Introduction: React Hooks introduction

If you don't like classes then hooks are here to help you. Hooks are functions that enable you to use React functionalities like state and lifecycle features without using classes. As the name suggests, it enables you to hook into React state and lifecycle features from function components.

Hooks do not work inside classes and are backward-compatible, meaning they have no breaking changes. So, it is your choice if you want to use them or not. This new feature gives you the power to use all React features, even the function components.

Make sure you write hooks in standard follow form. Call hooks at the top level of React functions to ensure that hooks are called in the same order every time. Try to avoid calling them in loops, nested functions, or inside conditions. Also, make sure you call them from the React function component, not from JavaScript functions. If you don't follow that rule then it may result in some unexceptional behaviors. React also provides a linter plugin to ensure that these rules apply automatically.

You don't have to install anything to use hooks. They come with React from the 16.8 version onward.

useState Hook

To declare, change and use states using hooks we have useState() function component. It returns two value pairs, the current state, and a function to update the state. It is something like this.setState in a class but it is different from this.setState by the fact that it does not combine the old and new state. UseState() can be called inside a function component or from an event handler. The initial state is passed inside useState() as an argument which is used during the first render. There is no bound on using multiple state hooks in a single component; we can use as many as we want.

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

export default function App() {
  const [counter, setCounter] = useState(1);
  const incrBy3 = () => {
    setCounter(counter + 3);
  };
  return (
    <div className="container">
      <h1>Increment By 3</h1>
      <div className="counter-box">
        <span>{counter}</span>
        <button onClick={incrBy3}>+ 3</button>
      </div>
    </div>
  );
}
```

Interesting Facts of useState Hook

A few points to emphasize here that we often ignore.

With the useState hook, the state gets created only at the first render using the initial value we pass as an argument to it.
For every re-render (subsequent renders after the initial render), ReactJS ignores the initial value we pass as the argument. In this case, it returns the current value of the state.
ReactJS provides us with a mechanism to get the previous state value when dealing with the current state value.
That's all about the interesting facts, but they may not make much sense without understanding their advantages. So, there are two primary advantages,

We can perform a lazy initialization of the state.
We can use the previous state value alongside the current one to solve a use case.

How to perform Lazy Initialization of the State?

If the initial state value is simple data like a number, string, etc., we are good with how we have created and initialized the state in the above example. At times, you may want to initialize the state with a computed value. The computation can be an intense and time-taking activity.

With the useState hook, you can pass a function as an argument to initialize the state lazily. As discussed, the initial value is needed only once at the first render. There is no point in performing this heavy computation on the subsequent renders.

```
const [counter, setCounter] = useState(() => Math.floor(Math.random() * 16));
```

The code snippet above lazily initializes the counter state with a random number. Please note, you don't have to do this always, but the knowledge is worthy. Now you know you have a way to perform lazy state initialization.

useState Previous state

The useState hook returns a function to update the state. In our example, we know it as the setCounter(value) method. A specialty of this method is, you can get the previous(or old) state value to update the state. Please take a look into the code snippet below,

```
const incrBy3 = () => {
   setCounter((prev) => prev + 3);
};
```

Here we pass a callback function to the setCounter() method that gives us the previous value to use. Isn't that amazing?

useState with object

One of React's most commonly used Hooks is useState, which manages states in React projects as well as objects' states. With an object, however, we can't update it directly or the component won't rerender.

To solve this problem, we'll look at how to use useState when working with objects, including the method of creating a temporary object with one property and using object destructuring to create a new object from the two existing objects.

In the following code sample, we'll create a state object, shopCart, and its setter, setShopCart. shopCart then carries the object's current state while setShopCart updates the state value of shopCart

```
const [shopCart, setShopCart] = useState({});

let updatedValue = {};
updatedValue = { "item1": "juice" };
setShopCart(shopCart => ({
  ...shopCart,
  ...updatedValue
}));
```

We can then create another object, updatedValue, which carries the state value to update shopCart.

By setting the updatedValue object to the new {"item1":"juice"} value, setShopCart can update the value of the shopCart state object to the value in updatedValue.

useState with array

Arrays are mutable in JavaScript, but you should treat them as immutable when you store them in state. Just like with objects, when you want to update an array stored in state, you need to create a new one (or make a copy of an existing one), and then set state to use the new array.'

In React, Let's first create a friends array. We will have two properties, name, and age.

