

pelement

)

ReactDOM.createRoot(document.getElementById('root')).render(

// <React.StrictMode>

element

// </React.StrictMode>,

)

Download Images: <https://www.pexels.com/>

For the components: <https://tailwindcss.com/>

<https://www.devui.io/>

**Hooks:**

**use():**

use() hook is same like the useContext() hook.

use is a React Hook that lets you read the value of a resource like a Promise or context.

import { createContext, use } from 'react';

const ThemeContext = createContext(null);

<ThemeContext.Provider value="dark">

<Form />

</ThemeContext.Provider>

const theme = use(ThemeContext);

const className = 'button-' + theme;

The use Hook returns the value that was read from the resource like the resolved value of a Promise or context.

Reference

use(resource)

Usage

Reading context with use

Streaming data from the server to the client

Dealing with rejected Promises

Troubleshooting

“Suspense Exception: This is not a real error!”

**useId():**

useId is a React Hook for generating unique IDs that can be passed to accessibility attributes.

Reference

useId()

Usage

Generating unique IDs for accessibility attributes

Generating IDs for several related elements

Specifying a shared prefix for all generated IDs

Using the same ID prefix on the client and the server

const id = useId()

const passwordHintId = useId();

**useState():**

use for change the state.

Reference

useState(initialState)

set functions, like setSomething(nextState)

Usage

Adding state to a component

Updating state based on the previous state

Updating objects and arrays in state

Avoiding recreating the initial state

Resetting state with a key

Storing information from previous renders

Troubleshooting

I’ve updated the state, but logging gives me the old value

I’ve updated the state, but the screen doesn’t update

I’m getting an error: “Too many re-renders”

My initializer or updater function runs twice

I’m trying to set state to a function, but it gets called instead

import { useState } from "react";

**const [color, setColor] = useState("red");**

<button

type="button"

**onClick={() => setColor("blue")}**

>Blue</button>

**useEffect():** use for run the function when some change occurred in the given dependencies list and synchronize a component with an external system.

Reference

useEffect(setup, dependencies?)

Usage

Connecting to an external system

Wrapping Effects in custom Hooks

Controlling a non-React widget

Fetching data with Effects

Specifying reactive dependencies

Updating state based on previous state from an Effect

Removing unnecessary object dependencies

Removing unnecessary function dependencies

Reading the latest props and state from an Effect

Displaying different content on the server and the client

Troubleshooting

My Effect runs twice when the component mounts

My Effect runs after every re-render

My Effect keeps re-running in an infinite cycle

My cleanup logic runs even though my component didn’t unmount

My Effect does something visual, and I see a flicker before it runs

useEffect(() => {

setColor(color);

}, [color]);

**useMemo():**

lets you cache the result of a calculation between re-renders. user to store the result in the cache. Same as useCallback() hook.

const cachedValue = useMemo(calculateValue, dependencies)

Reference

useMemo(calculateValue, dependencies)

Usage

Skipping expensive recalculations

Skipping re-rendering of components

Memoizing a dependency of another Hook

Memoizing a function

Troubleshooting

My calculation runs twice on every re-render

My useMemo call is supposed to return an object, but returns undefined

Every time my component renders, the calculation in useMemo re-runs

I need to call useMemo for each list item in a loop, but it’s not allowed

import { useState, useMemo } from "react";

const calculation = **useMemo**(() => expensiveCalculation(count), [count]);

**useCallback():** use for store the result in the cache and optimize it.

Skipping re-rendering of components.

const addTodo = **useCallback**(() => {

**setTodo**((prev) => [...prev, "New Todo"]);

}, [todos]);

You should only rely on useCallback as a performance optimization.

Difference between useMemo and useCallback is useMemo() memorize the value where useCallback() memorized the function.

useMemo is used to memoize values or computations, helping to optimize expensive calculations or data transformations. useCallback is used to memoize callback functions, reducing unnecessary re-creation of functions and optimizing component re-renders.

Reference

useCallback(fn, dependencies)

Usage

Skipping re-rendering of components

Updating state from a memoized callback

Preventing an Effect from firing too often

Optimizing a custom Hook

Troubleshooting

Every time my component renders, useCallback returns a different function

I need to call useCallback for each list item in a loop, but it’s not allowed

**useRef():**

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

Changing a ref using the ref.current.[...] does not trigger a re-render.

