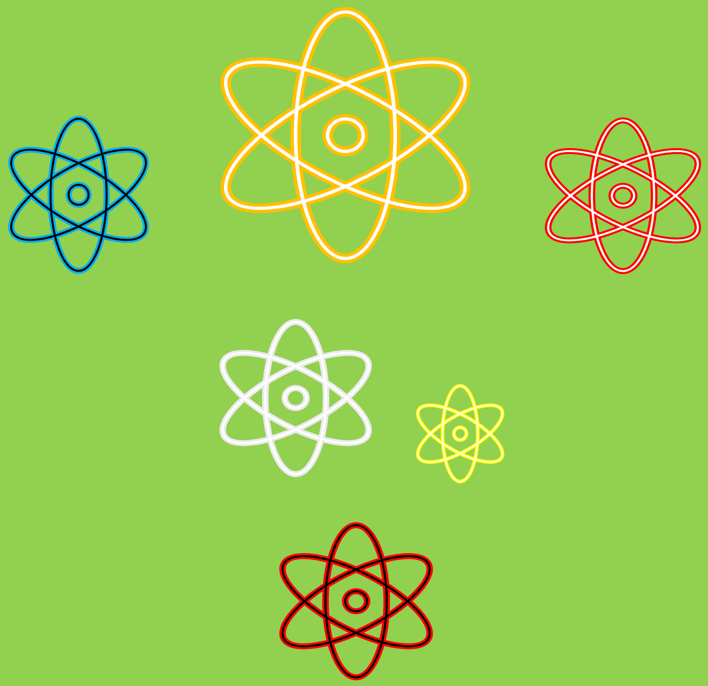
Free Tutorial

React Hooks

for beginners



Satoshi Yoshida
Simona Yoshida

For more information, email address: react3001@gmail.com

Release History: FIRST EDITION September 2021

About this book

This free book is extracted from chapter 4, “React Hooks Redux in 48 hours”, released from Amazon.com in July 2021.

With CodeSandbox examples, you can run sample programs in your browser, no need to waste your time setting up the environment. The example codes used in this book are not for production projects but only for learning the concept. Many diagrams also help you understand visually. The goal is to help you learn the React ecosystem quickly and guide you in the right direction.

Prerequisite:

- Basic knowledge of JavaScript (especially scope and closure)
- Basic knowledge of React classic. (JSX, functional component, props and state)

4. REACT HOOKS

Divide each difficulty into as many parts as is feasible and necessary to resolve it.

Each problem that I solved became a rule, which served afterward to solve other problems.

(René Descartes 1596-1650)

If you are used to functional programming, you will likely enjoy using hooks.
If you come from class-based programming such as Java, ES6 Class components may be more natural. Although hooks are trendy nowadays, there is nothing wrong with using Class components. The choice is mostly a matter of personal preference.

React Hooks allows us to use only functional components for all our needs to avoid complicated patterns. Hooks are a simpler way to encapsulate stateful behavior and side effects in a user interface while using less code and increasing readability.

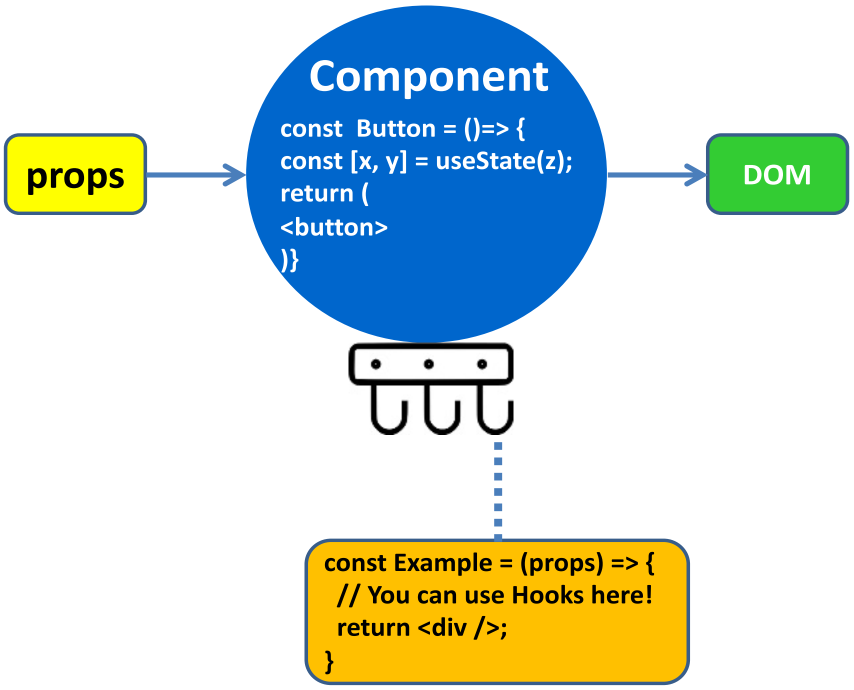
React Hooks let us use state and other React features without using ES6 Classes and allow us to access state and Lifecycle methods in a functional component. Hooks relies on closures to store data.

Before Hooks had been released, the most common way to use state was with ES6 Classes. We had to use *this.state* and *this.setState( )* to refer and update state respectively. With **Hooks** API, setState and ES6 Classes are not needed anymore. No more confusing *this*, but now we have to understand the JavaScript closures.

React hooks present a new set of functionality that allows React to be written in a completely functional manner, opening the door to all sorts of possibilities for performance, lower memory usage, and responsive (non-blocking / concurrent) web applications.

There are only several rules to follow to use Hooks.

- Declare Hooks at the top level in your function components before the return declaration.
- Do not call within a conditional statement, loop statement.
- Do not call within nested functions.
- Do not call from Class components. You can call Hooks from React Functions only. i.e., Functional Components and Custom Hooks.



```
const  Button = ()=> {
const [x, y] = useState(z);
return (
<button>
)}
```

```
const Example = (props) => {
  // You can use Hooks here!
  return <div />;
}
```

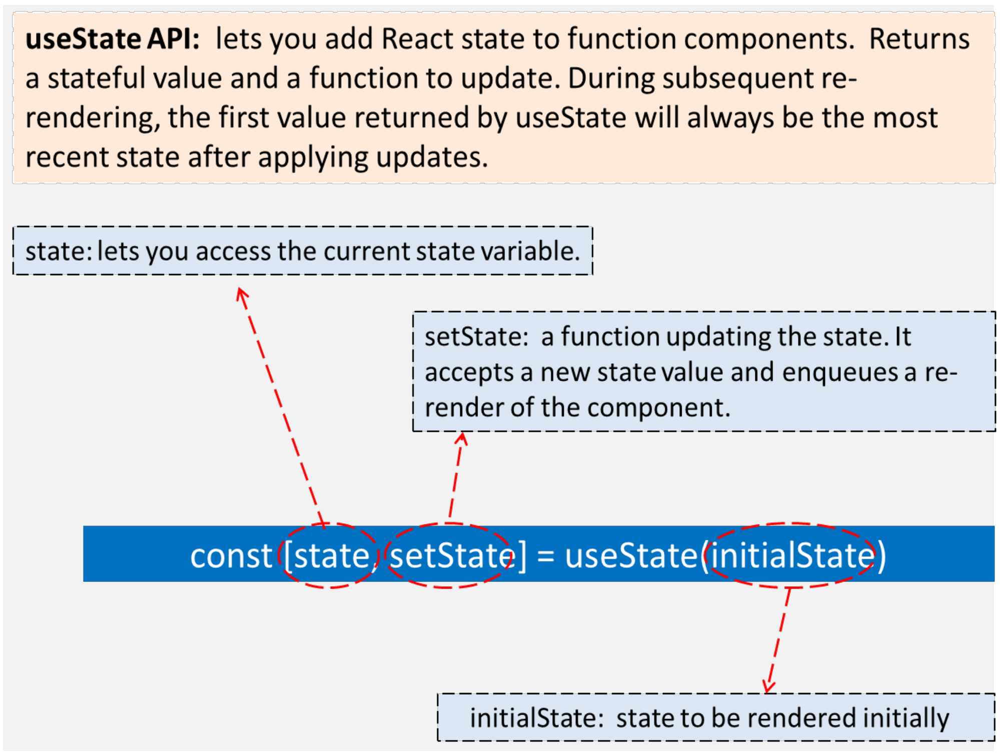
Since React is a component-oriented UI library, creating and managing a global state requires a third-party dependency like Redux. With the introduction of React Hooks, it can be done easily with React without installing any third-party dependency. However, its use case and performance are limited when using in large-scale projects. This means that state managers (such as Redux, Recoil, MobX) are still required for many cases. Hooks and state managers are not mutually exclusive.