```
const friendsArray = [
   {
      name: "John",
      age: 19,
   },
   {
      name: "Candy",
      age: 18,
   },
   {
      name: "mandy",
      age: 20,
   },
];
```

Now let's work with this array and useState

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

const App = () => {
```

```
  const [friends, setFriends] = useState(friendsArray); // Setting default value

  const handleAddFriend = () => {
    ...
  };

  return (
    <main>
      <ul>
        // Mapping over array of friends
        {friends.map((friend, index) => (
          // Setting "index" as key because name and age can be repeated, It will be
better if you assign uniqe id as key
          <li key={index}>
            <span>name: {friend.name}</span>{" "}
            <span>age: {friend.age}</span>
          </li>
        ))}
        <button onClick={handleAddFriend}>Add Friends</button>
      </ul>
    </main>
  );
};
export default App;
```

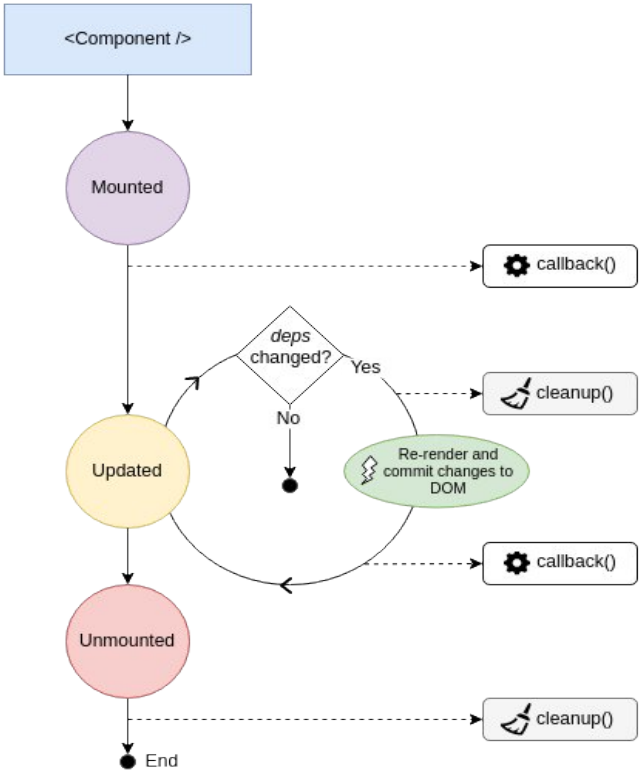
Here, we are mapping over a friends array and displaying it.

Let's now see How to add new values to this array

```
const handleAddFriend = () => {
  setFriends((prevFriends) => [
    ...prevFriends,
    {
      name: "Random Friend Name",
      age: 20, // Random age
    },
  ]);
};
```

Here, setState lets us define an anonymous function that has its previews state as an argument to the function, then we are using spread operator to get our all previews value(state) now after this we can add our new value.

useEffect: useEffect Hook



To perform side effects operations like changing DOM, data fetching, etc., from React function components we use Effect hook, i.e., **useEffect**. These operations are called so since they can affect other components and can't be performed during rendering. **UseEffect** hook provides us the power of **componentDidMount**, **componentDidUpdate**, and **componentWillUnmount**.

UseEffect is declared inside the components because they need to have access to their state and props. UseEffect can be run at first render, after every render, or used to specify clean up based on how we declare them. By default, they run after every render including the first render.

```
import React, {
  useState,
  useEffect
} from 'react';

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

  /* default behavior is similar to componentDidMount and componentDidUpdate: */
  useEffect(() => {
```

```
  // Change document title
  document.title = `Clicked {count} times`;
 });

 return (
  <div>
   <p>Clicked count = {count}</p>
   <button onClick={() => setCount(count + 1)}>
    Button
   </button>
  </div>
 );
}
```

In the above example we are updating the document title every time the count gets updated. Here useEffect is working similarly to componentDidMount and componentDidUpdate combined since it will run during the first render and also after every update. This is the default behavior of useEffect, but it can be changed. Let's see how.

useEffect after render

If you want to use useEffect as componentDidMount then you can pass an empty array [] in the second argument to useEffect as in the below example:

```
import React, {
 useState,
 useEffect
} from 'react';

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

 // Similar to componentDidMount
 useEffect(() => {
  // Only update when component mount
  document.title = `You clicked {count} times`;
 }, []);

 return (
  <div>
   <p>Your click count is {count}</p>
   <button onClick={() => setCount(count + 1)}>
    Button
```

```
    </button>
   </div>
  );
}
```

