**React Hooks**

**useToggle**

import { useCallback, useState } from 'react';

// Usage

function App() {

// Call the hook which returns, current value and the toggler function

const [isTextChanged, setIsTextChanged] = useToggle();

return (

<button onClick={setIsTextChanged}>{isTextChanged ? 'Toggled' : 'Click to Toggle'}</button>

);

}

// Hook

// Parameter is the boolean, with default "false" value

const useToggle = (initialState = false) => {

// Initialize the state

const [state, setState] = useState(initialState);

// Define and memorize toggler function in case we pass down the component,

// This function change the boolean value to it's opposite value

const toggle = useCallback(() => setState(state => !state), []);

return [state, toggle]

}

**useFireStoreQuery**

This hook makes it super easy to subscribe to data in your Firestore database without having to worry about state management. Instead of calling Firestore's query.onSnapshot() method you simply pass a query to useFirestoreQuery() and you get back everything you need, including status, data, and error. Your component will re-render when data changes and your subscription will be automatically removed when the component unmounts. Our example even supports dependent queries where you can wait on needed data by passing a falsy value to the hook. Read through the recipe and comments below to see how it works.

function ProfilePage({ uid }) {

// Subscribe to Firestore document

const { data, status, error } = useFirestoreQuery(

firestore.collection("profiles").doc(uid)

);

if (status === "loading") {

return "Loading...";

}

if (status === "error") {

return `Error: ${error.message}`;

}

return (

<div>

<ProfileHeader avatar={data.avatar} name={data.name} />

<Posts posts={data.posts} />

</div>

);

}

// Reducer for hook state and actions

const reducer = (state, action) => {

switch (action.type) {

case "idle":

return { status: "idle", data: undefined, error: undefined };

case "loading":

return { status: "loading", data: undefined, error: undefined };

case "success":

return { status: "success", data: action.payload, error: undefined };

case "error":

return { status: "error", data: undefined, error: action.payload };

default:

throw new Error("invalid action");

}

};

// Hook

function useFirestoreQuery(query) {

// Our initial state

// Start with an "idle" status if query is falsy, as that means hook consumer is

// waiting on required data before creating the query object.

// Example: useFirestoreQuery(uid && firestore.collection("profiles").doc(uid))

const initialState = {

status: query ? "loading" : "idle",

data: undefined,

error: undefined,

};

// Setup our state and actions

const [state, dispatch] = useReducer(reducer, initialState);

// Get cached Firestore query object with useMemoCompare (https://usehooks.com/useMemoCompare)

// Needed because firestore.collection("profiles").doc(uid) will always being a new object reference

// causing effect to run -> state change -> rerender -> effect runs -> etc ...

// This is nicer than requiring hook consumer to always memoize query with useMemo.

const queryCached = useMemoCompare(query, (prevQuery) => {

// Use built-in Firestore isEqual method to determine if "equal"

return prevQuery && query && query.isEqual(prevQuery);

});

useEffect(() => {

// Return early if query is falsy and reset to "idle" status in case

// we're coming from "success" or "error" status due to query change.

if (!queryCached) {

dispatch({ type: "idle" });

return;

}

dispatch({ type: "loading" });

// Subscribe to query with onSnapshot

// Will unsubscribe on cleanup since this returns an unsubscribe function

return queryCached.onSnapshot(

(response) => {

// Get data for collection or doc

const data = response.docs

? getCollectionData(response)

: getDocData(response);

dispatch({ type: "success", payload: data });

},

(error) => {

dispatch({ type: "error", payload: error });

}

);

}, [queryCached]); // Only run effect if queryCached changes

return state;

}

// Get doc data and merge doc.id

function getDocData(doc) {

return doc.exists === true ? { id: doc.id, ...doc.data() } : null;

}

// Get array of doc data from collection

function getCollectionData(collection) {

return collection.docs.map(getDocData);

}

**useMemoCompare**

This hook is similar to useMemo, but instead of passing an array of dependencies we pass a custom compare function that receives the previous and new value. The compare function can then compare nested properties, call object methods, or anything else to determine equality. If the compare function returns true then the hook returns the old object reference.