Reference

useRef(initialValue)

Usage

Referencing a value with a ref

Manipulating the DOM with a ref

Avoiding recreating the ref contents

Troubleshooting

I can’t get a ref to a custom component

import { useRef } from "react";

const inputElement = useRef();

const focusInput = () => {

inputElement.current.focus();

};

<button onClick={focusInput}>Focus Input</button>

**useContext():** user to manage state globally to handle the prop drilling.

Ex:

import { useState, createContext } from "react";

**const UserContext = createContext()**

function Component1() {

const [user, setUser] = useState("Jesse Hall");

return (

**<UserContext.Provider value={user}>**

<h1>{`Hello ${user}!`}</h1>

<Component2 user={user} />

**</UserContext.Provider>**

);

}

import { useState, createContext, useContext } from "react";

function Component5() {

**const user = useContext(UserContext);**

return (

<>

<h1>Component 5</h1>

<h2>{`Hello ${user} again!`}</h2>

</>

);

}

Reference

useContext(SomeContext)

Usage

Passing data deeply into the tree

Updating data passed via context

Specifying a fallback default value

Overriding context for a part of the tree

Optimizing re-renders when passing objects and functions

Troubleshooting

My component doesn’t see the value from my provider

I am always getting undefined from my context although the default value is different

**useDebugValue():** use to check the debug value in the hook section inside the component in devtool.

let date = new Date();

console.log(date);

useDebugValue(date, date => date.toDateString());

**useDeferredValue():** lets you defer updating a part of the UI.

Reference

useDeferredValue(value)

Usage

Showing stale content while fresh content is loading

Indicating that the content is stale

Deferring re-rendering for a part of the UI

Ex: search with the word “a” then “ab” till not load the data of the “ab” it shows the “a” data.

const [query, setQuery] = useState('');

const **deferredQuery** = **useDeferredValue**(**query**);

const isStale = query !== deferredQuery;

<Suspense fallback={<h2>Loading...</h2>}>

<div style={{

opacity: isStale ? 0.5 : 1,

transition: isStale ? 'opacity 0.2s 0.2s linear' : 'opacity 0s 0s linear'

}}>

<SearchResults query={**deferredQuery**} />

</div>

</Suspense>

**useImperativeHandle:**

useImperativeHandle is a React Hook that lets you customize the handle exposed as a ref.

useImperativeHandle(ref, createHandle, dependencies?)

Reference

useImperativeHandle(ref, createHandle, dependencies?)

Usage

Exposing a custom ref handle to the parent component

Exposing your own imperative methods

**useInsertionEffect:**

useInsertionEffect allows inserting elements into the DOM before any layout effects fire.

useInsertionEffect(setup, dependencies?)

Reference

useInsertionEffect(setup, dependencies?)

Usage

Injecting dynamic styles from CSS-in-JS libraries

function useCSS(rule) {

**useInsertionEffect**(() => {

// As explained earlier, we don't recommend runtime injection of <style> tags.

// But if you have to do it, then it's important to do in useInsertionEffect.

if (!isInserted.has(rule)) {

isInserted.add(rule);

document.head.appendChild(getStyleForRule(rule));

}

});

return rule;

}

function Button() {

const className = useCSS('...');

return <div className={className} />;

}

**useLayoutEffect:**

useLayoutEffect is a version of useEffect that fires before the browser repaints the screen.

useLayoutEffect can hurt performance. Prefer useEffect when possible.

Reference

useLayoutEffect(setup, dependencies?)

Usage

Measuring layout before the browser repaints the screen

Troubleshooting

I’m getting an error: “useLayoutEffect does nothing on the server”

import { useState, useRef, useLayoutEffect } from 'react';

function Tooltip() {

const ref = useRef(null);

const [tooltipHeight, setTooltipHeight] = useState(0);

**useLayoutEffect**(() => {

const { height } = ref.current.getBoundingClientRect();

setTooltipHeight(height);

}, []);

**useOptimistic():**

lets you optimistically update the UI. This state is called the “optimistic” state because it is usually used to immediately present the user with the result of performing an action, even though the action actually takes time to complete.