Conditionally run effects, run effects only once

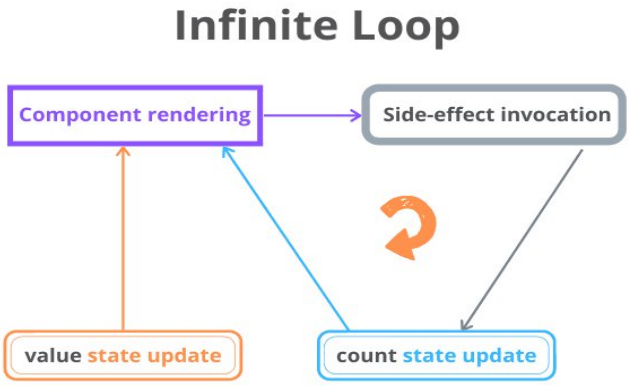
If you want to use it as componentDidUpdate, which only re-renders if a specific state is updated, then we can pass that state value in the second argument of useEffect as below:

```
useEffect(() => {
  document.title = `Clicked {count} times`;
}, [count]); /* if count changes then only effect is re-run */
The above code is similar to the below code when we use classes.

componentDidUpdate(prevProps, prevState) {
  if (prevState.count !== this.state.count) {
    document.title = `Clicked {this.state.count} times`;
  }
}
```

useEffect with cleanup,
If you also want it to perform the componentWillUnmount function, (which can be used for clean up), then we can return value from useEffect.

```
useEffect(() => {
  function handleStatusChange(status) {
    setStatus(status);
  }

  API.changeStatusToTrue(props.status, handleStatusChange);
  return () => {
    /* for implementing componentDidUnmount behaviour */
    API.changeStatusToFalse(props.status, handleStatusChange);
  };
}, [props.status]); /* re-run only if props.status changes */
```

useEffect with incorrect dependency



useEffect() hook manages the side-effects like fetching over the network, manipulating DOM directly, and starting/ending timers.

Although the useEffect() is one of the most used hooks along with useState(), it requires time to familiarize and use correctly.

A pitfall you might experience when working with useEffect() is the infinite loop of component renderings. In this post, I'll describe the common scenarios that generate infinite loops and how to avoid them.

Let's say you want to create a component having an input field, and also display how many times the user changed that input.

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

function CountInputChanges() {
  const [value, setValue] = useState('');
  const [count, setCount] = useState(-1);

  useEffect(() => setCount(count + 1));
  const onChange = ({ target }) => setValue(target.value);
  return (
    <div>
      <input type="text" value={value} onChange={onChange} />
      <div>Number of changes: {count}</div>
    </div>
  )
}
```

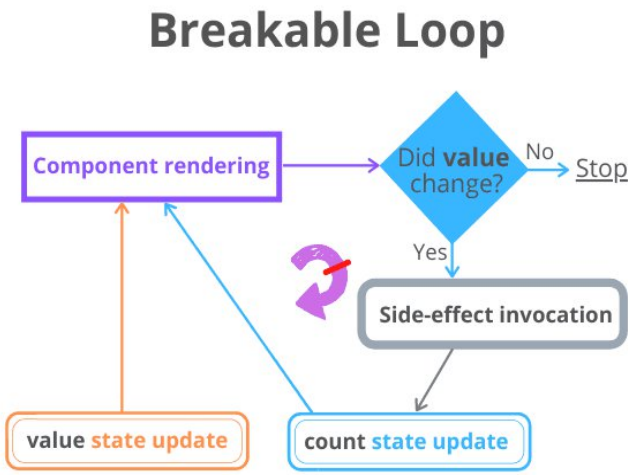
After initial rendering, useEffect() executes the side-effect callback and updates the state. The state update triggers re-rendering. After re-rendering useEffect() executes the side-effect callback and again updates the state, which triggers again a re-rendering. ...and so on indefinitely.

The infinite loop is fixed with correct management of the useEffect(callback, dependencies) dependencies argument.

Because you want count to increment when value changes, you can simply add value as a dependency of the side-effect:

```
import { useEffect, useState } from 'react';
function CountInputChanges() {
  const [value, setValue] = useState('');
  const [count, setCount] = useState(-1);
  useEffect(() => setCount(count + 1), [value]); // Solution
  const onChange = ({ target }) => setValue(target.value);
  return (
    <div>
      <input type="text" value={value} onChange={onChange} />
      <div>Number of changes: {count}</div>
    </div>
  );
}
```

By adding [value] as a dependency of useEffect(..., [value]), the count state variable will only be updated when [value] changes. This solves the infinite loop.



Now, as soon as you type into the input field, the count state correctly displays the number of input value changes.

Fetching data:
Fetching data with useEffect

Side-effects then, are operations that change things outside of your function, making the function impure.

Fetching data from an API, communicating with a database, and sending logs to a logging service are all considered side-effects, as it's possible to have a different output for the same input. For example, your request might fail, your database might be unreachable, or your logging service might have reached its quota.

This is why useEffect is the hook for us - by fetching data, we're making our React component impure, and useEffect provides us a safe place to write impure code.

You might notice a few things are missing from this example:

- we're not doing anything with the data once we fetch it
- we've hardcoded the URL to fetch data from

```
useEffect(() => {
  const fetchData = async () => {
    const response = await fetch(`https://swapi.dev/api/people/1/`);
    const newData = await response.json();
  };

  fetchData();
});
```

To make this useEffect useful, we'll need to:

- update our useEffect to pass a prop called id to the URL,
- use a dependency array, so that we only run this useEffect when id changes, and then
- use the useState hook to store our data so we can display it later

```
import React, { useEffect, useState } from 'react';

export default function DataDisplayer(props) {
  const [data, setData] = useState(null);

  useEffect(() => {
    const fetchData = async () => {
      const response = await fetch(`https://swapi.dev/api/people/${props.id}/`);
      const newData = await response.json();
```

```
    setData(newData);
   };

   fetchData();
  }, [props.id]);

  if (data) {
   return <div>{data.name}</div>;
  } else {
   return null;
  }
}
```

You might find fetching data in this way results in quite a bit of repeated code, especially if you're using fetch, and handle error and loading states. To avoid this, a common solution is to write a custom hook.

useContext Hook

React Context is a way to manage state globally.
It can be used together with the useState Hook to share state between deeply nested components more easily than with useState alone.

Create Context

To create context, you must Import createContext and initialize it:

```
import { useState, createContext } from "react";
import ReactDOM from "react-dom/client";

const UserContext = createContext()
```

Next we'll use the Context Provider to wrap the tree of components that need the state Context.

Context Provider

Wrap child components in the Context Provider and supply the state value.

```
function Component1() {
  const [user, setUser] = useState("Jesse Hall");

  return (
    <UserContext.Provider value={user}>
      <h1>{`Hello ${user}!`}</h1>
      <Component2 user={user} />
```

```
    </UserContext.Provider>
  );
}
```

Use the useContext Hook

In order to use the Context in a child component, we need to access it using the useContext Hook.

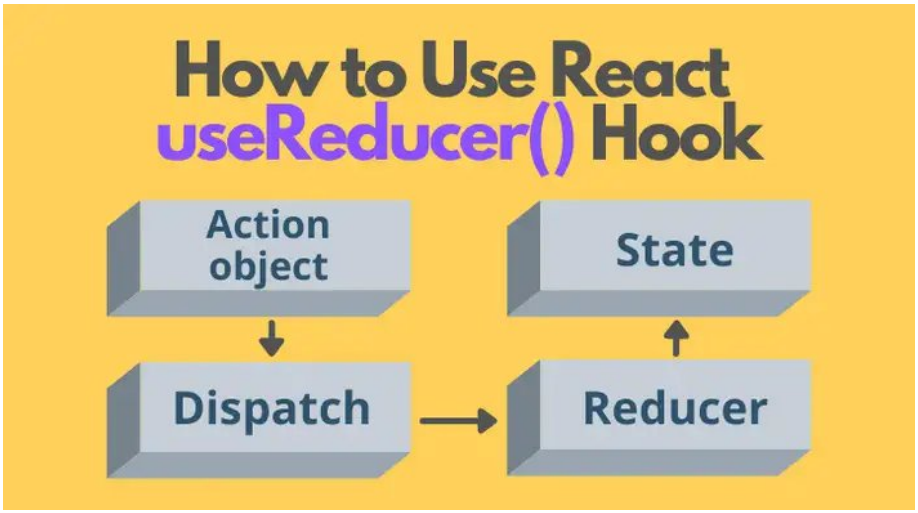
First, include the useContext in the import statement:

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

Then you can access the user Context in all components:

```
function Component5() {
  const user = useContext(UserContext);

  return (
    <>
      <h1>Component 5</h1>
      <h2>{`Hello ${user} again!`}</h2>
    </>
  );
}
```

useReducer Hook:
useReducer – simple state and action



If you've used the useState() hook to manage a non-trivial state like a list of items, where you need to add, update and remove items in the state, you can notice that the state management logic takes a good part of the component's body.

A React component should usually contain the logic that calculates the output. But the state management logic is a different concern that should be managed in a separate place. Otherwise, you get a mix of state management and rendering logic in one place, and that's difficult to read, maintain, and test!

To help you separate the concerns (rendering and state management) React provides the hook useReducer(). The hook does so by extracting the state management out of the component.

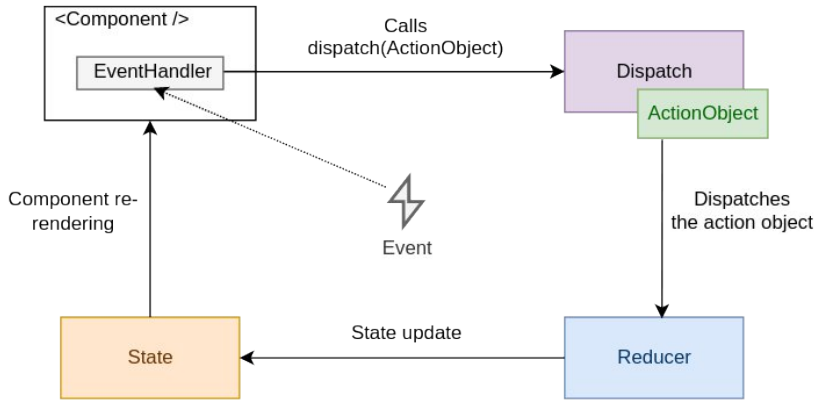
Let's see how the useReducer() hook works with accessible real-world examples.

```
import { useReducer } from 'react';
function MyComponent() {
  const [state, dispatch] = useReducer(reducer, initialState);
  const action = {
    type: 'ActionType'
  };
  return (
    <button onClick={() => dispatch(action)}>
      Click me
    </button>
  );
}
```

The following reducer function supports the increase and decrease of a counter state:

```
function reducer(state, action) {
  let newState;
  switch (action.type) {
    case 'increase':
      newState = { counter: state.counter + 1 };
      break;
    case 'decrease':
      newState = { counter: state.counter - 1 };
      break;
    default:
      throw new Error();
  }
  return newState;
}
```

complex state and action



The dispatch function with the action object is called as a result of an event handler or completing a fetch request.

Then React redirects the action object and the current state value to the reducer function.

The reducer function uses the action object and performs a state update, returning the new state.

React then checks whether the new state differs from the previous one. If the state has been updated, React re-renders the component, and useReducer() returns the new state value: [newState, ...] = useReducer(...).

multiple useReducers

We will use the same reducer function for multiple counters. We don't need to initialize the different states for the second counter. All useReducer hooks will behave as individual states.

```
import React, { useReducer } from "react";

const initialState = 0;

const reducer = (state, action) => {
  switch (action) {
    case 'increment':
      return state + 1;
    case 'decrement':
      return state - 1;
    case 'reset':
      return initialState;
    default:
      return state;
  }
}

function App() {
  const [count, dispatch] = useReducer(reducer, initialState);
  const [count2, dispatch2] = useReducer(reducer, initialState);

  return (
    <div classname="App">
      <h3>Multiple useReducer hook - <a href="https://www.cluemediator.com"
target="_blank" rel="noopener noreferrer">Clue Mediator</a></h3>
      <span>First Counter: {count}</span>
      <button onclick={() ==> dispatch('increment')}>Increment</button>
```

