React built-in hooks, custom hooks, built-in components
legacy, client and server apis

1. useCallback
2. useMemo
3. useContext
4. useDebugValue
5. useDeferredValue
6. useEffect
7. useId
8. useImperativeHandle
9. useInsertionEffect
10. useLayoutEffect
11. useReducer
12. useRef
13. useState
14. useSyncExternalStore
15. useTransition
16. Custom Hooks: useWindowResize
17. <Fragment>
18. <Profiler>
19. <StrictMode>
20. <Suspense>
21. renderToNodeStream
22. renderToPipeableStream
23. renderToReadableStream
24. renderToStaticMarkup
25. renderToStaticNodeStream
26. renderToString
27. isValidElement
28. PureComponent
29. createContext
30. forwardRef
31. lazy
32. memo
33. startTransition
34. createPortal
35. flushSync
36. findDOMNode
37. hydrate
38. render
39. unmountComponentAtNode
40. createRoot
41. hydrateRoot
42. Children
43. cloneElement
44. Component
45. createElement
46. createFactory
47. createRef
48. Throttle & Debounce
49. Cohesion, Hoisting & Closure
50. Coupling, Error Boundaries

React built-in hooks

1. useCallback:

It is a React Hook that lets us to cache a function definition between re-renders.

```
const cachedFn = useCallback(fn, dependencies)
```

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

export default function ProductPage({ productId, referrer, theme }) {
  const handleSubmit = useCallback((orderDetails) => {
    post('/product/' + productId + '/buy', {
      referrer,
      orderDetails,
    });
  }, [productId, referrer]);
```

Parameters

- fn: The function value that you want to cache. It can take any arguments and return any values. React will return (not call!) your function back to you during the initial render. On next renders, React will give you the same function again if the dependencies have not changed since the last render. Otherwise, it will give you the function that you have passed during the current render, and store it in case it can be reused later. React will not call your function. The function is returned to you so you can decide when and whether to call it.

- dependencies: The list of all reactive values referenced inside of the fn code. Reactive values include props, state, and all the variables and functions declared directly inside your component body. If your linter is configured for React, it will verify that every reactive value is correctly specified as a dependency. The list of dependencies must have a constant number of items and be written inline like [dep1, dep2, dep3]. React will compare each dependency with its previous value using the Object.is comparison algorithm.

Returns

On the initial render, useCallback returns the fn function you have passed.

During subsequent renders, it will either return an already stored fn function from the last render (if the dependencies haven't changed), or return the fn function you have passed during this render.

Caveats

- useCallback 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.
- React will not throw away the cached function unless there is a specific reason to do that. For example, in development, React throws away the cache when you edit the file of your component.

Both in development and in production, React will throw away the cache if your component suspends during the initial mount. In the future, React may add more features that take advantage of throwing away the cache—for example, if React adds built-in support for virtualized lists in the future, it would make sense to throw away the cache for items that scroll out of the virtualized table viewport. This should match your expectations if you rely on useCallback as a performance optimization. Otherwise, a state variable or a ref may be more appropriate.

Usage :

- Skipping re-rendering of components

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

function ProductPage({ productId, referrer, theme }) {
  const handleSubmit = useCallback((orderDetails) => {
    post('/product/' + productId + '/buy', {
      referrer,
      orderDetails,
    });
  }, [productId, referrer]);
```

You need to pass two things to useCallback:

- A function definition that you want to cache between re-renders.
- A list of dependencies including every value within your component that’s used inside your function.

On the initial render, the returned function you'll get from useCallback will be the function you passed.

On the following renders, React will compare the dependencies with the dependencies you passed during the previous render. If none of the dependencies have changed (compared with Object.is), useCallback will return the same function as before. Otherwise, useCallback will return the function you passed on this render.

In other words, useCallback caches a function between re-renders until its dependencies change.

Let's walk through an example to see when this is useful.
Say you're passing a handleSubmit function down from the ProductPage to the ShippingForm component:

```
function ProductPage({ productId, referrer, theme }) {
  // ...
  return (
    <div className={theme}>
      <ShippingForm onSubmit={handleSubmit} />
    </div>
  );
```

You've noticed that toggling the theme prop freezes the app for a moment, but if you remove <ShippingForm /> from your JSX, it feels fast. This tells you that it's worth trying to optimize the ShippingForm component.

By default, when a component re-renders, React re-renders all of its children recursively. This is why, when ProductPage re-renders with a different theme, the ShippingForm component also re-renders. This is fine for components that don't require much calculation to re-render. But if you verified a re-render is slow, you can tell ShippingForm to skip re-rendering when its props are the same as on last render by wrapping it in memo:

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

const ShippingForm = memo(function ShippingForm({ onSubmit }) {
  // ...
});
```

With this change, ShippingForm will skip re-rendering if all of its props are the same as on the last render. This is when caching a function becomes important! Let's say you defined handleSubmit without useCallback:

```
function ProductPage({ productId, referrer, theme }) {
  // Every time the theme changes, this will be a different function...
  function handleSubmit(orderDetails) {
    post('/product/' + productId + '/buy', {
      referrer,
      orderDetails,
    });
  }
```

```
  return (
    <div className={theme}>
      {/* ... so ShippingForm's props will never be the same, and it will re-render every time */}
      <ShippingForm onSubmit={handleSubmit} />
    </div>
  );
}
```

In JavaScript, a function () {} or () => {} always creates a different function, similar to how the {} object literal always creates a new object. Normally, this wouldn't be a problem, but it means that ShippingForm props will never be the same, and your memo optimization won't work. This is where useCallback comes in handy:

```
function ProductPage({ productId, referrer, theme }) {
  // Tell React to cache your function between re-renders...
  const handleSubmit = useCallback((orderDetails) => {
    post('/product/' + productId + '/buy', {
      referrer,
      orderDetails,
    });
  }, [productId, referrer]); // ...so as long as these dependencies don't change...

  return (
    <div className={theme}>
      {/* ...ShippingForm will receive the same props and can skip re-rendering */}
      <ShippingForm onSubmit={handleSubmit} />
    </div>
  );
}
```

By wrapping handleSubmit in useCallback, you ensure that it's the same function between the re-renders (until dependencies change). You don't have to wrap a function in useCallback unless you do it for some specific reason. In this example, the reason is that you pass it to a component wrapped in memo, and this lets it skip re-rendering. There are other reasons you might need useCallback which are described further on this page.

You should only rely on useCallback as a performance optimization. If your code doesn't work without it, find the underlying problem and fix it first. Then you may add useCallback back.

https://react.dev/reference/react/useCallback#examples-rerendering

- Updating state from a memoized callback

Sometimes, you might need to update state based on previous state from a memoized callback.

This handleAddTodo function specifies todos as a dependency because it computes the next todos from it:

```
function TodoList() {
  const [todos, setTodos] = useState([]);

  const handleAddTodo = useCallback((text) => {
    const newTodo = { id: nextId++, text };
    setTodos([...todos, newTodo]);
  }, [todos]);
  // ...
```

You'll usually want memoized functions to have as few dependencies as possible. When you read some state only to calculate the next state, you can remove that dependency by passing an updater function instead:

```
function TodoList() {
  const [todos, setTodos] = useState([]);

  const handleAddTodo = useCallback((text) => {
    const newTodo = { id: nextId++, text };
    setTodos(todos => [...todos, newTodo]);
  }, []); // ✅ No need for the todos dependency
  // ...
```

Here, instead of making todos a dependency and reading it inside, you pass an instruction about how to update the state (todos => [...todos, newTodo]) to React

- Preventing an Effect from firing too often

Sometimes, you might want to call a function from inside an Effect:

```
function ChatRoom({ roomId }) {
  const [message, setMessage] = useState('');

  function createOptions() {
    return {
      serverUrl: 'https://localhost:1234',
      roomId: roomId
    };
  }

  useEffect(() => {
    const options = createOptions();
    const connection = createConnection();
    connection.connect();
    // ...
```

This creates a problem. Every reactive value must be declared as a dependency of your Effect. However, if you declare createOptions as a dependency, it will cause your Effect to constantly reconnect to the chat room:

```
  useEffect(() => {
    const options = createOptions();
    const connection = createConnection();
    connection.connect();
    return () => connection.disconnect();
  }, [createOptions]); // 🔴 Problem: This dependency changes on every render
  // ...
```

To solve this, you can wrap the function you need to call from an Effect into useCallback:

```
function ChatRoom({ roomId }) {
  const [message, setMessage] = useState('');

  const createOptions = useCallback(() => {
    return {
      serverUrl: 'https://localhost:1234',
      roomId: roomId
    };
  }, [roomId]); // ✅ Only changes when roomId changes

  useEffect(() => {
    const options = createOptions();
    const connection = createConnection();
    connection.connect();
    return () => connection.disconnect();
  }, [createOptions]); // ✅ Only changes when createOptions changes
  // ...
```

This ensures that the createOptions function is the same between re-renders if the roomId is the same. However, it's even better to remove the need for a function dependency. Move your function inside the Effect:

```
function ChatRoom({ roomId }) {
  const [message, setMessage] = useState('');

  useEffect(() => {
    function createOptions() { // ✅ No need for useCallback or function dependencies!
      return {
        serverUrl: 'https://localhost:1234',
        roomId: roomId
      };
    }

    const options = createOptions();
    const connection = createConnection();
```

- Optimizing a custom Hook

```
function useRouter() {
  const { dispatch } = useContext(RouterStateContext);

  const navigate = useCallback((url) => {
    dispatch({ type: 'navigate', url });
  }, [dispatch]);

  const goBack = useCallback(() => {
    dispatch({ type: 'back' });
  }, [dispatch]);

  return {
    navigate,
    goBack,
  };
}
```

This ensures that the consumers of your Hook can optimize their own code when needed.

Troubleshooting

Every time my component renders, useCallback returns a different function

```
function ProductPage({ productId, referrer }) {
  const handleSubmit = useCallback((orderDetails) => {
    post('/product/' + productId + '/buy', {
      referrer,
      orderDetails,
    });
  }); // 🔴 Returns a new function every time: no dependency array
  // ...
```

```
function ProductPage({ productId, referrer }) {
  const handleSubmit = useCallback((orderDetails) => {
    post('/product/' + productId + '/buy', {
      referrer,
      orderDetails,
    });
  }, [productId, referrer]); // ✅ Does not return a new function unnecessarily
  // ...
```

f this doesn't help, then the problem is that at least one of your dependencies is different from the previous ender. You can debug this problem by manually logging your dependencies to the console:

```
  const handleSubmit = useCallback((orderDetails) => {
    // ..
  }, [productId, referrer]);

  console.log([productId, referrer]);
```

You can then right-click on the arrays from different re-renders in the console and select "Store as a global variable" for both of them. Assuming the first one got saved as `temp1` and the second one got saved as `temp2`, you can then use the browser console to check whether each dependency in both arrays is the same:

```
Object.is(temp1[0], temp2[0]); // Is the first dependency the same between the arrays?
Object.is(temp1[1], temp2[1]); // Is the second dependency the same between the arrays?
Object.is(temp1[2], temp2[2]); // ... and so on for every dependency ...
```

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

Suppose the Chart component is wrapped in memo. You want to skip re-rendering every Chart in the list when the ReportList component re-renders. However, you can't call useCallback in a loop:

```
function ReportList({ items }) {
  return (
    <article>
      {items.map(item => {
        // 🔴 You can't call useCallback in a loop like this:
        const handleClick = useCallback(() => {
          sendReport(item)
        }, [item]);

        return (
          <figure key={item.id}>
            <Chart onClick={handleClick} />
          </figure>
        );
      })}
    </article>
  );
}
```

```
function ReportList({ items }) {
  // ...
}

const Report = memo(function Report({ item }) {
  function handleClick() {
    sendReport(item);
  }

  return (
    <figure>
      <Chart onClick={handleClick} />
    </figure>
  );
});
```

```
function ReportList({ items }) {
  return (
    <article>
      {items.map(item =>
        <Report key={item.id} item={item} />
      )}
    </article>
  );
}

function Report({ item }) {
  // ✅ Call useCallback at the top level:
  const handleClick = useCallback(() => {
    sendReport(item)
  }, [item]);

  return (
    <figure>
      <Chart onClick={handleClick} />
    </figure>
  );
}
```

2. useMemo

useMemo is a React Hook that lets us to cache the result of a calculation between re-renders.

```
const cachedValue = useMemo(calculateValue, dependencies)
```

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

function TodoList({ todos, tab }) {
  const visibleTodos = useMemo(
    () => filterTodos(todos, tab),
    [todos, tab]
  );
  // ...
}
```

Parameters:

- calculateValue: The function calculating the value that you want to cache. It should be pure, should take no arguments, and should return a value of any type. React will call your function during the initial render. On next renders, React will return the same value again if the dependencies have not changed since the last render. Otherwise, it will call calculateValue, return its result, and store it so it can be reused later.
- dependencies: The list of all reactive values referenced inside of the calculateValue code. Reactive values include props, state, and all the variables and functions declared directly inside your component body. If your linter is configured for React, it will verify that every reactive value is correctly specified as a dependency.

The list of dependencies must have a constant number of items and be written inline like [dep1, dep2, dep3]. React will compare each dependency with its previous value using the Object.is comparison.

Returns:

On the initial render, useMemo returns the result of calling calculateValue with no arguments.

During next renders, it will either return an already stored value from the last render (if the dependencies haven't changed), or call calculateValue again, and return the result that calculateValue has returned.

Caveats:

- useMemo 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 calculation function twice in order to help you find accidental impurities. This is development-only behavior and does not affect production. If your calculation function is pure (as it should be), this should not affect your logic. The result from one of the calls will be ignored.
- React will not throw away the cached value unless there is a specific reason to do that. For example, in development, React throws away the cache when you edit the file of your component.

Both in development and in production, React will throw away the cache if your component suspends during the initial mount. In the future, React may add more features that take advantage of throwing away the cache—for example, if React adds built-in support for virtualized lists in the future, it would make sense to throw away the cache for items that scroll out of the virtualized table viewport. This should be fine if you rely on useMemo solely as a performance optimization. Otherwise, a state variable or a ref may be more appropriate.

Caching return values like this is also known as memoization, which is why this Hook is called useMemo.

Usage

- Skipping expensive recalculations

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

function TodoList({ todos, tab, theme }) {
  const visibleTodos = useMemo(() => filterTodos(todos, tab), [todos, tab]);
  // ...
}
```

You need to pass two things to useMemo:

- A calculation function that takes no arguments, like () =>, and returns what you wanted to calculate.
- A list of dependencies including every value within your component that's used inside your calculation.

By default, React will re-run the entire body of your component every time that it re-renders. For example, if this TodoList updates its state or receives new props from its parent, the filterTodos function will re-run:

```
function TodoList({ todos, tab, theme }) {
  const visibleTodos = filterTodos(todos, tab);
  // ...
}
```

Usually, this isn't a problem because most calculations are very fast. However, if you're filtering or transforming a large array, or doing some expensive computation, you might want to skip doing it again if data hasn't changed. If both todos and tab are the same as they were during the last render, wrapping the calculation in useMemo like earlier lets you reuse visibleTodos you've already calculated before.

This type of caching is called memoization.

You should only rely on useMemo as a performance optimization. If your code doesn't work without it, find the underlying problem and fix it first. Then you may add useMemo to improve performance.

If your app is like this site, and most interactions are coarse (like replacing a page or an entire section), memoization is usually unnecessary. On the other hand, if your app is more like a drawing editor, and most interactions are granular (like moving shapes), then you might find memoization very helpful.

Optimizing with useMemo is only valuable in a few cases:

- The calculation you're putting in useMemo is noticeably slow, and its dependencies rarely change.
- You pass it as a prop to a component wrapped in memo. You want to skip re-rendering if the value hasn't changed. Memoization lets your component re-render only when dependencies aren't the same.
- The value you're passing is later used as a dependency of some Hook. For example, maybe another useMemo calculation value depends on it. Or maybe you are depending on this value from useEffect.

https://react.dev/reference/react/useMemo#examples-recalculation

- Skipping re-rendering of components

```
export default function TodoList({ todos, tab, theme }) {
  // ...
  return (
    <div className={theme}>
      <List items={visibleTodos} />
    </div>
  );
}
```

You've noticed that toggling the theme prop freezes the app for a moment, but if you remove <List /> from your JSX, it feels fast. This tells you that it's worth trying to optimize the List component.

By default, when a component re-renders, React re-renders all of its children recursively. This is why, when TodoList re-renders with a different theme, the List component also re-renders. This is fine for components that don't require much calculation to re-render.

But if you've verified that a re-render is slow, you can tell List to skip re-rendering when its props are the same as on last render by wrapping it in memo:

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

const List = memo(function List({ items }) {
  // ...
});
```

With this change, List will skip re-rendering if all of its props are the same as on the last render. This is where caching the calculation becomes important! Imagine that you calculated visibleTodos without useMemo:

```
export default function TodoList({ todos, tab, theme }) {
  // Every time the theme changes, this will be a different array...
  const visibleTodos = filterTodos(todos, tab);
  return (
    <div className={theme}>
      {/* ... so List's props will never be the same, and it will re-render every time */}
      <List items={visibleTodos} />
    </div>
  );
}
```

```
export default function TodoList({ todos, tab, theme }) {
  // Tell React to cache your calculation between re-renders...
  const visibleTodos = useMemo(
    () => filterTodos(todos, tab),
    [todos, tab] // ...so as long as these dependencies don't change...
  );
  return (
    <div className={theme}>
```

- Memoizing a dependency of another Hook

```
function Dropdown({ allItems, text }) {
  const searchOptions = { matchMode: 'whole-word', text };

  const visibleItems = useMemo(() => {
    return searchItems(allItems, searchOptions);
  }, [allItems, searchOptions]); // 🚩 Caution: Dependency on an object created in the component body
  // ...
```

```
function Dropdown({ allItems, text }) {
  const searchOptions = useMemo(() => {
    return { matchMode: 'whole-word', text };
  }, [text]); // ✅ Only changes when text changes

  const visibleItems = useMemo(() => {
    return searchItems(allItems, searchOptions);
  }, [allItems, searchOptions]); // ✅ Only changes when allItems or searchOptions changes
  // ...
```

Memoizing a function:

```
export default function ProductPage({ productId, referrer }) {
  function handleSubmit(orderDetails) {
    post('/product/' + productId + '/buy', {
      referrer,
      orderDetails
    });
  }

  return <Form onSubmit={handleSubmit} />;
}
```

Just as {} creates a different object, function declarations like function() {} and expressions like () => {} produce a different function on every re-render. By itself, creating a new function is not a problem.

This is not something to avoid! However, if the Form component is memoized, presumably you want to skip re-rendering it when no props have changed. A prop that is always different would defeat the point of memoization.

To memoize a function with useMemo, your calculation function would have to return another function:

```
export default function Page({ productId, referrer }) {
  const handleSubmit = useMemo(() => {
    return (orderDetails) => {
      post('/product/' + productId + '/buy', {
        referrer,
        orderDetails
      });
    };
  }, [productId, referrer]);

  return <Form onSubmit={handleSubmit} />;
}
```

This looks clunky! Memoizing functions is common enough that React has a built-in Hook specifically for that. Wrap your functions into useCallback instead of useMemo to avoid having to write an extra nested function:

```
export default function Page({ productId, referrer }) {
  const handleSubmit = useCallback((orderDetails) => {
    post('/product/' + productId + '/buy', {
      referrer,
      orderDetails
    });
  }, [productId, referrer]);

  return <Form onSubmit={handleSubmit} />;
}
```

Troubleshooting:

My calculation runs twice on every re-render

In Strict Mode, React will call some of your functions twice instead of once:

```
function TodoList({ todos, tab }) {
  // This component function will run twice for every render.

  const visibleTodos = useMemo(() => {
    // This calculation will run twice if any of the dependencies change.
    return filterTodos(todos, tab);
  }, [todos, tab]);

  // ...
```

This is expected and shouldn't break your code.