It's worth noting that, unlike useMemo, this hook isn't meant to avoid expensive calculations. It needs to be passed a computed value so that it can compare it to the old value. Where this comes in handy is if you want to offer a library to other developers and it would be annoying to force them to memoize an object before passing it to your library. If that object is created in the component body (often the case if it's based on props) then it's going to be a new object on every render. If that object is a useEffect dependency then it's going to cause the effect to fire on every render, which can lead to problems or even an infinite loop. This hook allows you to avoid that scenario by using the old object reference instead of the new one if your custom comparison function deems them equal.

import React, { useState, useEffect, useRef } from "react";

// Usage

function MyComponent({ obj }) {

const [state, setState] = useState();

// Use the previous obj value if the "id" property hasn't changed

const objFinal = useMemoCompare(obj, (prev, next) => {

return prev && prev.id === next.id;

});

// Here we want to fire off an effect if objFinal changes.

// If we had used obj directly without the above hook and obj was technically a

// new object on every render then the effect would fire on every render.

// Worse yet, if our effect triggered a state change it could cause an endless loop

// where effect runs -> state change causes rerender -> effect runs -> etc ...

useEffect(() => {

// Call a method on the object and set results to state

return objFinal.someMethod().then((value) => setState(value));

}, [objFinal]);

// So why not pass [obj.id] as the dependency array instead?

useEffect(() => {

// Then eslint-plugin-hooks would rightfully complain that obj is not in the

// dependency array and we'd have to use eslint-disable-next-line to work around that.

// It's much cleaner to just get the old object reference with our custom hook.

return obj.someMethod().then((value) => setState(value));

}, [obj.id]);

return <div> ... </div>;

}

// Hook

function useMemoCompare(next, compare) {

// Ref for storing previous value

const previousRef = useRef();

const previous = previousRef.current;

// Pass previous and next value to compare function

// to determine whether to consider them equal.

const isEqual = compare(previous, next);

// If not equal update previousRef to next value.

// We only update if not equal so that this hook continues to return

// the same old value if compare keeps returning true.

useEffect(() => {

if (!isEqual) {

previousRef.current = next;

}

});

// Finally, if equal then return the previous value

return isEqual ? previous : next;

}

**useAsync**

It's generally a good practice to indicate to users the status of any async request. An example would be fetching data from an API and displaying a loading indicator before rendering the results. Another example would be a form where you want to disable the submit button when the submission is pending and then display either a success or error message when it completes.

Rather than litter your components with a bunch of useState calls to keep track of the state of an async function, you can use our custom hook which takes an async function as an input and returns the value, error, and status values we need to properly update our UI. Possible values for status prop are: "idle", "pending", "success", "error". As you'll see in the code below, our hook allows both immediate execution and delayed execution using the returned execute function.

import React, { useState, useEffect, useCallback } from "react";

// Usage

function App() {

const { execute, status, value, error } = useAsync(myFunction, false);

return (

<div>

{status === "idle" && <div>Start your journey by clicking a button</div>}

{status === "success" && <div>{value}</div>}

{status === "error" && <div>{error}</div>}

<button onClick={execute} disabled={status === "pending"}>

{status !== "pending" ? "Click me" : "Loading..."}

</button>

</div>

);

}

// An async function for testing our hook.

// Will be successful 50% of the time.

const myFunction = () => {

return new Promise((resolve, reject) => {

setTimeout(() => {

const rnd = Math.random() \* 10;

rnd <= 5

? resolve("Submitted successfully 🙌")

: reject("Oh no there was an error 😞");

}, 2000);

});

};

// Hook

const useAsync = (asyncFunction, immediate = true) => {

const [status, setStatus] = useState("idle");

const [value, setValue] = useState(null);

const [error, setError] = useState(null);

// The execute function wraps asyncFunction and

// handles setting state for pending, value, and error.

// useCallback ensures the below useEffect is not called

// on every render, but only if asyncFunction changes.

const execute = useCallback(() => {

setStatus("pending");

setValue(null);

setError(null);

return asyncFunction()

.then((response) => {

setValue(response);

setStatus("success");

})

.catch((error) => {

setError(error);

setStatus("error");

});

}, [asyncFunction]);

// Call execute if we want to fire it right away.

// Otherwise execute can be called later, such as

// in an onClick handler.

useEffect(() => {

if (immediate) {

execute();

}

}, [execute, immediate]);

return { execute, status, value, error };

};

**useAuth**

A very common scenario is you have a bunch of components that need to render different depending on whether the current user is logged in and sometimes call authentication methods like signin, signout, sendPasswordResetEmail, etc.