The concept of Hooks is easy and clean, but in practice, Hooks APIs are not simple. As a result, you may need considerable time practicing to use it properly.

4.1 useState

As the name implies, **useState** is a hook that allows you to use state in your function. **useState** accepts the initial value of the state item and returns an array containing the state variable and the function you call to update the state. Since it returns an array, we use ES6 array destructuring to access each item, for example, like this:
const [state, setState] = useState(0)

The first variable is the value that is similar to *this.state* in the class components. The second variable is a function to update that value, which is identical to *this.setState*. The final part is the argument that we pass to it. (initial state value).


useState API: lets you add React state to function components. Returns a stateful value and a function to update. During subsequent re-rendering, the first value returned by useState will always be the most recent state after applying updates.
state: lets you access the current state variable.
setState: a function updating the state. It accepts a new state value and enqueues a re-render of the component.
const [state, setState] = useState(initialState)
initialState: state to be rendered initially

Besides the different syntax, useState( ) works differently than setState( ) in class-based components. **Unlike class-based components, the state is not getting merged. When you set a new state with React Hooks, the old state will always be replaced by a new state.**

(Basic pattern 1)
The simplest example is like this. It displays the “Hello world” using the initial value:

```
import React, { useState } from "react";
import ReactDOM from "react-dom"

const App = () => {
  const [greeting] = useState( "Hello world" );

  return (
    <div>
      <h1>{greeting}</h1>
    </div>
  );
};

ReactDOM.render(<App />, document.getElementById("root"));
```

(Basic pattern 2)
You can initialize state variable with function:

```
import React, { useState } from "react";
import ReactDOM from "react-dom";

const App = () => {
  const [greeting] = useState( ( ) => "Hello world" );

  return (
    <div>
      <h1>{greeting}</h1>
    </div>
  );
};

ReactDOM.render(<App />, document.getElementById("root"));
```

(Basic pattern 3)
You can use a function to update the state variable.
The initial message “Hello World” will be replaced by “Hello Mars”:

```
import React, { useState } from "react";
import ReactDOM from "react-dom";

const App = () => {
  const [greeting, updateGreeting] = useState("Hello world" );

  return (
    <div>
      <h2>{greeting}</h2>
      <button onClick={() => updateGreeting("Hello Mars") }>
        Update greeting
      </button>
    </div>
  );
};
ReactDOM.render(<App />, document.getElementById("root"));
```

(Basic pattern 4)
Instead of object, use a function parameter to access the previous state variable.

```
import React, { useState } from "react";
import ReactDOM from "react-dom";
const App = () => {
  const [greeting, updateGreeting] = useState( "Hello world" );
  return (
    <div>
      <h2>{greeting}</h2>
      <button
        onClick={() => updateGreeting((greeting) => `${greeting} + Hello Mars`) }
      >
        Update greeting
      </button>
    </div>
  );
};
ReactDOM.render(<App />, document.getElementById("root"));
```

Code sample: https://codesandbox.io/s/hook-usestate1-f7wn6

In the following example, *count* holds the current state, *setCount* is a function that increments or decrements the count depending on the button clicked.

```
import React, { useState } from "react"
import ReactDOM from "react-dom"

function App() {
  const [count, setCount] = useState(0);
  return (
    <div>
         <h1>count: {count} </h1>
         <button onClick={() => setCount(count + 1)}>increment</button>
         <button onClick={() => setCount(count - 1)}>decrement</button>
    </div>
  )
}

ReactDOM.render(<App />, document.getElementById("root"))
```

Code sample: https://codesandbox.io/s/hook-usestate0-jddjq

useState() confusing part

Much like setState in the class component, the usage of useState is a bit tricky.

1. **<u>Objects being used as a function updater parameter</u>**

In the class component, initialization is in the constructor and runs only one. On the other hand, in the useState hook, every time the component re-renders, initialization takes place. In the below code, no matter how many times setCount(count+1) runs, the result is always 1 because functions overwrite the values.

```
function App() {const [count, setCount] = useState(0);

const incrementCount1 = () => {
        setCount(count + 1);                    //1
        setCount(count + 1);                    //1  (we expect 2)
        setCount(count + 1);                    //1  (we expect 3)
};
```

2. **<u>Functions being used as a function updater parameter</u>**

If you intend to use the previous value to update the state, you must pass a function that receives the previous value and returns an updated value. In the code below, every time setCount((prevCount) => prevCount + 1) are executed, values are incremented from the previous value.

```
function App() {const [count, setCount] = useState(0);

const incrementCount3 = () => {
        setCount((prevCount) => prevCount + 1);    //1
        setCount((prevCount) => prevCount + 1);    //2
        setCount((prevCount) => prevCount + 1);    //3  (as you expect)
 };
```

You can play around with the following code to get used to the subtlety of the useState parameters:

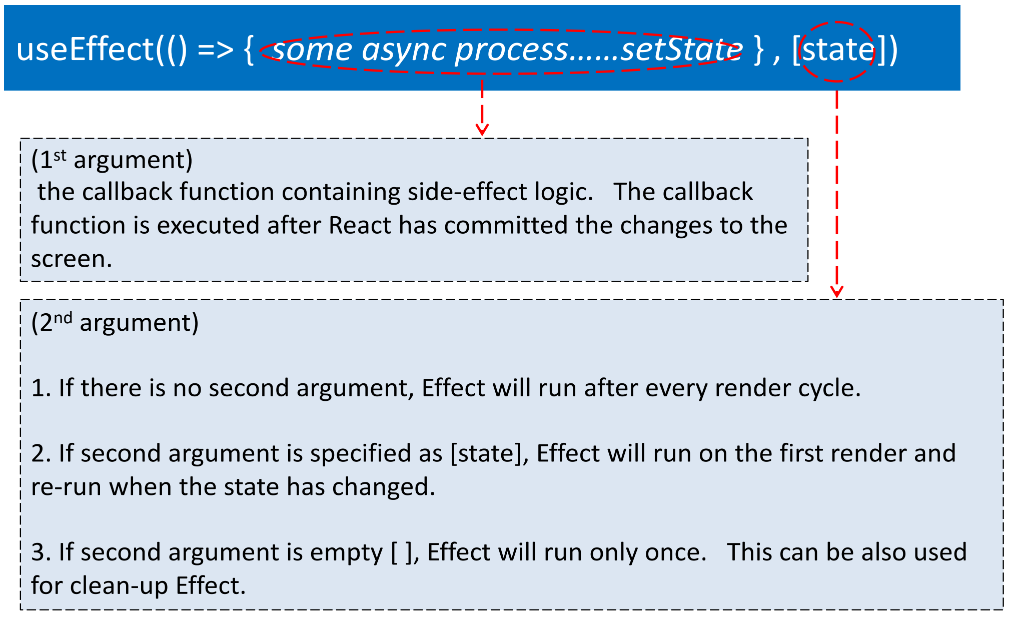
```
//sample code from "learns React/Hooks/Redux in 48 hours"
import React, { useState } from "react";
import ReactDOM from "react-dom";

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

  //plus 1. values based on the Redered values.
  //functions overwite the rendered values each other

const incrementCount1 = () => {
    setCount(count + 1);
    setCount(count + 1);
    setCount(count + 1);
  };
  //plus 3. values based on the previous valus passed into
  const incrementCount3 = () => {
    setCount((prevCount) => prevCount + 1);
    setCount((prevCount) => prevCount + 1);
    setCount((prevCount) => prevCount + 1);
  };
  return (
    <>
      <h1>count: {count} </h1>
      <button onClick={incrementCount1}>increment by 1</button>
      <button onClick={incrementCount3}>increment by 3</button>
    </>
  );
}
ReactDOM.render(<App />, document.getElementById("root"));
```

Code sample : https://codesandbox.io/s/hook-usestate-yvjk2

4.2 useEffect

The **useEffect** hook takes the side-effect function (asynchronous function) as an argument and runs this function whenever the component is mounted to the DOM. **useEffect** is run after the render. The useEffect hook is like handling **componentDidMount**, **componentDidUpdate**, and **componentWillUnmount** all in one call. It should be used for any side-effects that you are executing in your render cycle. By default, effects run after every completed render, but you can also choose to trigger only when specific values have changed. If you don't want it to be executed after each re-render, then you will add a second parameter with an empty array [ ].

useEffect API: The callback function passed to useEffect will run after the render is committed to the screen. Mutations, subscriptions, timers, logging, and other side effects can be used.



useEffect(() => { *some async process……setState* } , [state])

(1st argument)
the callback function containing side-effect logic.   The callback function is executed after React has committed the changes to the screen.

(2nd argument)

1. If there is no second argument, Effect will run after every render cycle.

2. If second argument is specified as [state], Effect will run on the first render and re-run when the state has changed.

3. If second argument is empty [ ], Effect will run only once.   This can be also used for clean-up Effect.

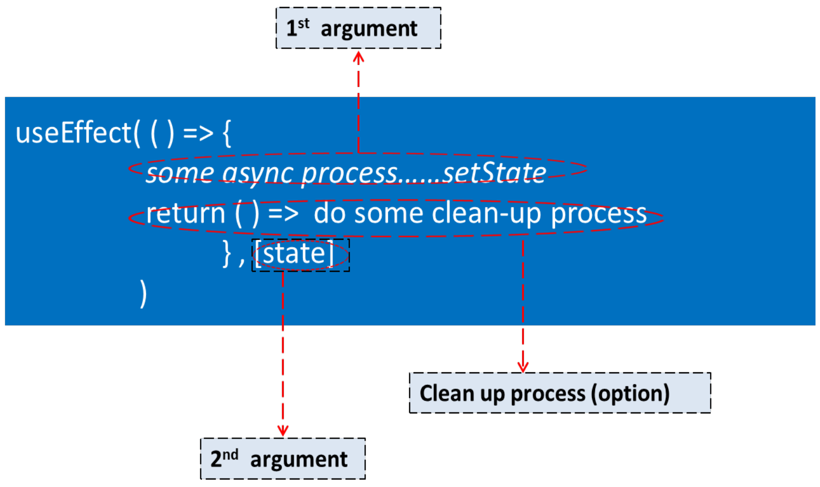
useEffect() can be called multiple times, which is nice feature to separate unrelated logic.

Cleanup after the effect execution

React performs the cleanup when the component unmounts. As **useEffect** can run for every render and not just once, React also needs to clean up effects from the previous render before running the effects next time. Otherwise, it may affect your application performance.

The **useEffect** Hook provides the ability to run a cleanup after the effect. This can be achieved by specifying a return function at the end of the effect.   It is like both **componentDidMount** and **componentDidUnmount** combined into a single effect.

The clean-up function runs before the component is removed from the UI to prevent memory overflows. Additionally, if a component renders multiple times, the previous effect can be cleaned up before executing the next effect.



```
useEffect( ( ) => {
          some async process……setState
          return ( ) =>  do some clean-up process
                  } , [state]
          )
```

Run the effect on every render

React uses the 2nd argument to determine whether it needs to execute the function passed to **useEffect**. If there is no 2nd argument, **useEffect** is run upon every render. This may cause performance issues or just be overkill.

```
import React, { useState, useEffect } from "react";
import ReactDOM from "react-dom";

// useEffect will be called whenever count state changes
function App() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    console.log("effect runs upon render!");
    return () => {
      //clearn-up
      console.log("clean-up done!");
    };
  });
  const handleClick = () => {
    setCount(count + 1);
  };
  return (
    <>
      <h2>count: {count}</h2>
      <button onClick={handleClick}>Click me</button>
    </>
  );
}

const rootElement = document.getElementById("root");
ReactDOM.render(<App />, rootElement);
```

Code sample: https://codesandbox.io/s/hooks-useeffect-ozxnt

Run the effect only once

If we only like to run the effect for a single time, pass an empty [ ] as the 2nd argument to **useEffect**. It is a replacement for **componentDidMount**. The below code accesses the external resource using the Fetch API.

```
import React from "react";
import { useState, useEffect } from "react";
import ReactDOM from "react-dom";

function App() {
  const [url, setUrl] = useState([]);
  //effect will run only once
  useEffect(() => {
    fetch("https://dog.ceo/api/breeds/image/random")
      .then((res) => res.json())
      .then((url) => setUrl(url.message));
  }, []);

  return (
    <>
      <h2>useEffect random dogs - click the reload button</h2>
      <img src={url} alt="loading images...." />
    </>
  );
}

ReactDOM.render(<App />, document.getElementById("root"));
```

Code sample: https://codesandbox.io/s/hooks-dogs-x7zx0rrqqo

Note: In order to focus on understanding the useEffect API concept, error handling procedures are intentionally omitted in this example.

Run the effect when data changes

If you want to run the effect only when a specific value changes (for example, updating some local storage or triggering an HTTP request), you can pass values that you are monitoring.

In the following code, setTimeout is called inside of the useEffect Hook. JavaScript setTimeout( ) function invokes a setCount function after 1 second. As the 2nd argument is specified as [count], the Effect will run on the first render and re-run when the count is changed.