This development-only behavior helps you keep components pure. React uses the result of one of the calls, and ignores the result of the other call. As long as your component and calculation functions are pure, this shouldn't affect your logic. However, if they are accidentally impure, this helps you notice and fix the mistake.

```
  const visibleTodos = useMemo(() => {
    // ▶ Mistake: mutating a prop
    todos.push({ id: 'last', text: 'Go for a walk!' });
    const filtered = filterTodos(todos, tab);
    return filtered;
  }, [todos, tab]);
```

React calls your function twice, so you'd notice the todo is added twice. Your calculation shouldn't change any existing objects, but it's okay to change any new objects you created during the calculation. For example, if the filterTodos function always returns a different array, you can mutate that array instead:

```
const visibleTodos = useMemo(() => {
  const filtered = filterTodos(todos, tab);
  // ✅ Correct: mutating an object you created during the calculation
  filtered.push({ id: 'last', text: 'Go for a walk!' });
  return filtered;
}, [todos, tab]);
```

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

```
// 🔴 You can't return an object from an arrow function with () => {
const searchOptions = useMemo(() => {
  matchMode: 'whole-word',
  text: text
}, [text]);

// This works, but is easy for someone to break again
const searchOptions = useMemo(() => ({
  matchMode: 'whole-word',
  text: text
}), [text]);

// ✅ This works and is explicit
const searchOptions = useMemo(() => {
  return {
    matchMode: 'whole-word',
    text: text
  };
}, [text]);
```

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

```
function TodoList({ todos, tab }) {
  // 🔴 Recalculates every time: no dependency array
  const visibleTodos = useMemo(() => filterTodos(todos, tab));
  // ...
```

```
function TodoList({ todos, tab }) {
  // ✅ Does not recalculate unnecessarily
  const visibleTodos = useMemo(() => filterTodos(todos, tab), [todos, tab]);
  // ...
```

```
  const visibleTodos = useMemo(() => filterTodos(todos, tab), [todos, tab]);
  console.log([todos, tab]);
```

You can then right-click on the arrays from different re-renders in the console and select “Store as a global variable” for both of them. Assuming the first one got saved as `temp1` and the second one got saved as `temp2`, you can then use the browser console to check whether each dependency in both arrays is the same:

```
Object.is(temp1[0], temp2[0]); // Is the first dependency the same between the arrays?
Object.is(temp1[1], temp2[1]); // Is the second dependency the same between the arrays?
Object.is(temp1[2], temp2[2]); // ... and so on for every dependency ...
```

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

```
function ReportList({ items }) {
  return (
    <article>
      {items.map(item => {
        // 🔴 You can't call useMemo in a loop like this:
        const data = useMemo(() => calculateReport(item), [item]);
        return (
          <figure key={item.id}>
            <Chart data={data} />
          </figure>
```

```
function ReportList({ items }) {
  return (
    <article>
      {items.map(item =>
        <Report key={item.id} item={item} />
      )}
    </article>
  );
}

function Report({ item }) {
  // ✅ Call useMemo at the top level:
  const data = useMemo(() => calculateReport(item), [item]);
  return (
    <figure>
      <Chart data={data} />
    </figure>
  );
}
```

Alternatively, you could remove useMemo and instead wrap Report itself in memo. If the item prop does not change, Report will skip re-rendering, so Chart will skip re-rendering too:

```
function ReportList({ items }) {
  // ...
}

const Report = memo(function Report({ item }) {
  const data = calculateReport(item);
  return (
    <figure>
      <Chart data={data} />
    </figure>
  );
});
```

```jsx
import { useState, useMemo } from 'react'

export function Parent() {
  const [count, setCount] = useState(0)
  const onClick = () => setCount(c => c + 1)

  return (
    <>
      <button onClick={onClick}>Increase {count}</button>
      {useMemo(() => {
        // Doesn't re-render even if `count` state changes
        return <Child />
      }, [])}
    </>
  )
}
```

3. useContext

It is a React Hook that lets us to read and subscribe to context from the component.

Context lets the parent component make some information available to any component in the tree below it—no matter how deep—without passing it explicitly through props.

Passing props is a great way to explicitly pipe data through your UI tree to the components that use it.

But passing props can become verbose and inconvenient when you need to pass some prop deeply through the tree, or if many components need the same prop. The nearest common ancestor could be far removed from the components that need data, and lifting state up that high can lead to a situation called “prop drilling”.

Context: an alternative to passing props

Context lets a parent component provide data to the entire tree below it using createContext and useContext hooks of react and the corresponding provider.

https://react.dev/learn/passing-data-deeply-with-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 :

- Passing data deeply into the tree

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

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

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

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

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:

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

Specifying a fallback default value:

```
const ThemeContext = createContext('light');
```

Overriding context for a part of the tree:

```
<ThemeContext.Provider value="dark">
  ...
  <ThemeContext.Provider value="light">
    <Footer />
  </ThemeContext.Provider>
  ...
</ThemeContext.Provider>
```

- Optimizing re-renders when passing objects and functions

```
function MyApp() {
  const [currentUser, setCurrentUser] = useState(null);

  function login(response) {
    storeCredentials(response.credentials);
    setCurrentUser(response.user);
  }

  return (
    <AuthContext.Provider value={{ currentUser, login }}>
      <Page />
    </AuthContext.Provider>
  );
}
```

```
import { useCallback, useMemo } from 'react';

function MyApp() {
  const [currentUser, setCurrentUser] = useState(null);

  const login = useCallback((response) => {
    storeCredentials(response.credentials);
    setCurrentUser(response.user);
  }, []);

  const contextValue = useMemo(() => ({
    currentUser,
    login
  }), [currentUser, login]);

  return (
    <AuthContext.Provider value={contextValue}>
      <Page />
    </AuthContext.Provider>
  );
```

Troubleshooting:

- My component doesn’t see the value from my provider

There are a few common ways that this can happen:

- You’re rendering <SomeContext.Provider> in the same component (or below) as where you’re calling useContext(). Move <SomeContext.Provider> above and outside the component calling useContext().
- You may have forgotten to wrap your component with <SomeContext.Provider>, or you might have put it in a different part of the tree than you thought. Check whether the hierarchy is right using React DevTools.
- You might be running into some build issue with your tooling that causes SomeContext as seen from the providing component and SomeContext as seen by the reading component to be two different objects. This can happen if you use symlinks, for example. You can verify this by assigning them to globals like window.SomeContext1 and window.SomeContext2 and then checking whether window.SomeContext1 === window.SomeContext2 in the console. If they’re not the same, fix that issue on the build tool level.

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

You might have a provider without a `value` in the tree:

```
// 🚩 Doesn't work: no value prop
<ThemeContext.Provider>
   <Button />
</ThemeContext.Provider>
```

If you forget to specify `value`, it's like passing `value={undefined}`.

You may have also mistakingly used a different prop name by mistake:

```
// 🚩 Doesn't work: prop should be called "value"
<ThemeContext.Provider theme={theme}>
   <Button />
</ThemeContext.Provider>
```

In both of these cases you should see a warning from React in the console. To fix them, call the pr

```
// ✅ Passing the value prop
<ThemeContext.Provider value={theme}>
   <Button />
</ThemeContext.Provider>
```

4. useDebugValue

useDebugValue is a React Hook that lets us to add a label to a custom Hook in React DevTools.

```
useDebugValue(value, format?)
```

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

function useOnlineStatus() {
  // ...
  useDebugValue(isOnline ? 'Online' : 'Offline');
  // ...
}
```

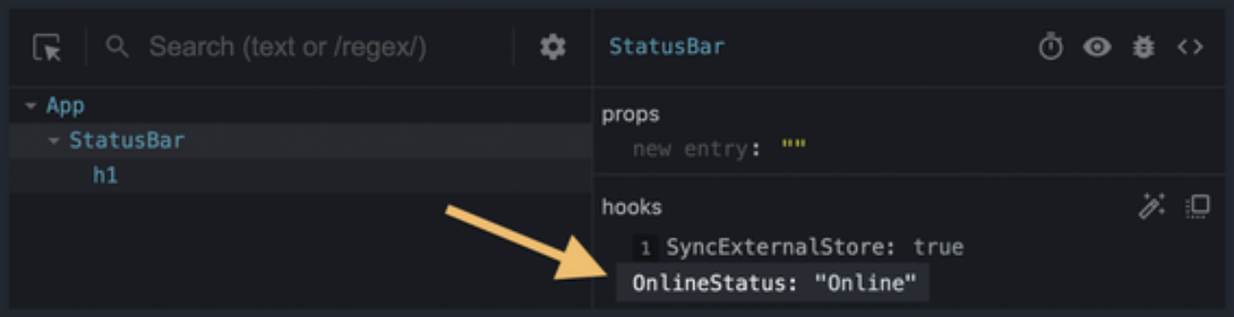
Parameters:

- value: The value you want to display in React DevTools. It can have any type.
- optional format: A formatting function. When the component is inspected, React DevTools will call the formatting function with the value as the argument, and then display the returned formatted value (which may have any type). If you don't specify the formatting function, the original value itself will be displayed.

Returns

useDebugValue does not return anything.

This gives components calling `useOnlineStatus` a label like `OnlineStatus: "Online"` when you inspect them:



Without the `useDebugValue` call, only the underlying data (in this example, `true`) would be displayed.

App.js useOnlineStatus.js Reset Fork

```
1  import { useSyncExternalStore, useDebugValue } from 'react'
2
3  export function useOnlineStatus() {
4    const isOnline = useSyncExternalStore(subscribe,
5    () => navigator.onLine, () => true);
6    useDebugValue(isOnline ? 'Online' : 'Offline');
7    return isOnline;
8  }
9
10 function subscribe(callback) {
11   window.addEventListener('online', callback);
12   window.addEventListener('offline', callback);
13   return () => {
14     window.removeEventListener('online', callback);
15     window.removeEventListener('offline', callback);
16   };
```

✅ Online

Show more

Deferring formatting of a debug value:

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

This lets you avoid running potentially expensive formatting logic unless the component is actually inspected. For example, if date is a Date value, this avoids calling toDateString() on it for every render.

5. useDeferredValue

useDeferredValue is a React Hook that lets user to defer updating a part of the UI.

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

function SearchPage() {
  const [query, setQuery] = useState('');
  const deferredQuery = useDeferredValue(query);
  // ...
}
```

Parameters

value: The value you want to defer. It can have any type.
Returns

During the initial render, the returned deferred value will be the same as the value you provided. During updates, React will first attempt a re-render with the old value (so it will return the old value), and then try another re-render in background with the new value (so it will return the updated value).

Caveats

- The values you pass to useDeferredValue should either be primitive values (like strings and numbers) or objects created outside of rendering. If you create a new object during rendering and immediately pass it to useDeferredValue, it will be different on every render, causing unnecessary background re-renders.

When useDeferredValue receives a different value (compared with Object.is), in addition to the current render (when it still uses the previous value), it schedules a re-render in the background with the new value. The background re-render is interruptible: if there's another update to the value, React will restart the background re-render from scratch. For example, if the user is typing into an input faster than a chart receiving its deferred value can re-render, the chart will only re-render after the user stops typing.

useDeferredValue is integrated with <Suspense>. If the background update caused by a new value suspends the UI, the user will not see the fallback. They will see the old deferred value until the data loads.