This is a perfect use-case for a useAuth hook that enables any component to get the current auth state and re-render if it changes. Rather than have each instance of the useAuth hook fetch the current user, the hook simply calls useContext to get the data from farther up in the component tree. The real magic happens in our <ProvideAuth> component and our useProvideAuth hook which wraps all our authentication methods (in this case we're using Firebase) and then uses React Context to make the current auth object available to all child components that call useAuth.

import React from "react";

import { ProvideAuth } from "./use-auth.js";

function App(props) {

return (

<ProvideAuth>

{/\*

Route components here, depending on how your app is structured.

If using Next.js this would be /pages/\_app.js

\*/}

</ProvideAuth>

);

}

// Any component that wants auth state

import React from "react";

import { useAuth } from "./use-auth.js";

function Navbar(props) {

// Get auth state and re-render anytime it changes

const auth = useAuth();

return (

<NavbarContainer>

<Logo />

<Menu>

<Link to="/about">About</Link>

<Link to="/contact">Contact</Link>

{auth.user ? (

<Fragment>

<Link to="/account">Account ({auth.user.email})</Link>

<Button onClick={() => auth.signout()}>Signout</Button>

</Fragment>

) : (

<Link to="/signin">Signin</Link>

)}

</Menu>

</NavbarContainer>

);

}

// Hook (use-auth.js)

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

import \* as firebase from "firebase/app";

import "firebase/auth";

// Add your Firebase credentials

firebase.initializeApp({

apiKey: "",

authDomain: "",

projectId: "",

appID: "",

});

const authContext = createContext();

// Provider component that wraps your app and makes auth object ...

// ... available to any child component that calls useAuth().

export function ProvideAuth({ children }) {

const auth = useProvideAuth();

return <authContext.Provider value={auth}>{children}</authContext.Provider>;

}

// Hook for child components to get the auth object ...

// ... and re-render when it changes.

export const useAuth = () => {

return useContext(authContext);

};

// Provider hook that creates auth object and handles state

function useProvideAuth() {

const [user, setUser] = useState(null);

// Wrap any Firebase methods we want to use making sure ...

// ... to save the user to state.

const signin = (email, password) => {

return firebase

.auth()

.signInWithEmailAndPassword(email, password)

.then((response) => {

setUser(response.user);

return response.user;

});

};

const signup = (email, password) => {

return firebase

.auth()

.createUserWithEmailAndPassword(email, password)

.then((response) => {

setUser(response.user);

return response.user;

});

};

const signout = () => {

return firebase

.auth()

.signOut()

.then(() => {

setUser(false);

});

};

const sendPasswordResetEmail = (email) => {

return firebase

.auth()

.sendPasswordResetEmail(email)

.then(() => {

return true;

});

};

const confirmPasswordReset = (code, password) => {

return firebase

.auth()

.confirmPasswordReset(code, password)

.then(() => {

return true;

});

};

// Subscribe to user on mount

// Because this sets state in the callback it will cause any ...

// ... component that utilizes this hook to re-render with the ...

// ... latest auth object.

useEffect(() => {

const unsubscribe = firebase.auth().onAuthStateChanged((user) => {

if (user) {

setUser(user);

} else {

setUser(false);

}

});

// Cleanup subscription on unmount

return () => unsubscribe();

}, []);

// Return the user object and auth methods

return {

user,

signin,

signup,

signout,

sendPasswordResetEmail,

confirmPasswordReset,

};

}

**useDarkMode**

This hook handles all the stateful logic required to add a ☾ dark mode toggle to your website. It utilizes localStorage to remember the user's chosen mode, defaults to their browser or OS level setting using the prefers-color-scheme media query and manages the setting of a .dark-mode className on body to apply your styles.

function App() {

const [darkMode, setDarkMode] = useDarkMode();

return (

<div>

<div className="navbar">

<Toggle darkMode={darkMode} setDarkMode={setDarkMode} />

</div>

<Content />

</div>

);

}

// Hook

function useDarkMode() {

// Use our useLocalStorage hook to persist state through a page refresh.

// Read the recipe for this hook to learn more: usehooks.com/useLocalStorage

const [enabledState, setEnabledState] = useLocalStorage("dark-mode-enabled");

// See if user has set a browser or OS preference for dark mode.