To clear the setTimeout, we need to call the clearTimeout method by setting our timeout variable as an argument.   *return () => clearTimeout(timeout)*

```
import React, { useState, useEffect } from "react"
import ReactDOM from "react-dom"

function App() {
  const [count, setCount] = useState(0)

  useEffect(() => {
    const timeout = setTimeout(() => {
      setCount(count + 1)
    }, 1000)
    return () => clearTimeout(timeout)
  }, [count])

  return <div>count: {count}</div>
}

ReactDOM.render(<App />, document.getElementById("root"))
```

Code sample: https://codesandbox.io/s/039p9zwomv

Using setInterval( ) in useEffect. (a bit tricky)

setInterval( ) and setTimeout( ) are JavaScript methods of Windows object.
setInterval( ) allows us to repeatedly run a function, starting after the interval of time, then repeating continuously at that interval. In contrast, setTimeout( ) runs a function once after the interval of time. If you use setInterval() within react hooks, the component may not work as you expect.

Counter example 1 (may not work as you expect)

By giving [ ] as the second argument, useEffect will call the function once after mounting.
Even though setInterval() is called only once, this code run incorrectly.
The count will increase from 0 to 1 and stay that way.

In the first rendering, the closure captures the count variable. However, the useEffect() is not invoked the second time. Even if the count increases later, it still uses a stale closure. It always has a value of 0 within the setInterval callback. The count is always the same.

The reason is that the callback passed into setInterval's closure that only accesses the variable in the first rendering. It doesn't have access to the new count value in the subsequent render because the useEffect() is not invoked the second time.

```
import React, { useState, useEffect } from "react";
import ReactDOM from "react-dom";

function App() {
  console.log("render");
  const [count, setCount] = useState(0);

  useEffect(() => {
    const interval = setInterval(() => {
      setCount(count + 1);
    }, 1000);
    return () => clearInterval(interval);
  }, [ ]);

  return <div> count: {count}</div>;
}

ReactDOM.render(<App />, document.getElementById("root"));
```

Counter Example 2 (works correctly, but clumsy)

One of the solutions is to add [count] as a dependency. This way, **useEffect( ) has the closure of [count] and correctly handles the reset of interval.** With the dependencies properly set, useEffect( ) updates the closure as soon as the count changes. Proper management of hooks dependencies is an efficient way to solve the stale closure problem.

However, this is not an ideal solution, as a new setInterval( ) will be created and executed on every change of count. You would need to clean it up on every render. Each setInterval( ) is always executed once. Then why not using a setTimeout( ) instead?

```
import React, { useState, useEffect } from "react";
import ReactDOM from "react-dom";

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

  useEffect(() => {
    const interval = setInterval(() => {
      setCount(count + 1);
    }, 1000);
    return () => clearInterval(interval);
  }, [count]);

  return <div> count: {count}</div>;
}

ReactDOM.render(<App />, document.getElementById("root"));
```

Code sample: https://codesandbox.io/s/hooks-useeffect2x-3xep6

Counter Example 3 (Recommended solution)

Like the setState, useEffect hooks have two forms of parameters:
(1) object: takes in the updated state.
(2) function: callback form in which the current state is passed.

You should use the second form and read the latest state value within the callback to ensure that you have the newest state value before incrementing it.

In the following sample, useState updates the previous state without capturing the current value. To do that, we need to provide a callback. This code works correctly and more efficiently. We are using a single setInterval during the lifecycle of a component. The clearInterval will only be called once after the component is unmounted.

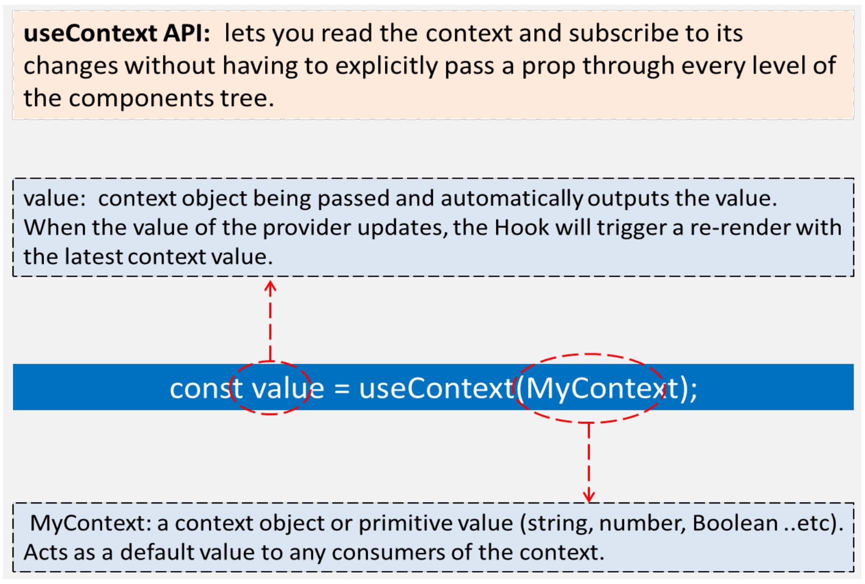
```
import React, { useState, useEffect } from "react";
import ReactDOM from "react-dom";

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

  useEffect(() => {
    const interval = setInterval(() => {
      setCount(count => count + 1);   //uses callback function
    }, 1000);
    return () => clearInterval(interval);
  }, [ ]);

  return <div> count: {count}</div>;
}

ReactDOM.render(<App />, document.getElementById("root"));
```

Code sample: https://codesandbox.io/s/hooks-useeffect3x-ve8zl

4.3 useContext

The React Context API is a simple alternative to prop-drilling up and down your component tree. Instead of passing local data around and through several layers of components, it helps data that needs to be shared across components. The downside is that using React Context will cause unnecessary re-renders for any components (within the Consumer) update.

The useContext hook provides all the same functionality you would expect from the Context API. The useContext lets you use context without using a Consumer keyword, which is a bit simpler than Context API.