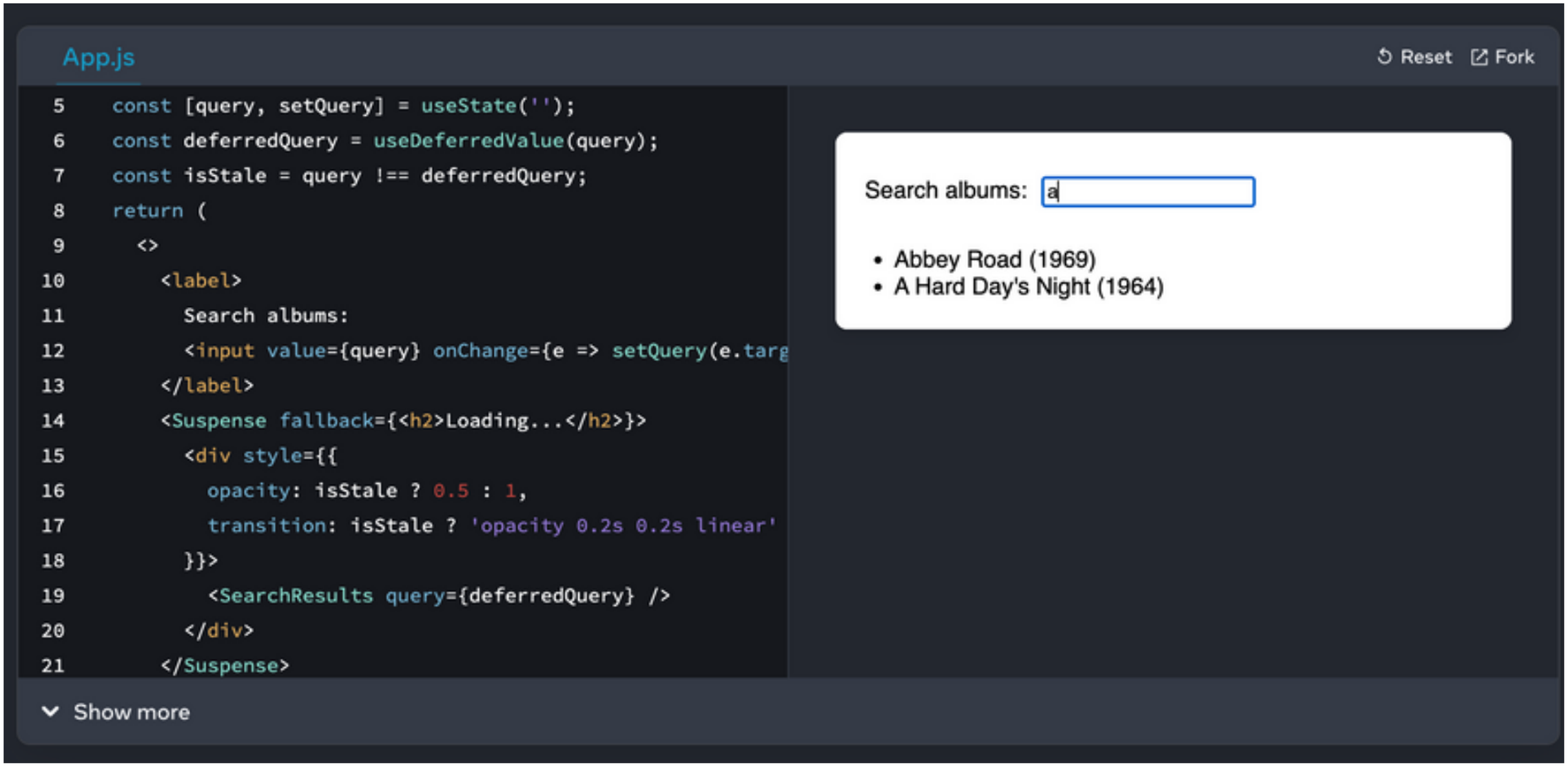
useDeferredValue does not by itself prevent extra network requests.

There is no fixed delay caused by useDeferredValue itself. As soon as React finishes the original re-render, React will immediately start working on the background re-render with the new deferred value. Any updates caused by events (like typing) will interrupt the background re-render and get prioritized over it.

The background re-render caused by useDeferredValue does not fire Effects until it's committed to the screen. If the background re-render suspends, its Effects will run after the data loads and the UI updates.

During the initial render, the deferred value will be the same as the value you provided.

During updates, the deferred value will “lag behind” the latest value. In particular, React will first re-render without updating the deferred value, and then try to re-render with the newly received value in background.



Loading will not appear here.

- Deferring re-rendering for a part of the UI

You can also apply useDeferredValue as a performance optimization. It is useful when a part of your UI is slow to re-render, there’s no easy way to optimize it, and you want to prevent it from blocking the rest of the UI.

```
function App() {
  const [text, setText] = useState('');
  return (
    <>
      <input value={text} onChange={e => setText(e.target.value)} />
      <SlowList text={text} />
    </>
  );
}
```

```
const SlowList = memo(function SlowList({ text }) {
  // ...
});

function App() {
  const [text, setText] = useState('');
  const deferredText = useDeferredValue(text);
  return (
    <>
      <input value={text} onChange={e => setText(e.target.value)} />
      <SlowList text={deferredText} />
    </>
  );
}
```

This optimization requires SlowList to be wrapped in memo. This is because whenever the text changes, React needs to be able to re-render the parent component quickly. During that re-render, deferredText still has its previous value, so SlowList is able to skip re-rendering (its props have not changed). Without memo, it would have to re-render anyway, defeating the point of the optimization.

6. useEffect

useEffect is a React Hook that lets us to synchronize a component with an external system.

```
useEffect(setup, dependencies?)
```

```
import { useEffect } from 'react';
import { createConnection } from './chat.js';

function ChatRoom({ roomId }) {
  const [serverUrl, setServerUrl] = useState('https://localhost:1234');

  useEffect(() => {
    const connection = createConnection(serverUrl, roomId);
    connection.connect();
    return () => {
      connection.disconnect();
    };
  }, [serverUrl, roomId]);
  // ...
}
```

Parameters:

- setup: The function with your Effect's logic. Your setup function may also optionally return a cleanup function. When your component is first added to the DOM, React will run your setup function. After every re-render with changed dependencies, React will first run the cleanup function (if you provided it) with the old values, and then run your setup function with the new values. After your component is removed from the DOM, React will run your cleanup function one last time.

- optional dependencies: The list of all reactive values referenced inside of the setup code. Reactive values include props, state, and all the variables and functions declared directly inside your component body. If your linter is configured for React, it will verify that every reactive value is correctly specified as a dependency. The list of dependencies must have a constant number of items and be written inline like [dep1, dep2, dep3]. React will compare each dependency with its previous value using the Object.is comparison. If you omit this argument, your Effect will re-run after every re-render of the component. See the difference between passing an array of dependencies, an empty array, and no dependencies at all.

Returns

useEffect returns undefined.

Caveats:

- useEffect 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.
- If you’re not trying to synchronize with some external system, you probably don’t need an Effect.
- When Strict Mode is on, React will run one extra development-only setup+cleanup cycle before the first real setup. This is a stress-test that ensures that your cleanup logic “mirrors” your setup logic and that it stops or undoes whatever the setup is doing. If this causes a problem, implement the cleanup function.

- If some of your dependencies are objects or functions defined inside the component, there is a risk that they will cause the Effect to re-run more often than needed. To fix this, remove unnecessary object and function dependencies. You can also extract state updates and non-reactive logic outside of your Effect.
- If your Effect wasn't caused by an interaction (like a click), React will let the browser paint the updated screen first before running your Effect. If your Effect is doing something visual (for example, positioning a tooltip), and the delay is noticeable (for example, it flickers), replace useEffect with useLayoutEffect.
- Even if your Effect was caused by an interaction (like a click), the browser may repaint the screen before processing the state updates inside your Effect. Usually, that's what you want. However, if you must block the browser from repainting the screen, you need to replace useEffect with useLayoutEffect.
- Effects only run on the client. They don't run during server rendering.

Usage:

- Connecting to an external system

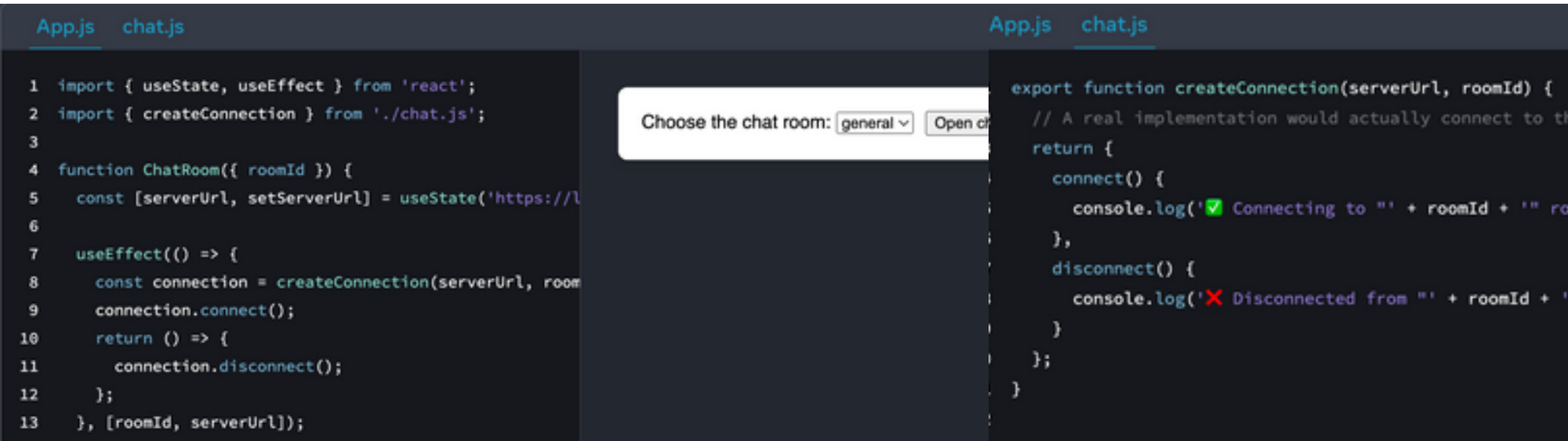
```
const [serverUrl, setServerUrl] = useState('https://localhost:1234');

useEffect(() => {
  const connection = createConnection(serverUrl, roomId);
  connection.connect();
  return () => {
    connection.disconnect();
  };
}, [serverUrl, roomId]);
// ...
```

You need to pass two arguments to useEffect:

- A setup function with setup code that connects to that system.
- It should return a cleanup function with cleanup code that disconnects from that system.
- A list of dependencies including every value from your component used inside of those functions.