```
    <button onclick={() ==> dispatch('decrement')}>Decrement</button>
    <button onclick={() ==> dispatch('reset')}>Reset</button>
    <br /><br />
    <span>Second Counter: {count2}</span>
    <button onclick={() ==> dispatch2('increment')}>Increment</button>
    <button onclick={() ==> dispatch2('decrement')}>Decrement</button>
    <button onclick={() ==> dispatch2('reset')}>Reset</button>
   </div>
  );
}

export default App;

import React, { useReducer } from "react";

const initialState = 0;

const reducer = (state, action) => {
  switch (action) {
    case 'increment':
      return state + 1;
    case 'decrement':
      return state - 1;
    case 'reset':
      return initialState;
    default:
      return state;
  }
}

function App() {
  const [count, dispatch] = useReducer(reducer, initialState);
  const [count2, dispatch2] = useReducer(reducer, initialState);

  return (
    <div classname="App">
      <h3>Multiple useReducer hook - <a href="https://www.cluemediator.com"
target="_blank" rel="noopener noreferrer">Clue Mediator</a></h3>
      <span>First Counter: {count}</span>
      <button onclick={() ==> dispatch('increment')}>Increment</button>
      <button onclick={() ==> dispatch('decrement')}>Decrement</button>
      <button onclick={() ==> dispatch('reset')}>Reset</button>
      <br /><br />
      <span>Second Counter: {count2}</span>
```

```
    <button onclick={() ==> dispatch2('increment')}>Increment</button>
    <button onclick={() ==> dispatch2('decrement')}>Decrement</button>
    <button onclick={() ==> dispatch2('reset')}>Reset</button>
   </div>
  );
}

export default App;
```

useContext

useContext is a React Hook that lets you read and subscribe to context from your component.

```
useContext(SomeContext)
```

Call **useContext** at the top level of your component to read and subscribe to context.

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

function MyComponent() {
  const theme = useContext(ThemeContext);
  // ...
```

Parameters

SomeContext: The context that you've previously created with createContext. The context itself does not hold the information, it only represents the kind of information you can provide or read from components.

Returns

useContext returns the context value for the calling component. It is determined as the value passed to the closest SomeContext.Provider above the calling component in the tree. If there is no such provider, then the returned value will be the defaultValue you have passed to createContext for that context.
The returned value is always up-to-date. React automatically re-renders components that read some context if it changes.

Caveats

useContext() call in a component is not affected by providers returned from the same component. The corresponding <Context.Provider> needs to be above the component doing the useContext() call.

React automatically re-renders all the children that use a particular context starting from the provider that receives a different value. The previous and the next values are compared with the Object.is comparison. Skipping re-renders with memo does not prevent the children receiving fresh context values.

If your build system produces duplicates modules in the output (which can happen with symlinks), this can break context. Passing something via context only works if SomeContext that you use to provide context and SomeContext that you use to read it are exactly the same object, as determined by a === comparison.

Usage

Call useContext at the top level of your component to read and subscribe to context.

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

function Button() {
  const theme = useContext(ThemeContext);
  // ...
```

useContext returns the context value for the context you passed. To determine the context value, React searches the component tree and finds the closest context provider above for that particular context.

To pass context to a Button, wrap it or one of its parent components into the corresponding context provider:

```
function MyPage() {
  return (
    <ThemeContext.Provider value="dark">
      <Form />
    </ThemeContext.Provider>
  );
}

function Form() {
  // ... renders buttons inside ...
}
```

It doesn’t matter how many layers of components there are between the provider and the Button. When a Button anywhere inside of Form calls useContext(ThemeContext), it will receive "dark" as the value.

NOTE:
useContext() always looks for the closest provider above the component that calls it. It searches upwards and does not consider providers in the component from which you’re calling useContext().

Updating data passed via context

Often, you'll want the context to change over time. To update context, combine it with state. Declare a state variable in the parent component, and pass the current state down as the context value to the provider.

```
function MyPage() {
  const [theme, setTheme] = useState('dark');
  return (
    <ThemeContext.Provider value={theme}>
      <Form />
      <Button onClick={() => {
        setTheme('light');
      }}>
        Switch to light theme
      </Button>
    </ThemeContext.Provider>
  );
}
```

useReducer

useReducer is a React Hook that lets you add a reducer to your component.

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

Call useReducer at the top level of your component to manage its state with a reducer.

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

function reducer(state, action) {
  // ...
}

function MyComponent() {
  const [state, dispatch] = useReducer(reducer, { age: 42 });
  // ...
```

Parameters

- **reducer**: The reducer function that specifies how the state gets updated. It must be pure, should take the state and action as arguments, and should return the next state. State and action can be of any type.

- **initialArg**: The value from which the initial state is calculated. It can be a value of any type. How the initial state is calculated from it depends on the next init argument.

- **optional init**: The initializer function that should return the initial state. If it's not specified, the initial state is set to initialArg. Otherwise, the initial state is set to the result of calling init(initialArg).