// The usePrefersDarkMode hook composes a useMedia hook (see code below).

const prefersDarkMode = usePrefersDarkMode();

// If enabledState is defined use it, otherwise fallback to prefersDarkMode.

// This allows user to override OS level setting on our website.

const enabled =

typeof enabledState !== "undefined" ? enabledState : prefersDarkMode;

// Fire off effect that add/removes dark mode class

useEffect(

() => {

const className = "dark-mode";

const element = window.document.body;

if (enabled) {

element.classList.add(className);

} else {

element.classList.remove(className);

}

},

[enabled] // Only re-call effect when value changes

);

// Return enabled state and setter

return [enabled, setEnabledState];

}

// Compose our useMedia hook to detect dark mode preference.

// The API for useMedia looks a bit weird, but that's because ...

// ... it was designed to support multiple media queries and return values.

// Thanks to hook composition we can hide away that extra complexity!

// Read the recipe for useMedia to learn more: usehooks.com/useMedia

function usePrefersDarkMode() {

return useMedia(["(prefers-color-scheme: dark)"], [true], false);

}

**useMedia**

This hook makes it super easy to utilize media queries in your component logic. In our example below we render a different number of columns depending on which media query matches the current screen width, and then distribute images amongst the columns in a way that limits column height difference (we don't want one column way longer than the rest).

import { useState, useEffect } from "react";

function App() {

const columnCount = useMedia(

// Media queries

["(min-width: 1500px)", "(min-width: 1000px)", "(min-width: 600px)"],

// Column counts (relates to above media queries by array index)

[5, 4, 3],

// Default column count

2

);

// Create array of column heights (start at 0)

let columnHeights = new Array(columnCount).fill(0);

// Create array of arrays that will hold each column's items

let columns = new Array(columnCount).fill().map(() => []);

data.forEach((item) => {

// Get index of shortest column

const shortColumnIndex = columnHeights.indexOf(Math.min(...columnHeights));

// Add item

columns[shortColumnIndex].push(item);

// Update height

columnHeights[shortColumnIndex] += item.height;

});

// Render columns and items

return (

<div className="App">

<div className="columns is-mobile">

{columns.map((column) => (

<div className="column">

{column.map((item) => (

<div

className="image-container"

style={{

// Size image container to aspect ratio of image

paddingTop: (item.height / item.width) \* 100 + "%",

}}

>

<img src={item.image} alt="" />

</div>

))}

</div>

))}

</div>

</div>

);

}

// Hook

function useMedia(queries, values, defaultValue) {

// Array containing a media query list for each query

const mediaQueryLists = queries.map((q) => window.matchMedia(q));

// Function that gets value based on matching media query

const getValue = () => {

// Get index of first media query that matches

const index = mediaQueryLists.findIndex((mql) => mql.matches);

// Return related value or defaultValue if none

return typeof values[index] !== "undefined" ? values[index] : defaultValue;

};

// State and setter for matched value

const [value, setValue] = useState(getValue);

useEffect(

() => {

// Event listener callback

// Note: By defining getValue outside of useEffect we ensure that it has ...

// ... current values of hook args (as this hook callback is created once on mount).

const handler = () => setValue(getValue);

// Set a listener for each media query with above handler as callback.

mediaQueryLists.forEach((mql) => mql.addListener(handler));

// Remove listeners on cleanup

return () =>

mediaQueryLists.forEach((mql) => mql.removeListener(handler));

},

[] // Empty array ensures effect is only run on mount and unmount

);

return value;

}

**useScript**

import { useState, useEffect } from "react";

// Usage

function App() {

const status = useScript(

"https://pm28k14qlj.codesandbox.io/test-external-script.js"

);

return (

<div>

<div>

Script status: <b>{status}</b>

</div>

{status === "ready" && (

<div>

Script function call response: <b>{TEST\_SCRIPT.start()}</b>

</div>

)}

</div>

);

}

// Hook

function useScript(src) {

// Keep track of script status ("idle", "loading", "ready", "error")

const [status, setStatus] = useState(src ? "loading" : "idle");

useEffect(

() => {

// Allow falsy src value if waiting on other data needed for

// constructing the script URL passed to this hook.

if (!src) {

setStatus("idle");

return;

}

// Fetch existing script element by src

// It may have been added by another intance of this hook

let script = document.querySelector(`script[src="${src}"]`);

if (!script) {

// Create script

script = document.createElement("script");

script.src = src;

script.async = true;

script.setAttribute("data-status", "loading");