React calls your setup and cleanup functions whenever it's necessary, which may happen multiple times:

- Your setup code runs when your component is added to the page (mounts).
- After every re-render of your component where the dependencies have changed:
- First, your cleanup code runs with the old props and state.
- Then, your setup code runs with the new props and state.
- Your cleanup code runs one final time after your component is removed from the page (unmounts).

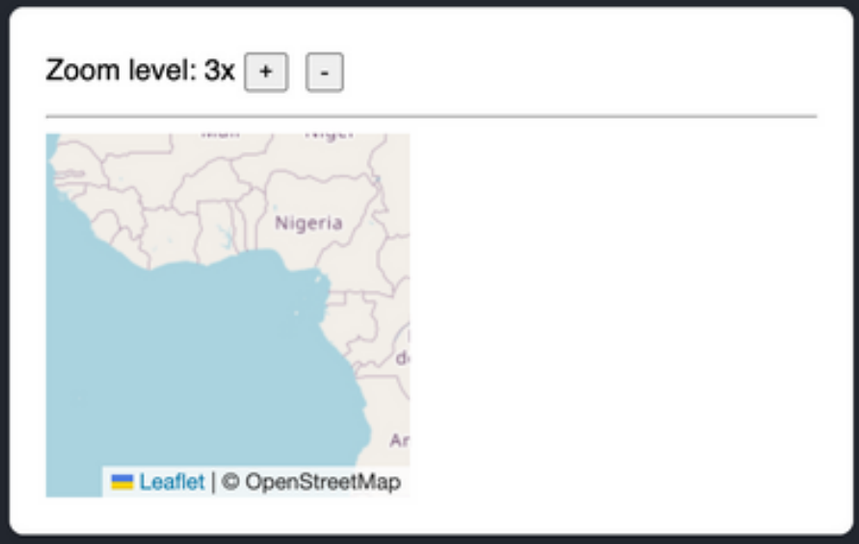


- Controlling a non-React widget

App.js Map.js map-widget.js Reset Fork

```
3
4  export default function Map({ zoomLevel }) {
5    const containerRef = useRef(null);
6    const mapRef = useRef(null);
7
8    useEffect(() => {
9      if (mapRef.current === null) {
10       mapRef.current = new MapWidget(containerRef.current);
11     }
12
13     const map = mapRef.current;
14     map.setZoom(zoomLevel);
15   }, [zoomLevel]);
16
17   return (
18     <div
19       style={{ width: 200, height: 200 }}
```

Show more



Fetching data with Effects:

```
useEffect(() => {
  let ignore = false;
  setBio(null);
  fetchBio(person).then(result => {
    if (!ignore) {
      setBio(result);
    }
  });
  return () => {
    ignore = true;
  };
}, [person]);
```

App.js

```
5    const [person, setPerson] = useState('Alice');
6    const [bio, setBio] = useState(null);
7    useEffect(() => {
8      async function startFetching() {
9        setBio(null);
10       const result = await fetchBio(person);
11       if (!ignore) {
12         setBio(result);
13       }
14     }
15
16     let ignore = false;
17     startFetching();
18     return () => {
19       ignore = true;
20     }
21   }, [person]);
```

Show more

Specifying reactive dependencies:

```
function ChatRoom({ roomId }) { // This is a reactive value
  const [serverUrl, setServerUrl] = useState('https://localhost:1234'); // This is a reactive value too

  useEffect(() => {
    const connection = createConnection(serverUrl, roomId); // This Effect reads these reactive values
    connection.connect();
    return () => connection.disconnect();
  }, [serverUrl, roomId]); // ✅ So you must specify them as dependencies of your Effect
  // ...
}
```

```
function ChatRoom({ roomId }) {
  const [serverUrl, setServerUrl] = useState('https://localhost:1234');

  useEffect(() => {
    const connection = createConnection(serverUrl, roomId);
    connection.connect();
    return () => connection.disconnect();
  }, []); // 🔴 React Hook useEffect has missing dependencies: 'roomId' and 'serverUrl'
  // ...
}
```

```
const serverUrl = 'https://localhost:1234'; // Not a reactive value anymore

function ChatRoom({ roomId }) {
  useEffect(() => {
    const connection = createConnection(serverUrl, roomId);
    connection.connect();
    return () => connection.disconnect();
  }, [roomId]); // ✅ All dependencies declared
  // ...
}
```

```
const serverUrl = 'https://localhost:1234'; // Not a reactive value anymore
const roomId = 'music'; // Not a reactive value anymore

function ChatRoom() {
  useEffect(() => {
    const connection = createConnection(serverUrl, roomId);
    connection.connect();
    return () => connection.disconnect();
  }, []); // ✅ All dependencies declared
  // ...
}
```

Updating state based on previous state from an Effect

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

  useEffect(() => {
    const intervalId = setInterval(() => {
      setCount(count + 1); // You want to increment the counter every second...
    }, 1000)
    return () => clearInterval(intervalId);
  }, [count]); // 🚩 ... but specifying `count` as a dependency always resets the interval.
  // ...
}
```

```
1  import { useState, useEffect } from 'react';
2
3  export default function Counter() {
4    const [count, setCount] = useState(0);
5
6    useEffect(() => {
7      const intervalId = setInterval(() => {
8        setCount(c => c + 1); // ✅ Pass a state updater
9      }, 1000);
10     return () => clearInterval(intervalId);
11   }, []); // ✅ Now count is not a dependency
12
```

- Removing unnecessary object dependencies:

```
const serverUrl = 'https://localhost:1234';

function ChatRoom({ roomId }) {
  const [message, setMessage] = useState('');

  const options = { // 🚩 This object is created from scratch on every re-render
    serverUrl: serverUrl,
    roomId: roomId
  };

  useEffect(() => {
    const connection = createConnection(options); // It's used inside the Effect
    connection.connect();
    return () => connection.disconnect();
  }, [options]); // 🚩 As a result, these dependencies are always different on a re-render
  // ...
```

App.js chat.js

Reset

```
function ChatRoom({ roomId }) {
  const [message, setMessage] = useState('');

  useEffect(() => {
    const options = {
      serverUrl: serverUrl,
      roomId: roomId
    };
    const connection = createConnection(options);
    connection.connect();
    return () => connection.disconnect();
  }, [roomId]);

  return (
    <>
      <h1>Welcome to the {roomId} room!</h1>
      <input value={message} onChange={e => setMessage(e.ta
```

Choose the chat room: general

Welcome to the general room!

Console (3)

Connecting to "general" room at https://localhost:1234.

Disconnected from "general" room at https://localhost:1

Connecting to "general" room at https://localhost:1234.

Show more

- Reading the latest props and state from an Effect

```
function Page({ url, shoppingCart }) {
  useEffect(() => {
    logVisit(url, shoppingCart.length);
  }, [url, shoppingCart]); // ✅ All dependencies declared
  // ...
}

function Page({ url, shoppingCart }) {
  const onVisit = useEffectEvent(visitedUrl => {
    logVisit(visitedUrl, shoppingCart.length)
  });

  useEffect(() => {
    onVisit(url);
  }, [url]); // ✅ All dependencies declared
  // ...
}
```

By reading shoppingCart inside of onVisit, you ensure that shoppingCart won't re-run your Effect.

https://react.dev/learn/separating-events-from-effects#reading-latest-props-and-state-with-effect-events

Displaying different content on the server and the client:

If your app uses server rendering (either directly or via a framework), your component will render in two different environments. On the server, it will render to produce the initial HTML. On the client, React will run the rendering code again so that it can attach your event handlers to that HTML. This is why, for hydration to work, your initial render output must be identical on the client and the server.

```
function MyComponent() {
  const [didMount, setDidMount] = useState(false);

  useEffect(() => {
    setDidMount(true);
  }, []);

  if (didMount) {
    // ... return client-only JSX ...
  }  else {
    // ... return initial JSX ...
  }
}
```

If your app uses server rendering (either directly or via a framework), your component will render in two different environments. On the server, it will render to produce the initial HTML. On the client, React will run the rendering code again so that it can attach your event handlers to that HTML. This is why, for hydration to work, your initial render output must be identical on the client and the server.

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

If your Effect must block the browser from painting the screen, replace useEffect with useLayoutEffect. Note that this shouldn’t be needed for the vast majority of Effects. You’ll only need this if it’s crucial to run your Effect before the browser paint: for example, to measure and position a tooltip before the user sees it.

7. useId

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

```
const id = useId()
```

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

function PasswordField() {
  const passwordHintId = useId();
  // ...
```

Parameters:

- useId does not take any parameters.

Returns:

- useId returns a unique ID string associated with this particular useId call in this particular component.

Caveats:

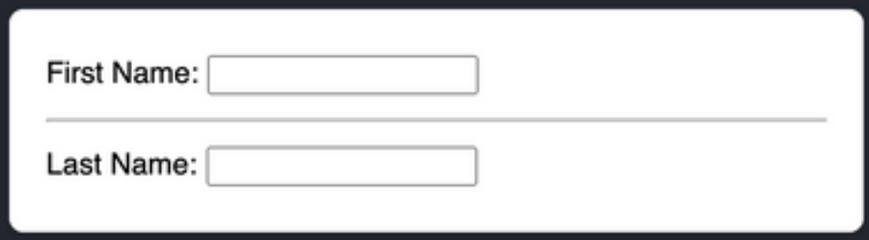
- useId is a Hook, so you can only call it at the top level of your component or your own Hooks. You can’t call it inside loops or conditions. If you need that, extract a new component and move the state into it.
- useId should not be used to generate keys in a list. Keys should be generated from your data.

- Do not call useId to generate keys in a list. Keys should be generated from your data.