Returns

useReducer returns an array with exactly two values:

The current state. During the first render, it's set to init(initialArg) or initialArg (if there's no init). The dispatch function that lets you update the state to a different value and trigger a re-render.

Caveats

useReducer is a Hook, so you can only call it at the top level of your component or your own Hooks. You can't call it inside loops or conditions. If you need that, extract a new component and move the state into it.

In Strict Mode, React will call your reducer and initializer twice in order to help you find accidental impurities. This is development-only behavior and does not affect production. If your reducer and initializer are pure (as they should be), this should not affect your logic.

The result from one of the calls is ignored.

Usage

Call useReducer at the top level of your component to manage state with a reducer.

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

function reducer(state, action) {
  // ...
}

function MyComponent() {
  const [state, dispatch] = useReducer(reducer, { age: 42 });
  // ...
```

useReducer returns an array with exactly two items:

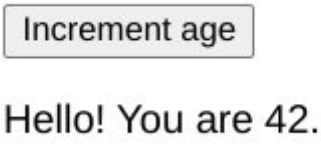
The current state of this state variable, initially set to the initial state you provided.
The dispatch function that lets you change it in response to interaction.

To update what's on the screen, call dispatch with an object representing what the user did, called an action:

```
function handleClick() {
  dispatch({ type: 'incremented_age' });
}
```

React will pass the current state and the action to your reducer function. Your reducer will calculate and return the next state. React will store that next state, render your component with it, and update the UI.

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

function reducer(state, action) {
  if (action.type === 'incremented_age') {
    return {
      age: state.age + 1
    };
  }
  throw Error('Unknown action.');
}

export default function Counter() {
  const [state, dispatch] = useReducer(reducer, { age: 42 });

  return (
    <>
      <button onClick={() => {
        dispatch({ type: 'incremented_age' })
      }}>
        Increment age
      </button>
      <p>Hello! You are {state.age}.</p>
    </>
  );
}
```



Fetching data with useReduer

Using the useReducer() hook, we are going to fetch data from the API. We have performed data fetching in the previous article from **'https://reqres.in'** API using Axios library; here, we will be using useReducer() hook for fetching data.

Let's look at the example first with the use of useEffect() hook with the same example we used for useReducer() hook,

First, to start with, we need to install the Axios library using a command mentioned below.

//GetData_Reduce.js

```
import React, { useEffect, useReducer } from 'react'
import axios from 'axios'

const initialState = {
   user: {},
   loading: true,
   error: ''
}

const reduce = (state, action) => {
   switch (action.type) {
      case 'OnSuccess':
         return {
            loading: false,
            user: action.payload,
            error: ''
         }
      case 'OnFailure':
         return {
            loading: false,
            user: {},
            error: 'Something went wrong'
         }

      default:
         return state
   }
}

function GetData_Reduce() {
   const [state, dispatch] = useReducer(reduce, initialState)
```

```
  useEffect(() => {
    axios.get('https://reqres.in/api/users/2')
      .then(response => {
        dispatch({ type: 'OnSuccess', payload: response.data.data })
      })
      .catch(error => {
        dispatch({ type: 'OnFailure' })
      })
  }, [])

  return (
    <div>
      {state.loading ? 'Loading!! Please wait' : state.user.email}
      {state.error ? state.error : null}
    </div>
  )
}

export default GetData_Reduce
```

//App.js

```
import React from 'react';
import './App.css';
import GetData_Reduce from './components/GetData_Reduce;

function App() {

  return (
    <div className="App">
      <GetData_Reduce/>
    </div>
  );
}

export default App;
```

First, it will display the loading screen.
Once the data is loaded from the API, the data will be displayed on the screen.

useState vs useReduce

What is the difference between useState() and useReducer() hook?

Scenario	useState()	useReducer()
Type of state	Use it when working with Number, Boolean, and String	Use it when working with object or Array
Number of state Transition	useState hook has only one or two transitions so it should be used when you have only 1 or 2 setState calls	useReducer() hook should be used when many transitions so it must be used when we have many setState that need to be called
Related State Transition	No related transition	It includes state transition
Business Logic	useState has no business logic	When your application involves complex data transformation or manipulation that this should be used
Local vs Global	When dealing with a single component and need to perform operations locally than useState should be used	When you want to deal with multiple components for passing data from one component to another then useReducer() is a better approach.