// Add script to document body

document.body.appendChild(script);

// Store status in attribute on script

// This can be read by other instances of this hook

const setAttributeFromEvent = (event) => {

script.setAttribute(

"data-status",

event.type === "load" ? "ready" : "error"

);

};

script.addEventListener("load", setAttributeFromEvent);

script.addEventListener("error", setAttributeFromEvent);

} else {

// Grab existing script status from attribute and set to state.

setStatus(script.getAttribute("data-status"));

}

// Script event handler to update status in state

// Note: Even if the script already exists we still need to add

// event handlers to update the state for \*this\* hook instance.

const setStateFromEvent = (event) => {

setStatus(event.type === "load" ? "ready" : "error");

};

// Add event listeners

script.addEventListener("load", setStateFromEvent);

script.addEventListener("error", setStateFromEvent);

// Remove event listeners on cleanup

return () => {

if (script) {

script.removeEventListener("load", setStateFromEvent);

script.removeEventListener("error", setStateFromEvent);

}

};

},

[src] // Only re-run effect if script src changes

);

return status;

}

**useMemo**

React has a built-in hook called useMemo that allows you to memoize expensive functions so that you can avoid calling them on every render. You simple pass in a function and an array of inputs and useMemo will only recompute the memoized value when one of the inputs has changed. In our example below we have an expensive function called computeLetterCount (for demo purposes we make it slow by including a large and completely unnecessary loop). When the current selected word changes you'll notice a delay as it has to recall computeLetterCount on the new word.

import { useState, useMemo } from "react";

// Usage

function App() {

// State for our counter

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

// State to keep track of current word in array we want to show

const [wordIndex, setWordIndex] = useState(0);

// Words we can flip through and view letter count

const words = ["hey", "this", "is", "cool"];

const word = words[wordIndex];

// Returns number of letters in a word

// We make it slow by including a large and completely unnecessary loop

const computeLetterCount = (word) => {

let i = 0;

while (i < 1000000000) i++;

return word.length;

};

// Memoize computeLetterCount so it uses cached return value if input array ...

// ... values are the same as last time the function was run.

const letterCount = useMemo(() => computeLetterCount(word), [word]);

// This would result in lag when incrementing the counter because ...

// ... we'd have to wait for expensive function when re-rendering.

//const letterCount = computeLetterCount(word);

return (

<div style={{ padding: "15px" }}>

<h2>Compute number of letters (slow 🐌)</h2>

<p>

"{word}" has {letterCount} letters

</p>

<button

onClick={() => {

const next = wordIndex + 1 === words.length ? 0 : wordIndex + 1;

setWordIndex(next);

}}

>

Next word

</button>

<h2>Increment a counter (fast ⚡️)</h2>

<p>Counter: {count}</p>

<button onClick={() => setCount(count + 1)}>Increment</button>

</div>

);

}

**useLocalStorage**

mport { useState } from "react";

// Usage

function App() {

// Similar to useState but first arg is key to the value in local storage.

const [name, setName] = useLocalStorage("name", "Bob");

return (

<div>

<input

type="text"

placeholder="Enter your name"

value={name}

onChange={(e) => setName(e.target.value)}

/>

</div>

);

}

// Hook

function useLocalStorage(key, initialValue) {

// State to store our value

// Pass initial state function to useState so logic is only executed once

const [storedValue, setStoredValue] = useState(() => {

if (typeof window === "undefined") {

return initialValue;

}

try {

// Get from local storage by key

const item = window.localStorage.getItem(key);

// Parse stored json or if none return initialValue

return item ? JSON.parse(item) : initialValue;

} catch (error) {

// If error also return initialValue

console.log(error);

return initialValue;

}

});

// Return a wrapped version of useState's setter function that ...

// ... persists the new value to localStorage.

const setValue = (value) => {

try {

// Allow value to be a function so we have same API as useState

const valueToStore =

value instanceof Function ? value(storedValue) : value;

// Save state

setStoredValue(valueToStore);

// Save to local storage

if (typeof window !== "undefined") {

window.localStorage.setItem(key, JSON.stringify(valueToStore));

}

} catch (error) {

// A more advanced implementation would handle the error case

console.log(error);

}

};

return [storedValue, setValue];

}