Reference

useOptimistic(state, updateFn)

Usage

Optimistically updating forms

import { useOptimistic, useState, useRef } from "react";

import { deliverMessage } from "./actions.js";

function Thread({ messages, sendMessage }) {

const formRef = useRef();

async function formAction(formData) {

addOptimisticMessage(formData.get("message"));

formRef.current.reset();

await sendMessage(formData);

}

const [optimisticMessages, addOptimisticMessage] = **useOptimistic**(

messages,

(state, newMessage) => [

...state,

{

text: newMessage,

sending: true

}

]

);

return (

<>

{optimisticMessages.map((message, index) => (

<div key={index}>

{message.text}

{!!message.sending && <small> (Sending...)</small>}

</div>

))}

<form action={formAction} ref={formRef}>

<input type="text" name="message" placeholder="Hello!" />

<button type="submit">Send</button>

</form>

</>

);

}

export default function App() {

const [messages, setMessages] = useState([

{ text: "Hello there!", sending: false, key: 1 }

]);

async function sendMessage(formData) {

const sentMessage = await deliverMessage(formData.get("message"));

setMessages((messages) => [...messages, { text: sentMessage }]);

}

return <Thread messages={messages} sendMessage={sendMessage} />;

}

**useSyncExternalStore():**

that lets you subscribe to an external store. mostly useful if you need to integrate with existing non-React code.

Reference

useSyncExternalStore(subscribe, getSnapshot, getServerSnapshot?)

Usage

Subscribing to an external store

Subscribing to a browser API

Extracting the logic to a custom Hook

Adding support for server rendering

Troubleshooting

I’m getting an error: “The result of getSnapshot should be cached”

My subscribe function gets called after every re-render

useSyncExternalStore(subscribe, getSnapshot, getServerSnapshot?)

Call useSyncExternalStore at the top level of your component to read a value from an external data store.

import { useSyncExternalStore } from 'react';

import { todosStore } from './todoStore.js';

function TodosApp() {

const todos = useSyncExternalStore(todosStore.subscribe, todosStore.getSnapshot);

// ...

}

Example: check whether the user is online or offline

import { useSyncExternalStore } from 'react';

export default function ChatIndicator() {

const isOnline = **useSyncExternalStore**(**subscribe**, **getSnapshot**);

return <h1>{isOnline ? '✅ Online' : '❌ Disconnected'}</h1>;

}

function **getSnapshot**() {

return navigator.onLine;

}

function **subscribe**(callback) {

window.addEventListener('online', callback);

window.addEventListener('offline', callback);

return () => {

window.removeEventListener('online', callback);

window.removeEventListener('offline', callback);

};

}

**useTransition():**

lets you update the state without blocking the UI.

const [isPending, startTransition] = useTransition()

Reference

useTransition()

startTransition function

Usage

Marking a state update as a non-blocking transition

Updating the parent component in a transition

Displaying a pending visual state during the transition

Preventing unwanted loading indicators

Building a Suspense-enabled router

Displaying an error to users with a error boundary

Troubleshooting

Updating an input in a transition doesn’t work

React doesn’t treat my state update as a transition

I want to call useTransition from outside a component

The function I pass to startTransition executes immediately

const [isPending, startTransition] = useTransition();

const [tab, setTab] = useState('about');

function selectTab(nextTab) {

startTransition(() => {

setTab(nextTab);

});

}

**useReducer():**

lets you add a reducer to your component.

const [state, dispatch] = useReducer(reducer, initialArg, init?)

Reference

useReducer(reducer, initialArg, init?)

dispatch function

Usage

Adding a reducer to a component

Writing the reducer function

Avoiding recreating the initial state

Troubleshooting

I’ve dispatched an action, but logging gives me the old state value

I’ve dispatched an action, but the screen doesn’t update

A part of my reducer state becomes undefined after dispatching

My entire reducer state becomes undefined after dispatching

I’m getting an error: “Too many re-renders”

My reducer or initializer function runs twice

import { useReducer } from 'react';

function reducer(state, action) {

// ...

}

function MyComponent() {

const [state, dispatch] = **useReducer**(reducer, { age: 42 });

Example:

import React, { useReducer } from "react";

function **reducer**(state, action) {

if (action.**type** === "**increment**") {

return {

age: state.age + 1,

};

}

}

function UseReducerHook() {

const [**state, dispatch**] = **useReducer**(reducer, { age: 50 });

return (

<div>

<input

type="button"

name="button"

id="button"

value={"ADD"}

onClick={() => {

**dispatch**({ **type**: "**increment**" });

}}

/>

<label>{state.age}</label>

</div>

);

}

export default UseReducerHook;