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

function PasswordField() {
  const passwordHintId = useId();
  // ...
<>
  <input type="password" aria-describedby={passwordHintId} />
  <p id={passwordHintId}>
</>

<label>
  Password:
  <input
    type="password"
    aria-describedby="password-hint"
  />
</label>
<p id="password-hint">
  The password should contain at least 18 characters
</p>

import { useId } from 'react';

function PasswordField() {
  const passwordHintId = useId();
  return (
    <>
      <label>
        Password:
        <input
          type="password"
          aria-describedby={passwordHintId}
        />
      </label>
```

- Generating IDs for several related elements

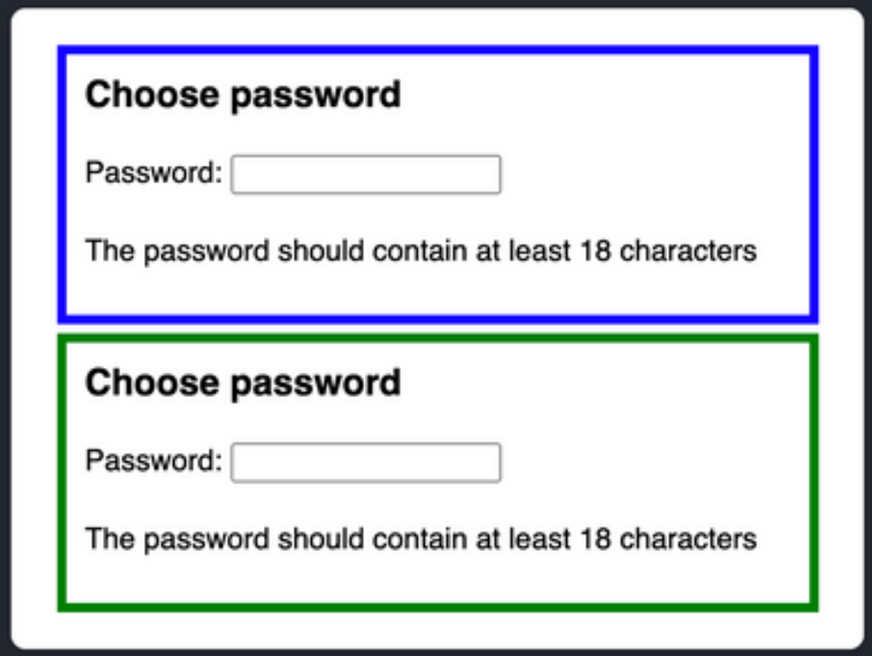
```jsx
import { useId } from 'react';

export default function Form() {
  const id = useId();
  return (
    <form>
      <label htmlFor={id + '-firstName'}>First Name:</label
      <input id={id + '-firstName'} type="text" />
      <hr />
      <label htmlFor={id + '-lastName'}>Last Name:</label>
      <input id={id + '-lastName'} type="text" />
    </form>
  );
}

```



Specifying a shared prefix for all generated IDs

index.html App.js index.js Reset Fork

```jsx
import { createRoot } from 'react-dom/client';
import App from './App.js';
import './styles.css';

const root1 = createRoot(document.getElementById('root1'),
  identifierPrefix: 'my-first-app-'
});
root1.render(<App />);

const root2 = createRoot(document.getElementById('root2'),
  identifierPrefix: 'my-second-app-'
});
root2.render(<App />);

```



index.html App.js index.js Reset Fork

```jsx

function PasswordField() {
  const passwordHintId = useId();
  console.log('Generated identifier:', passwordHintId)
```

8. useImperativeHandle

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

```
useImperativeHandle(ref, createHandle, dependencies?)
```

```
import { forwardRef, useImperativeHandle } from 'react';

const MyInput = forwardRef(function MyInput(props, ref) {
  useImperativeHandle(ref, () => {
    return {
      // ... your methods ...
    };
  }, []);
  // ...
```

Parameters:

ref: The ref you received as the second argument from the forwardRef render function.

createHandle: A function that takes no arguments and returns the ref handle you want to expose. That ref handle can have any type. Usually, you will return an object with the methods you want to expose.

optional dependencies: The list of all reactive values referenced inside of the createHandle code. Reactive values include props, state, and all the variables and functions declared directly inside your component body. If your linter is configured for React, it will verify that every reactive value is correctly specified as a dependency.

The list of dependencies must have a constant number of items and be written inline like [dep1, dep2, dep3]. React will compare each dependency with its previous value using the Object.is comparison. If a re-render resulted in a change to some dependency, or if you omitted this argument, your createHandle function will re-execute, and the newly created handle will be assigned to the ref.

Returns:

useImperativeHandle returns undefined.

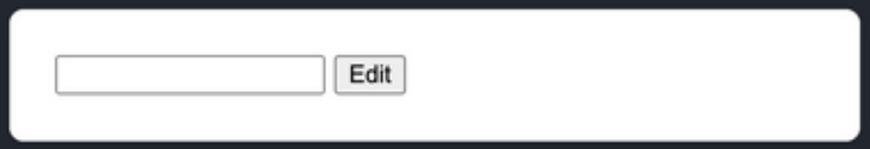
Usage:

- Exposing a custom ref handle to the parent component

```
const MyInput = forwardRef(function MyInput(props, ref) {
  const inputRef = useRef(null);

  useImperativeHandle(ref, () => {
    return {
      focus() {
        inputRef.current.focus();
      },
      scrollIntoView() {
        inputRef.current.scrollIntoView();
      },
    };
  }, []);

  return <input {...props} ref={inputRef} />;
});
```

App.js MyInput.js Reset Fork

```
import { useRef } from 'react';
import MyInput from './MyInput.js';

export default function Form() {
  const ref = useRef(null);

  function handleClick() {
    ref.current.focus();
    // This won't work because the DOM node isn't exposed:
    // ref.current.style.opacity = 0.5;
  }

  return (
    <form>
      <MyInput label="Enter your name:" ref={ref} />
      <button type="button" onClick={handleClick}>
```

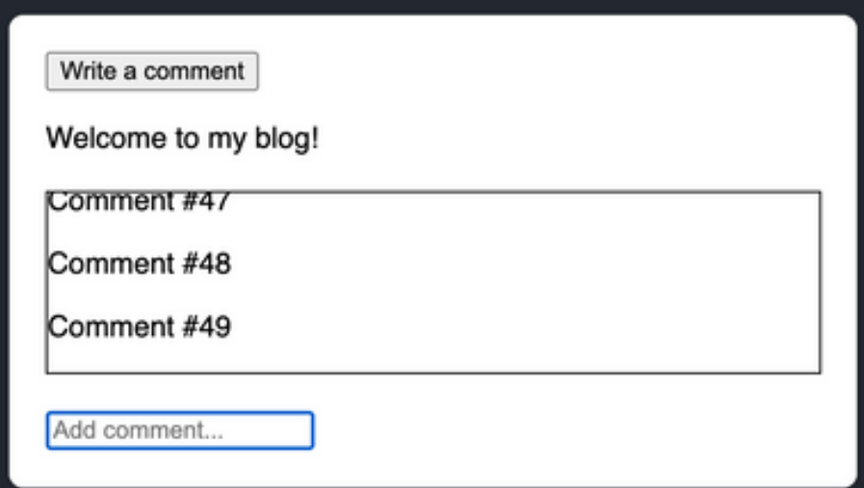


If you can express something as a prop, you should not use a ref. For example, instead of exposing an imperative handle like { open, close } from a Modal component, it is better to take isOpen as a prop like <Modal isOpen={isOpen} />. Effects can help you expose imperative behaviors via props.

- Exposing your own imperative methods:

App.js Post.js CommentList.js AddComment.js Reset Fork

```
  useImperativeHandle(ref, () => {
    return {
      scrollAndFocusAddComment() {
        commentsRef.current.scrollToBottom();
        addCommentRef.current.focus();
      }
    };
  }, []);

  return (
    <>
      <article>
        <p>Welcome to my blog!</p>
      </article>
      <CommentList ref={commentsRef} />
      <AddComment ref={addCommentRef} />
```



Do not overuse refs. You should only use refs for imperative behaviors that you can't express as props: for example, scrolling to a node, focusing a node, triggering an animation, selecting text, and so on.

9. useInsertionEffect

useInsertionEffect is for CSS-in-JS library authors. Unless you are working on a CSS-in-JS library and need a place to inject the styles, you probably want useEffect or useLayoutEffect instead.

useInsertionEffect is a version of useEffect that fires before any DOM mutations.

```
useInsertionEffect(setup, dependencies?)
```

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

// Inside your CSS-in-JS library
function useCSS(rule) {
  useInsertionEffect(() => {
    // ... inject <style> tags here ...
  });
  return rule;
}
```

Parameters:

- setup: The function with your Effect's logic. Your setup function may also optionally return a cleanup function. Before your component is first added to the DOM, React will run your setup function. After every re-render with changed dependencies, React will first run the cleanup function (if you provided it) with the old values, and then run your setup function with the new values. Before your component is removed from the DOM, React will run your cleanup function one last time.

- optional dependencies: The list of all reactive values referenced inside of the setup code. Reactive values include props, state, and all the variables and functions declared directly inside your component body. If your linter is configured for React, it will verify that every reactive value is correctly specified as a dependency. The list of dependencies must have a constant number of items and be written inline like [dep1, dep2, dep3]. React will compare each dependency with its previous value using the Object.is comparison algorithm. If you don't specify the dependencies at all, your Effect will re-run after every re-render of the component.

Returns

- useInsertionEffect returns undefined.

Caveats

- Effects only run on the client. They don't run during server rendering.
- You can't update state from inside useInsertionEffect.
- By the time useInsertionEffect runs, refs are not attached yet, and DOM is not yet updated.

Usage

- Injecting dynamic styles from CSS-in-JS libraries

```
// In your JS file:
<button className="success" />

// In your CSS file:
.success { color: green; }
```

```
// Inside your CSS-in-JS library
let isInserted = new Set();
function useCSS(rule) {
  useInsertionEffect(() => {
    // As explained earlier, we don't recommend runtime injection of <style> tags.
    // But if you have to do it, then it's important to do in useInsertionEffect.
    if (!isInserted.has(rule)) {
      isInserted.add(rule);
      document.head.appendChild(getStyleForRule(rule));
    }
  });
  return rule;
}

function Button() {
  const className = useCSS('...');
  return <div className={className} />;
}

let collectedRulesSet = new Set();

function useCSS(rule) {
  if (typeof window === 'undefined') {
    collectedRulesSet.add(rule);
  }
  useInsertionEffect(() => {
    // ...
  });
  return rule;
}
```

Similarly to useEffect, useInsertionEffect does not run on the server. If you need to collect which CSS rules have been used on the server, you can do it during rendering:

10. useLayoutEffect

useLayoutEffect can hurt performance. Prefer useEffect when possible.