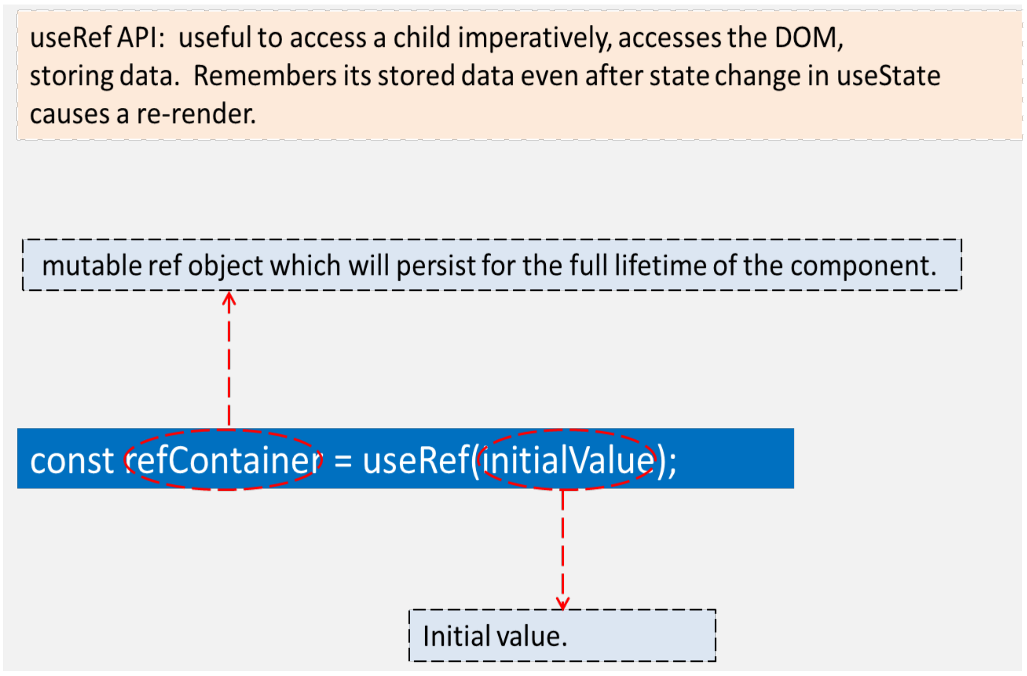

useContext API: lets you read the context and subscribe to its changes without having to explicitly pass a prop through every level of the components tree.
value: context object being passed and automatically outputs the value. When the value of the provider updates, the Hook will trigger a re-render with the latest context value.
const value = useContext(MyContext);
MyContext: a context object or primitive value (string, number, Boolean ..etc). Acts as a default value to any consumers of the context.

```
import React, { useContext } from "react";
import ReactDOM from "react-dom";

// *GreetingContext is created with default value
// *it returns two components Provider and Consumer
const GreetingContext = React.createContext(null);

//* context provider
function App() {
  return (
    <div>
      <GreetingContext.Provider value="World">
        <Post />
      </GreetingContext.Provider>
    </div>
  );
}

// *consuming the GreetingContext data inside Post component
function Post() {
  const theme = useContext(GreetingContext);
  return (
    <div>
      <h1>Hello {theme}</h1>
    </div>
  );}

ReactDOM.render(<App />, document.getElementById("root"));
```

Code sample: https://codesandbox.io/s/usecontext-vo95t

4.4 useRef

useRef hook is a solution to use React *refs* within functional components. *useRef* can be used when you want to avoid re-render or when you want to access the DOM element imperatively. Much like *refs*, *useRef* should not be overused.


useRef API: useful to access a child imperatively, accesses the DOM, storing data. Remembers its stored data even after state change in useState causes a re-render.
mutable ref object which will persist for the full lifetime of the component.
const refContainer = useRef(initialValue);
Initial value.

```
import React, { useState, useRef } from "react";
import ReactDOM from "react-dom";

function App() {
  const [name, setData] = useState("Enter Your Name");

  const dataRef = useRef(null);

  const submitData = () => {
    setData(dataRef.current.value);
  };
      //React will set its current property (dataRef.current.value)
     //to the corresponding DOM node whenever that node changes.
  return (
    <div className="App">
      <p>{name}</p>

      <div>
        <input ref={dataRef} type="text" />
        <button type="button" onClick={submitData}>
          Submit
        </button>
      </div>
    </div>
  );
}

ReactDOM.render(<App />, document.getElementById("root"));
```

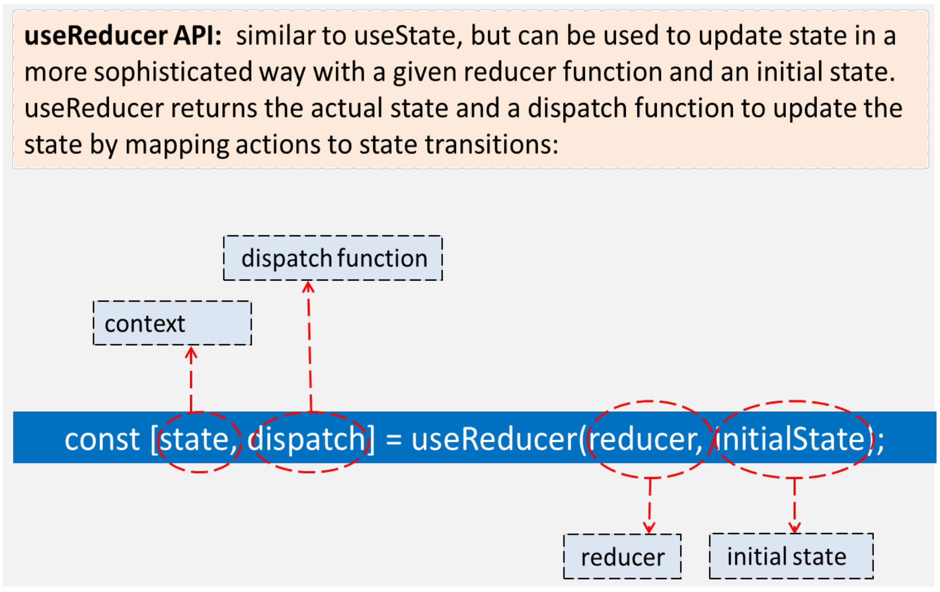
Code sample: https://codesandbox.io/s/useref-7q7zs

4.5 useReducer

useReducer is an alternative to useState. The recommended way to manage component state when your application is more complex, containing objects with multiple sub-values, or when the next state depends on the previous one.

***useReducer* allows you to update parts of your component's state when specific actions are dispatched.** In a way, it is similar to Redux. However, *useReducer* cannot be used as a global state container because the dispatch function operates only on one reducer function.

While Redux creates one global state container located somewhere above your whole application, ***useReducer* creates an independent state container within your component.**



useReducer API: similar to useState, but can be used to update state in a more sophisticated way with a given reducer function and an initial state. useReducer returns the actual state and a dispatch function to update the state by mapping actions to state transitions:

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

Counter example 1

The following example is a counter that increments by 10. When you click the button, value 10 is dispatched. The reducer returns current state + 10.

```
import React, { useReducer } from "react";
import ReactDOM from "react-dom";

function App() {
  const [value, dispatch] = useReducer((state, action) => {
    return state + action;
  }, 0);

  return (
    <div>
        <h2>count: {value}</h2>
        <button onClick={() => dispatch(10)}>increment by 10</button>
    </div>
  );
}

ReactDOM.render(<App />, document.getElementById("root"));
```

Code sample: https://codesandbox.io/s/usereducer-px8ld

Counter example 2

In the following example, a counter will be incremented, decremented, or reset depending on the dispatched action.type. With useRedux, you can easily create a Finite State Machine (FSM):

```
const ACTIONS = {
   INCREMENT: "increment",
   DECREMENT: "decrement",
   RESET: "reset"
};

const reducer = (state, action) => {
  console.log(state, action);
  switch (action.type) {
    case ACTIONS.INCREMENT:
      return { count: state.count + 1 };
    case ACTIONS.DECREMENT:
      return { count: state.count - 1 };
    case ACTIONS.RESET:
      return { count: (state.count = 0) };
    default:
      return state;
  }
};

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

  const increment = () => {dispatch({ type: "increment" });
  };
  const decrement = () => {dispatch({ type: "decrement" });
  };
  const reset = () => {dispatch({ type: "reset" });
  };
  return (
    <div>
      <h2>count: {state.count}</h2>
      <button onClick={increment}>+</button>
      <button onClick={decrement}>-</button>
      <button onClick={reset}>Reset</button>
    </div>
  );
}
ReactDOM.render(<App />, document.getElementById("root"));
```