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

```
useLayoutEffect(setup, dependencies?)
```

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

function Tooltip() {
  const ref = useRef(null);
  const [tooltipHeight, setTooltipHeight] = useState(0);

  useLayoutEffect(() => {
    const { height } = ref.current.getBoundingClientRect();
    setTooltipHeight(height);
  }, []);
  // ...
```

Parameters

- setup: The function with your Effect's logic. Your setup function may also optionally return a cleanup function. Before your component is first added to the DOM, React will run your setup function. After every re-render with changed dependencies, React will first run the cleanup function (if you provided it) with the old values, and then run your setup function with the new values. Before your component is removed from the DOM, React will run your cleanup function one last time.

- optional dependencies: The list of all reactive values referenced inside of the setup code. Reactive values include props, state, and all the variables and functions declared directly inside your component body. If your linter is configured for React, it will verify that every reactive value is correctly specified as a dependency. The list of dependencies must have a constant number of items and be written inline like [dep1, dep2, dep3]. React will compare each dependency with its previous value using the Object.is comparison. If you omit this argument, your Effect will re-run after every re-render of the component.

Returns

- useLayoutEffect returns undefined.

Caveats

- useLayoutEffect 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 component and move the Effect there.
- When Strict Mode is on, React will run one extra development-only setup+cleanup cycle before the first real setup. This is a stress-test that ensures that your cleanup logic “mirrors” your setup logic and that it stops or undoes whatever the setup is doing. If this causes a problem, implement the cleanup function.

- If some of your dependencies are objects or functions defined inside the component, there is a risk that they will cause the Effect to re-run more often than needed. To fix this, remove unnecessary object and function dependencies. You can also extract state updates and non-reactive logic outside of your Effect.
- Effects only run on the client. They don't run during server rendering.
- The code inside useLayoutEffect and all state updates scheduled from it block the browser from repainting the screen. When used excessively, this makes your app slow. When possible, prefer useEffect

Usage

Measuring layout before the browser repaints the screen
Most components don't need to know their position and size on the screen to decide what to render. They only return some JSX. Then the browser calculates their layout (position and size) and repaints the screen.

Sometimes, that's not enough. Imagine a tooltip that appears next to some element on hover. If there's enough space, the tooltip should appear above the element, but if it doesn't fit, it should appear below. In order to render the tooltip at the right final position, you need to know its height (i.e. whether it fits at the top).

To do this, you need to render in two passes:

- Render the tooltip anywhere (even with a wrong position).

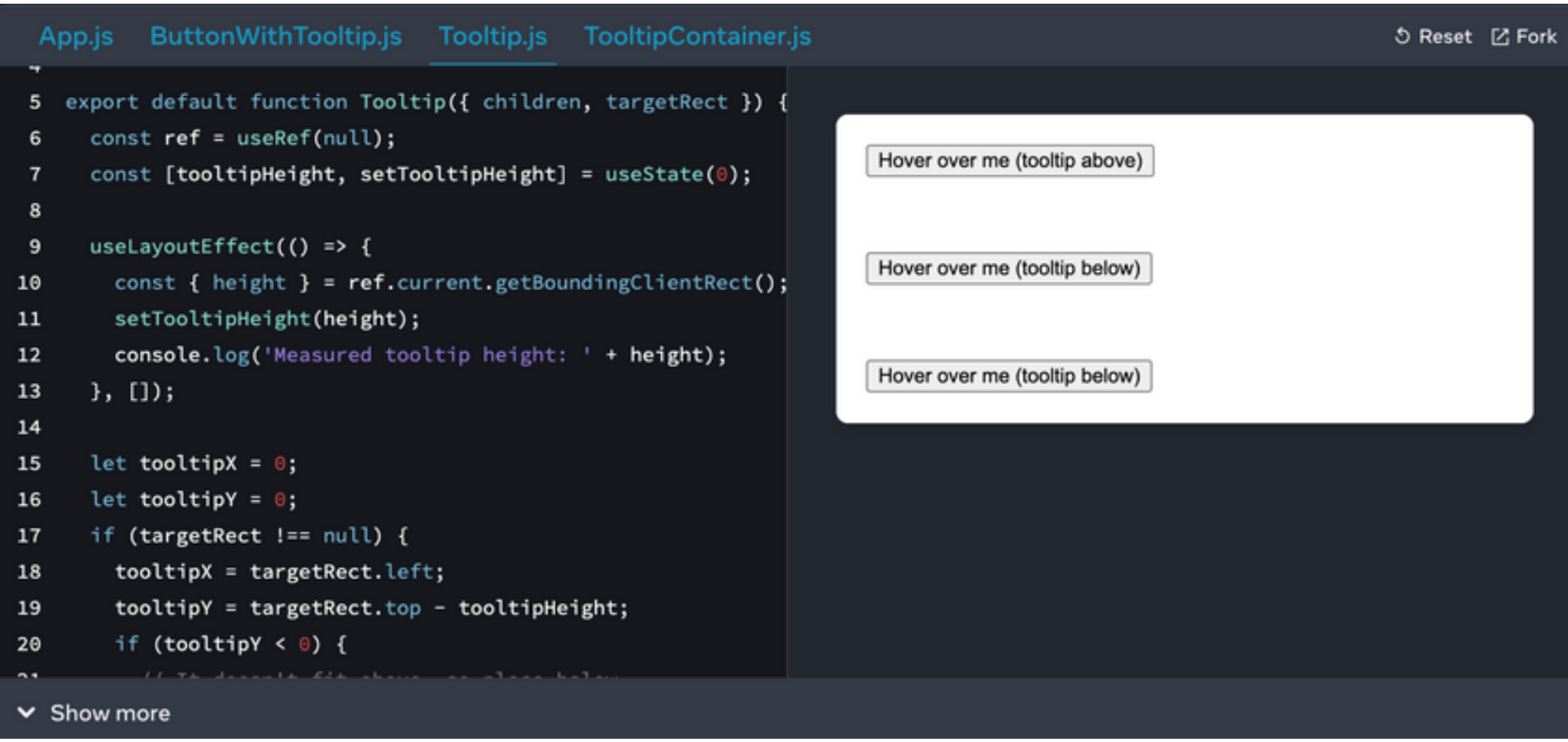
- Measure its height and decide where to place the tooltip.
- Render the tooltip again in the correct place.

All of this needs to happen before the browser repaints the screen. You don't want the user to see the tooltip moving. Call useLayoutEffect to perform the layout measurements before the browser repaints the screen:

```
function Tooltip() {
  const ref = useRef(null);
  const [tooltipHeight, setTooltipHeight] = useState(0); // You don't know real height yet

  useLayoutEffect(() => {
    const { height } = ref.current.getBoundingClientRect();
    setTooltipHeight(height); // Re-render now that you know the real height
  }, []);

  // ...use tooltipHeight in the rendering logic below...
}
```

Here's how this works step by step:

- Tooltip renders with the initial tooltipHeight = 0 (so the tooltip may be wrongly positioned).
- React places it in the DOM and runs the code in useLayoutEffect.
- Your useLayoutEffect measures the height of the tooltip content and triggers an immediate re-render.
- Tooltip renders again with the real tooltipHeight (so the tooltip is correctly positioned).
- React updates it in the DOM, and the browser finally displays the tooltip.



Notice that even though the Tooltip component has to render in two passes (first, with tooltipHeight initialized to 0 and then with the real measured height), you only see the final result. This is why you need useLayoutEffect instead of useEffect for this example.

the useLayoutEffect call in the Tooltip component lets it position itself correctly

If you synchronize your component with an external data store and rely on useLayoutEffect for different reasons than measuring layout, consider useSyncExternalStore instead which supports server rendering.

11. useReducer

useReducer is a React Hook that lets user to add a reducer to the component.

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

Reference:

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 types.

- 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.

dispatch function:

The dispatch function returned by useReducer lets you update the state to a different value and trigger a re-render. You need to pass the action as the only argument to the dispatch function:

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

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

React will set the next state to the result of calling the reducer function you've provided with the current state and the action you've passed to dispatch.

Parameters

- action: The action performed by the user. It can be a value of any type. By convention, an action is usually an object with a type property identifying it and, optionally, other properties with additional information.

Returns
dispatch functions do not have a return value.

Caveats

- The dispatch function only updates the state variable for the next render.

- If you read the state variable after calling the dispatch function, you will still get the old value that was on the screen before your call.
- If the new value you provide is identical to the current state, as determined by an Object.is comparison, React will skip re-rendering the component and its children. This is an optimization. React may still need to call your component before ignoring the result, but it shouldn't affect your code.
- React batches state updates. It updates the screen after all the event handlers have run and have called their set functions. This prevents multiple re-renders during a single event. In the rare case that you need to force React to update the screen earlier, for example to access the DOM, you can use flushSync.

Usage:

- Adding a reducer to a component

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:

1. The current state of this state variable, initially set to the initial state you provided.
2. 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.

useReducer is very similar to useState, but it lets you move the state update logic from event handlers into a single function outside of your component.

```
function reducer(state, action) {
  switch (action.type) {
    case 'incremented_age': {
      return {
        name: state.name,
        age: state.age + 1
      };
    }
    case 'changed_name': {
      return {
        name: action.nextName,
        age: state.age
      };
    }
```

```
function Form() {
  const [state, dispatch] = useReducer(reducer, { name: 'Taylor', age: 42 });

  function handleButtonClick() {
    dispatch({ type: 'incremented_age' });
  }

  function handleInputChange(e) {
    dispatch({
      type: 'changed_name',
      nextName: e.target.value
    });
  }
  // ...
```

- Avoiding recreating the initial state

```
function createInitialState(username) {
  // ...
}

function TodoList({ username }) {
  const [state, dispatch] = useReducer(reducer, createInitialState(username));
  // ...

function createInitialState(username) {
  // ...
}

function TodoList({ username }) {
  const [state, dispatch] = useReducer(reducer, username, createInitialState);
  // ...
```

12. useRef

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

```
const ref = useRef(initialValue)
```

Reference:

useRef(initialValue)
Call useRef at the top level of your component to declare a ref.

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

function MyComponent() {
  const intervalRef = useRef(0);
  const inputRef = useRef(null);
  // ...
```

Parameters

- initialValue: The value you want the ref object’s current property to be initially. It can be a value of any type. This argument is ignored after the initial render.

Returns

- useRef returns an object with a single property:

current: Initially, it’s set to the initialValue you have passed. You can later set it to something else. If you pass the ref object to React as a ref

attribute to a JSX node, React will set its current property.
On the next renders, useRef will return the same object.

Caveats

- You can mutate the ref.current property. Unlike state, it is mutable. However, if it holds an object that is used for rendering (for example, a piece of your state), then you shouldn't mutate that object.
- When you change the ref.current property, React does not re-render your component. React is not aware of when you change it because a ref is a plain JavaScript object.
- Do not write or read ref.current during rendering, except for initialization. This makes your component's behavior unpredictable.
- In Strict Mode, React will call your component function twice in order to help you find accidental impurities. This is development-only behavior and does not affect production. Each ref object will be created twice, but one of the versions will be discarded. If your component function is pure (as it should be), this should not affect the behavior.