Reducer to update the state based on the action.type

Dispatch action.type selected by user

Code sample: https://codesandbox.io/s/usereducer1-qrx0f

Todo list example

With useReducer, the Todo list is simpler than using useState.

```
function App() {
  const [todoValue, setTodoValue] = useState();
  let dataRef = useRef(null);

//***reducer***
  const reducer = (state, action) => {
    switch (action.type) {
      case "add":                          //*add todo item and ID
        setTodoValue("");
        return [...state, { id: state.length + 1, text: todoValue }];

      case "remove":                      //*remove the selected todo item
        const newList = state.filter((item) => item.id !== action.id);
        return newList;
      default:
        return state;
    }
  };
//***end reducer***
  const [state, dispatch] = useReducer(reducer, []);

  return (
    <div className="App">
      <form onSubmit={(e) => e.preventDefault()}>
          <input type="text"  ref={dataRef} onChange={(e) => setTodoValue(e.target.value)}/>
          <button onClick={ ( ) => dispatch({ type: "add", text: todoValue }) }>
             Add Todo
          </button>
      </form>
      <div>
        {state.map((todo) => (
          <div className="Row" key={todo.id}>
            <p>{todo.text}</p>
            <button onClick={ ( ) => dispatch({ type: "remove", id: todo.id }) }>
               Remove
            </button>
          </div>
        ))}
      </div>
    </div>
  );
}
ReactDOM.render(<App />, document.getElementById("root"));
```

Code sample: https://codesandbox.io/s/usereducer-todo-nid5i

If the action.type is "add", then add a todo item and generate its id.

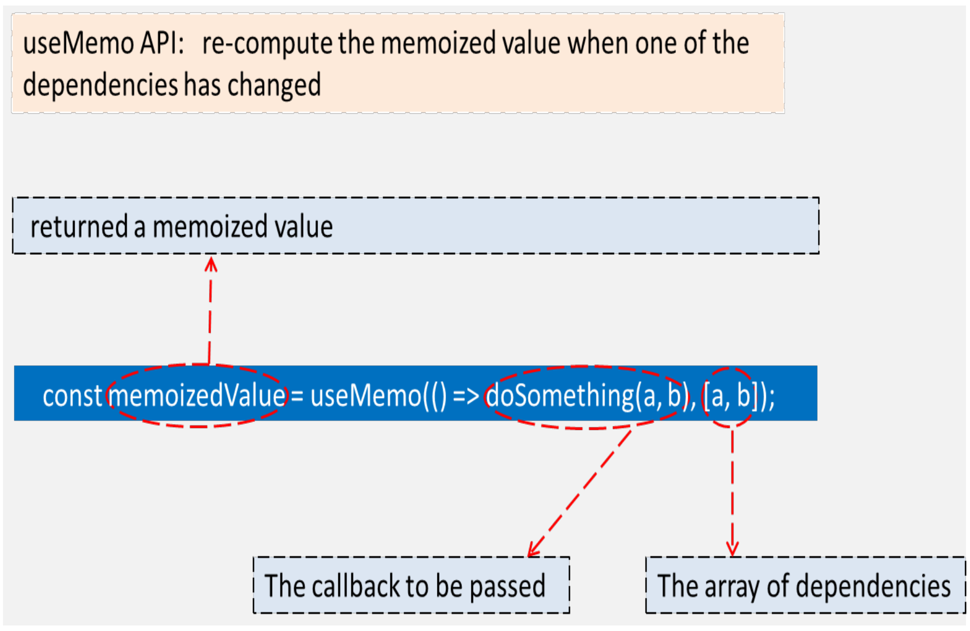
If the action.type is "remove", since we already have a value of id property, we simply filter the source array against this id. The filter method removes items that have a value of id equal to the target id.

```
const reducer = (state, action) => {
  switch (action.type) {
    case "add":                //*add todo item and ID
      setTodoValue("");
      return [...state, { id: state.length + 1, text: todoValue }];

    case "remove":             //*remove the selected todo item
      const newList = state.filter((item) => item.id !== action.id);
      return newList;

    default:
      return state;
  }
};
```

4.6 useMemo

In a nutshell, memoization is the programming pattern to make long recursive/iterative functions run much faster. When React checks for changes in a component, it may cause unnecessarily re-rendering to the component tree. This is where a software technique called memoization comes in.

useMemo prevents unnecessary re-renders, making your code way more efficient by returning a **memoized value** resulting from the callback function.


useMemo API: re-compute the memoized value when one of the dependencies has changed
returned a memoized value
The callback to be passed
The array of dependencies

```
const memoizedValue = useMemo(() => doSomething(a, b), [a, b]);
```

In the following example, useMemo allows you to cache the value of a variable along with a dependency list (*memoCount)*. If any variables in this dependency list change, React will re-run the processing for this data and re-cache it.

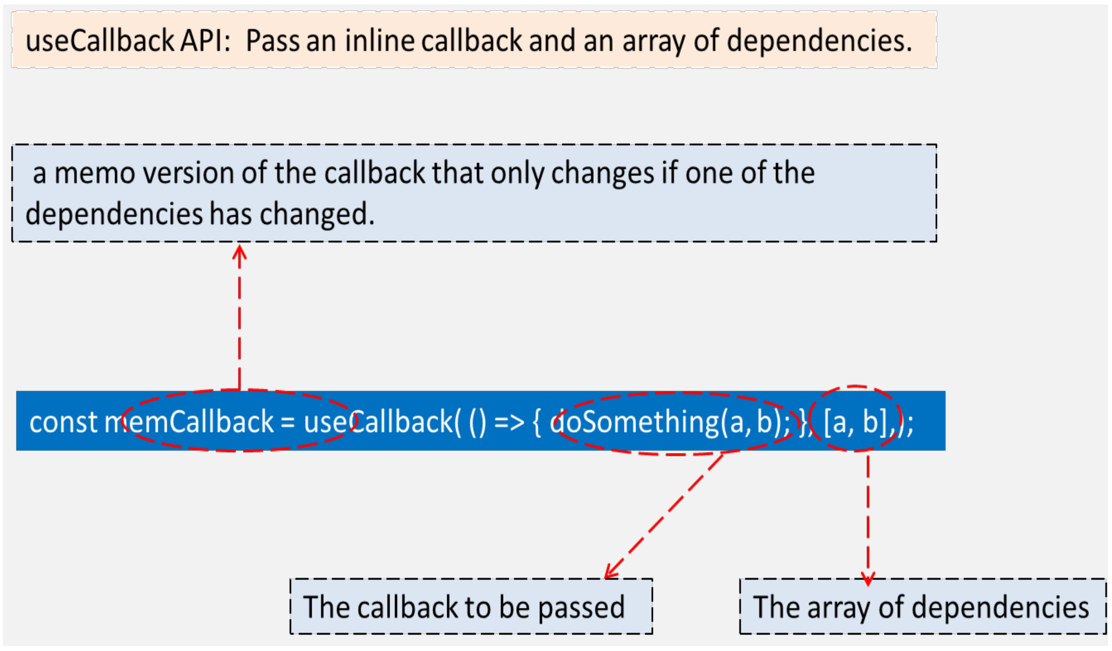
If the variable value in the dependency list was previously cached, React gets the value from the cache. In this way, you can improve the performance of repetitive processing.

```
import React, { useState, useMemo } from "react";
import ReactDOM from "react-dom";

const App = () => {
  const [memoCount, setMemoCount] = useState(0);
  const memoFunction = () => {
    console.log("memoized value", memoCount);
     /*  You can add code that requires heavy processing */
  };

  /* create the memo hook, when memoCount changes,  memoFunction will be executed again */
  useMemo(memoFunction, [memoCount]);

   /* After creating useMemo, each change of memoCount triggers the function passed to the hook,
      otherwise the memoized value will be returned */

  return (

    <div>
      <button onClick={() => setMemoCount(memoCount + 1)}>
        Increment memo count
      </button>
      <p>click the Console below</p>
    </div>
  );
};

ReactDOM.render(<App />, document.getElementById("root"));
```

Code sample: https://codesandbox.io/s/usememo-b29ug

4.7 useCallback

useCallback prevents unnecessary re-renders, making your code more efficient by returning a memoized callback function. *useCallback* helps memoize the callback function, which requires heavy computation. ***useCallback* returns its function when the dependencies change.** (On the other hand, *useMemo* calls its function and returns the memoized value.)


useCallback API: Pass an inline callback and an array of dependencies.
a memo version of the callback that only changes if one of the dependencies has changed.
The callback to be passed
The array of dependencies

```
const memCallback = useCallback( () => { doSomething(a, b); }, [a, b],);
```

```
import React, { useEffect, useState, useCallback } from "react";
import ReactDOM from "react-dom";

const App = () => {
  const [callbackCount, setCallbackCount] = useState(0);

  /* if we give an empty array of dependencies,
     the callback function will return the old value of callbackCount
     because useCallback will return its memoized version */

  const callbackFunction = useCallback(() => {
    console.log("callback", callbackCount);
    return callbackCount;
  }, [callbackCount]);

  return (
    <>
      {/* It will receive a function that will change when the dependency value changes */}
      <Child action={callbackFunction} />

      {/* Change the callback hook dependency to trigger a change in the child */}
      <button onClick={() => setCallbackCount(callbackCount + 1)}>
        Increment callback count
      </button>
      <p>click the Console below</p>
    </>
  );
};

const Child = ({ action }) => {
  const [value, setValue] = useState(0);

  useEffect(() => {
    const number = action();
    setValue(number);
  }, [action]);

  return <p>callback count: {value}</p>;
};


ReactDOM.render(<App />, document.getElementById("root"));
```

Code sample: https://codesandbox.io/s/usecallbak-rnhci

4.8 Custom Hooks

You can create your own custom hooks to avoid code repetition and make the code cleaner. Custom hooks are normal JavaScript functions that can use other hooks with the *use* prefix. Custom hooks are just functional components, and so it is easy to understand.
The general rules of hooks also apply to custom hooks:

- Only call hooks at the top level. Do not call hooks inside loops, conditions, or nested functions. React relies on the order in which hooks are called to associate the hooks with a specific local state. Placing a hook inside conditions may change this order resulting in subsequent hooks failing to get called.
- Only call hooks from React function components. You cannot call hooks from regular JavaScript functions.
- You can call custom hooks from your own custom hooks.

In the following example, we create a custom hook *useCounter* that processes increment, decrement or reset, depending on the input parameters.

```
import React, { useState } from "react";
import ReactDOM from "react-dom";

//  custom hook to count up/down/reset
const useCounter = (initialValue) => {
  const [count, setCount] = useState(initialValue);
  return {
    value: count,
    increase: () => setCount((prevCount) => prevCount + 1),
    decrease: () => setCount((prevCount) => prevCount - 1),
    reset: () => setCount(initialValue)
  };
};

//  counter calling useCounter hook.
const Counter = ({ initialValue }) => {
  const counter = useCounter(initialValue);
  return (
    <>
      <h2>count: {counter.value}</h2>
      <button onClick={counter.increase}>increment</button>
      <button onClick={counter.decrease}>decrement</button>
      <button onClick={counter.reset}>Reset</button>
    </>
  );
};

function App() {
  return (
    <div className="App">
      <Counter initialValue={0} />
    </div>
  );
}


ReactDOM.render(<App />, document.getElementById("root"));
```

Code sample: https://codesandbox.io/s/custom-hooks-v02fj

In the following example, we create a custom hook *useClock* that displays a local time by calling a JavaScript Date method.

```
import React from "react";
import ReactDOM  from "react-dom";
import { useState, useEffect } from "react";

import "./index.css";
//using custom hook
const App = () => {
  return (
    <div>
      <h1>Current local time</h1>
      <br />
      <h2> {useClock()} </h2>
      <h3> {useClock()} </h3>
      <h4> {useClock()} </h4>
    </div>
  );
};
//custome hook
const useClock = () => {
  const [clock, setClock] = useState(new Date().toLocaleTimeString());

  useEffect(() => {
    const myInterval = setInterval(() => {
      setClock(new Date().toLocaleTimeString());
    }, 1000);

    return () => clearInterval(myInterval);
  }, []);

  return clock;
};

ReactDOM.render(<App />, document.getElementById("app"));
```

Code sample: https://codesandbox.io/s/react-custom-hooks1-c6obt

You need to click the reload button to start.

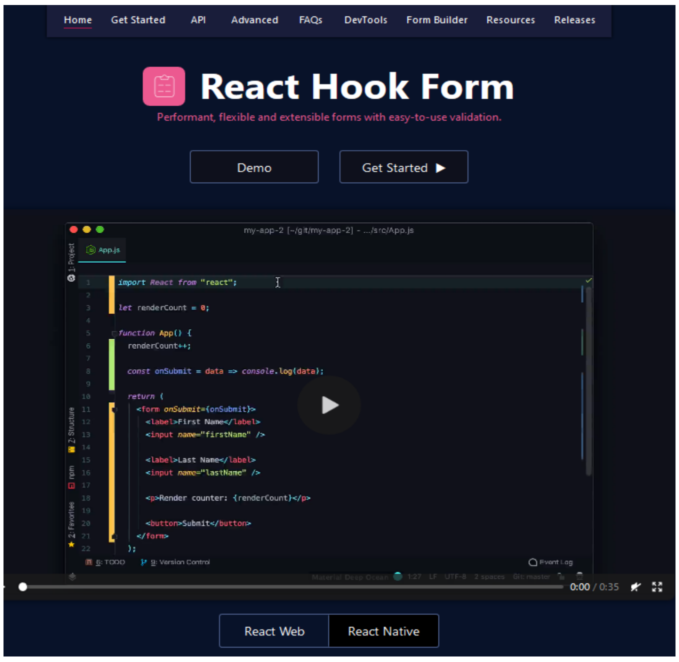
4.9 React Hook Form library

React Hook Form is the third-party custom hooks library, having a clean structure with a single hook call abstracting away many underlying setups, and it is flexible for many different situations. It includes local validation, the ability to pass in a validation schema, ability to change output shape within its form. PropTypes becomes much easier for defining your form components because React Hook Form manages it. React Hook Form must be used within a functional component.

Try this CodeSandbox sample from the official docs.
https://codesandbox.io/s/kw7z2q2n15

We encourage you to take a closer look if you think it might be for you.

official docs: https://react-hook-form.com/



```
import React from "react";

let renderCount = 0;

function App() {
  renderCount++;

  const onSubmit = data => console.log(data);

  return (
    <form onSubmit={onSubmit}>
      <label>First Name</label>
      <input name="firstName" />

      <label>Last Name</label>
      <input name="lastName" />

      <p>Render counter: {renderCount}</p>

      <button>Submit</button>
    </form>
  );
```

4.10 SWR

SWR (Stale-While-Revalidate) is a data fetching library made by the team at ZEIT, the same team that created Next.js. This new library is made for React applications as a custom hook with a straightforward interface.

SWR first returns the data from cache (stale), then sends the fetch request (revalidate), and finally comes with the up-to-date data. With SWR, components will get a stream of data updates constantly and automatically.

The following example demonstrates the fundamental API of SWR. The useSWR hook accepts a key string and a fetcher function. A key is a unique identifier of the data (the URL API) and passed to the fetcher. The fetcher is an asynchronous function that returns the data. The hook returns two values: data and error, based on the status of the request. In this example, we use fetch API, but you can use any library you like (such as Axois).

```
import useSWR from "swr";

const fetcher = (url) => fetch(url).then((res) => res.json());

export default function App() {
  const { data, error } = useSWR(
    "https://dog.ceo/api/breeds/image/random",
    fetcher
  );

  if (error) return "An error has occurred.";
  if (!data) return "Loading...";
  return (
    <div>
      <h2>SWR random dogs</h2>
      <h4>click the reload button to change</h4>
      <img src={data.message} alt="loading images...." />
    </div>
  );
}
```

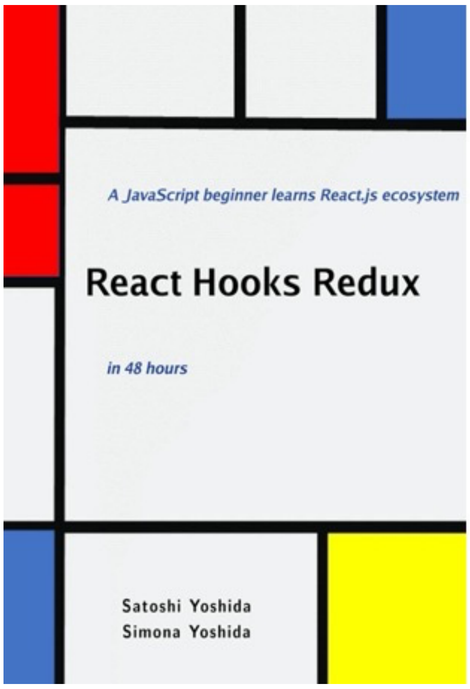
Sample code: https://codesandbox.io/s/react-swr1-f7nom

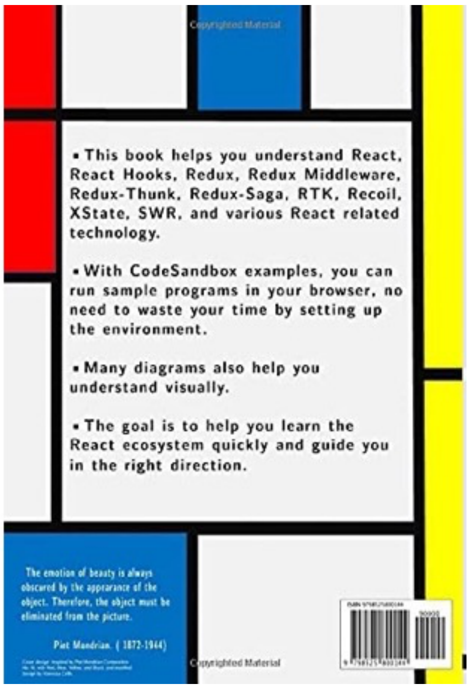
SWR official docs: https://swr.vercel.app/

If this free book helps you understand Hooks, we would recommend the following book for you. You can learn JavaScript, React, Hooks, and Redux in a structured learning method.

React Hooks Redux in 48 hours:
A JavaScript beginner learns React.js ecosystem

by Satoshi Yoshida, Simona Yoshida





- This book helps you understand React, React Hooks, Redux, Redux Middleware, Redux-Thunk, Redux-Saga, RTK, Recoil, XState, SWR, and various React related technology.
- With CodeSandbox examples, you can run sample programs in your browser, no need to waste your time by setting up the environment.
- Many diagrams also help you understand visually.
- The goal is to help you learn the React ecosystem quickly and guide you in the right direction.

The emotion of beauty is always obscured by the appearance of the object. Therefore, the object must be eliminated from the picture.

Piet Mondrian. (1872-1944)

ISBN-13 : 979-8525800144

Learning React with CodeSandbox

This book helps you learn ReactJS, React Hooks, Redux, Redux Middleware, Redux-Thunk, Redux-Saga, RTK, Recoil, XState, SWR, React Router, Jest, React Test Library, and various React related technology.

With 99 CodeSandbox code examples we have created, you can run sample programs in your browser, no need to waste your time by setting up the environment. The goal is to help you learn the React ecosystem quickly and guide you in the right direction.

Who this book is for:

- JavaScript beginners: some experience in Web design using HTML, CSS, and JavaScript.
- JavaScript programmers who are not familiar with functional programming and ES6.
- Busy developers who do not have time to go through React/Redux official documentations.
- Non-Web software engineers, managers, or system architects: at least know one of the high-level languages.

This book is not for:

- No experience in any programming.
- React experts: Please refer to the React and Redux official docs.

Learning schedule:

- Overview (one hour reading)
- Intermediate JavaScript (6 hours reading and exercise)
- React Fundamentals (10 hours reading and exercise)
- React Hooks (10 hours reading and exercise)
- React Router (2 hours reading and exercise)
- Test-Driven Development (2 hours reading and exercise)
- Redux, Middleware, Thunks, Saga, and RTK (12 hours reading and exercise)
- Recoil (2 hours reading and exercise)
- Various State Management Libraries (one hour reading and exercise)
- APPENDIX - various libraries/frameworks (2 hours reading and exercise)

CodeSandbox environment:

You will need a Windows PC, Mac, Chromebook, Android device, iOS device, or Linux-based device. We have tested all code on Chrome, Firefox, and Edge browser.

About an author

Satoshi Yoshida started his professional carrier in the 1980s, developing CPU circuits, mobile phone modem LSIs and firmware. He also had management and architect experiences in various multinational companies in Japan, Canada, France, and Germany. He has 16 patents in wireless communication and has been working as a freelance consultant (wireless hardware, IoT, smart metering, UI software) since 2018.

About a co-author

Simona Yoshida has been working on web-based UI software and e-commerce platform since 2015. Sample codes in chapter 2, chapter 3, and chapter 4 were written and tested by her. In her spare time, she takes care of dogs and cats from animal shelters.

Contact information: react3001@gmail.com

Recommendation by BookAuthority.org:

- **14 Best New React Hooks Books To Read In 2021**

As featured on CNN, Forbes and Inc – BookAuthority identifies and rates the best books in the world, based on recommendations by thought leaders and experts.

https://bookauthority.org/books/new-react-hooks-books

How to order the "React Hooks Redux in 48 hours"

You can order it from Amazon in your country/region.

Paperback	ASIN : B097SSBP2V
eBook	ASIN : B0987SZHW4

Amazon.com	(USA)
Amazon.co.uk	(UK)
Amazon.de	(Germany)
Amazon.fr	(France)
Amazon.es	(Spain)
Amazon.it	(Italy)
Amazon.nl	(Netherlands)
Amazon.co.jp	(Japan)
Amazon.in	(India)
Amazon.ca	(Canada)
Amazon.com.br	(Brasil)
Amazon.com.mx	(Mexico)
Amazon.com.au	(Australia)