Usage

- Referencing a value with a ref

Call useRef at the top level of your component to declare one or more refs.

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

function Stopwatch() {
  const intervalRef = useRef(0);
  // ...
```

useRef returns a ref object with a single current property initially set to the initial value you provided.

On the next renders, useRef will return the same object. You can change its current property to store information and read it later. This might remind you of state, but there is an important difference.

Changing a ref does not trigger a re-render. This means refs are perfect for storing information that doesn't affect the visual output of your component. For example, if you need to store an interval ID and retrieve it later, you can put it in a ref. To update the value inside the ref, you need to manually change its current property:

```
function handleStartClick() {
  const intervalId = setInterval(() => {
    // ...
  }, 1000);
  intervalRef.current = intervalId;
}
```

Later, you can read that interval ID from the ref so that you can call clear that interval:

```
function handleStopClick() {
  const intervalId = intervalRef.current;
  clearInterval(intervalId);
```

By using a ref, you ensure that:

- You can store information between re-renders (unlike regular variables, which reset on every render).
- Changing it does not trigger a re-render (unlike state variables, which trigger a re-render).
- The information is local to each copy of your component (unlike the variables outside, which are shared).

Changing a ref does not trigger a re-render, so refs are not appropriate for storing information you want to display on the screen. Use state for that instead. Read more about choosing between useRef and useState.

- Manipulating the DOM with a ref

It's particularly common to use a ref to manipulate the DOM. React has built-in support for this.

First, declare a ref object with an initial value of null:

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

function MyComponent() {
  const inputRef = useRef(null);

  return <input ref={inputRef} />;

  function handleClick() {
    inputRef.current.focus();
  }
```

- Avoiding recreating the ref contents

```
function Video() {
  const playerRef = useRef(new VideoPlayer());
  // ...
```

Although the result of new VideoPlayer() is only used for the initial render, you're still calling this function on every render. This can be wasteful if it's creating expensive objects.

```
function Video() {
  const playerRef = useRef(null);
  if (playerRef.current === null) {
    playerRef.current = new VideoPlayer();
  }
  // ...
```

13. useState

useState is a React Hook that lets user to add a state variable to the component.

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

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

function MyComponent() {
  const [age, setAge] = useState(28);
  const [name, setName] = useState('Taylor');
  const [todos, setTodos] = useState(() => createTodos());
  // ...
```

Parameters:

- initialState: The value you want the state to be initially. It can be a value of any type, but there is a special behavior for functions. This argument is ignored after the initial render.
- If you pass a function as initialState, it will be treated as an initializer function. It should be pure, should take no arguments, and should return a value of any type. React will call your initializer function when initializing the component, and store its return value as the initial state. See an example below.

Returns

useState returns an array with exactly two values:

- The current state. During the first render, it will match the initialState you have passed.
- The set function that lets you update the state to a different value and trigger a re-render.

Caveats

- useState 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 initializer function twice in order to help you find accidental impurities. This is development-only behavior and does not affect production. If your initializer function is pure (as it should be), this should not affect the behavior. The result from one of the calls will be ignored.

set functions, like setSomething(nextState)

The set function returned by useState lets you update the state to a different value and trigger a re-render. You can pass the next state directly, or a function that calculates it from the previous state:

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

function handleClick() {
  setName('Taylor');
  setAge(a => a + 1);
  // ...
```

Parameters

nextState: The value that you want the state to be. It can be a value of any type, but there is a special behavior for functions.

If you pass a function as nextState, it will be treated as an updater function. It must be pure, should take the pending state as its only argument, and should return the next state. React will put your updater function in a queue and re-render your component. During the next render, React will calculate the next state by applying all of the queued updaters to the previous state.

Returns
set functions do not have a return value.
Caveats

- The set function only updates the state variable for the next render. If you read the state variable after calling the set function, you will still get the old value that was on the screen before your call.
- If the new value you provide is identical to the current state, as determined by an Object.is comparison, React will skip re-rendering the component and its children. This is an optimization. Although in some cases React may still need to call your component before skipping the children, it shouldn't affect your code.
- React batches state updates. It updates the screen after all the event handlers have run and have called their set functions. This prevents multiple re-renders during a single event. In the rare case that you need to force React to update the screen earlier, for example to access the DOM, you can use flushSync.
- Calling the set function during rendering is only allowed from within the currently rendering component. React will discard its output and immediately attempt to render it again with the new state. This pattern is rarely needed, but you can use it to store information from the previous renders.

- In Strict Mode, React will call your updater function twice in order to help you find accidental impurities. This is development-only behavior and does not affect production. If your updater function is pure (as it should be), this should not affect the behavior. The result from one of the calls will be ignored.

```
function handleClick() {
  setAge(a => a + 1); // setAge(42 => 43)
  setAge(a => a + 1); // setAge(43 => 44)
  setAge(a => a + 1); // setAge(44 => 45)
}
```

```
function handleClick() {
  setAge(age + 1); // setAge(42 + 1)
  setAge(age + 1); // setAge(42 + 1)
  setAge(age + 1); // setAge(42 + 1)
}
```

App.js

Download Reset Fork

```
1  import { useState } from 'react';
2
3  export default function Counter() {
4    const [age, setAge] = useState(42);
5
6    function increment() {
7      setAge(a => a + 1);
8    }
9
10   return (
11     <>
12       <h1>Your age: {age}</h1>
13       <button onClick={() => {
14         increment();
15         increment();
16         increment();
```

Your age: 42

+3

+1

Show more

- Updating objects and arrays in state

Updating objects and arrays in state

You can put objects and arrays into state. In React, state is considered read-only, so **you should *replace* it rather than *mutate* your existing objects**. For example, if you have a `form` object in state, don't mutate it:

```
// 🚩 Don't mutate an object in state like this:
form.firstName = 'Taylor';
```

Instead, replace the whole object by creating a new one:

```
// ✅ Replace state with a new object
setForm({
  ...form,
  firstName: 'Taylor'
});
```

Avoiding recreating the initial state

React saves the initial state once and ignores it on the next renders.

```
function TodoList() {
  const [todos, setTodos] = useState(createInitialTodos());
  // ...
```

Although the result of `createInitialTodos()` is only used for the initial render, you're still calling
on every render. This can be wasteful if it's creating large arrays or performing expensive calculatio

To solve this, you may **pass it as an *initializer* function** to `useState` instead:

```
function TodoList() {
  const [todos, setTodos] = useState(createInitialTodos);
  // ...
```

Notice that you're passing `createInitialTodos`, which is the *function itself*, and not `createIniti`
which is the result of calling it. If you pass a function to `useState`, React will only call it during init

React may call your initializers twice in development to verify that they are pure.

- Resetting state with a key

```
mport { useState } from 'react';

xport default function App() {
 const [version, setVersion] = useState(0);

 function handleReset() {
   setVersion(version + 1);
 }

 return (
   <>
     <button onClick={handleReset}>Reset</button
     <Form key={version} />
   </>
 );
```

- Storing information from previous renders

App.js CountLabel.js

Reset Fork

```
1  import { useState } from 'react';
2
3  export default function CountLabel({ count }) {
4    const [prevCount, setPrevCount] = useState(count);
5    const [trend, setTrend] = useState(null);
6    if (prevCount !== count) {
7      setPrevCount(count);
8      setTrend(count > prevCount ? 'increasing' : 'decreasing
9    }
10   return (
11     <>
12       <h1>{count}</h1>
13       {trend && <p>The count is {trend}</p>}
14     </>
15   );
16 }
```

Increment Decrement

0

App.js CountLabel.js

/CountLabel.js

```
1  import { useState } from 'react';
2  import CountLabel from './CountLabel.js';
3
4  export default function App() {
5    const [count, setCount] = useState(0);
6    return (
7      <>
8        <button onClick={() => setCount(count + 1)}>
9          Increment
10       </button>
11       <button onClick={() => setCount(count - 1)}>
12         Decrement
13       </button>
14       <CountLabel count={count} />
15     </>
16   );
```

Increment Decrement

1

The count is decreasing

Show more

14. useSyncExternalStore

useSyncExternalStore is a React Hook that lets user to subscribe to an external store.

```
import { useSyncExternalStore } from 'react';
import { todosStore } from './todoStore.js';

function TodosApp() {
  const todos = useSyncExternalStore(todosStore.subscribe, todosStore.getSnapshot);
  // ...
}
```

It returns the snapshot of the data in the store. You need to pass two functions as arguments:

- The subscribe function should subscribe to the store and return a function that unsubscribes.
- The getSnapshot function should read a snapshot of the data from the store.

Parameters:

- subscribe: A function that takes a single callback argument and subscribes it to the store. When the store changes, it should invoke the provided callback. This will cause the component to re-render. The subscribe function should return a function that cleans up the subscription.

- getSnapshot: A function that returns a snapshot of the data in the

Returns

The current snapshot of the store which you can use in your rendering logic.

Caveats

- The store snapshot returned by getSnapshot must be immutable. If the underlying store has mutable data, return a new immutable snapshot if the data has changed. Otherwise, return a cached last snapshot.
- If a different subscribe function is passed during a re-render, React will re-subscribe to the store using the newly passed subscribe function. You can prevent this by declaring subscribe outside the component.

Usage

- Subscribing to an external store

```
import { useSyncExternalStore } from 'react';
import { todosStore } from './todoStore.js';

function TodosApp() {
  const todos = useSyncExternalStore(todosStore.subscribe, todosStore.getSnapshot);
  // ...
}
```

App.js todoStore.js

Reset Fork

```
import { useSyncExternalStore } from 'react';
import { todosStore } from './todoStore.js';

export default function TodosApp() {
  const todos = useSyncExternalStore(todosStore.subscribe,
                                     todosStore.getSnapshot);

  return (
    <>
      <button onClick={() => todosStore.addTodo()}>
        Add todo
      </button>
      <hr />
      <ul>
        {todos.map(todo => (
          <li key={todo.id}>{todo.text}</li>
```

App.js todoStore.js

Reset Fork

```
let nextId = 0;
let todos = [{ id: nextId++, text: 'Todo #1' }];
let listeners = [];

export const todosStore = {
  addTodo() {
    todos = [...todos, { id: nextId++,
                         text: 'Todo #' + nextId }]
    emitChange();
  },
  subscribe(listener) {
    listeners = [...listeners, listener];
    return () => {
      listeners = listeners.filter(l => l !== listener);
    };
  },
  getSnapshot() {
    return todos;
  }
};

function emitChange() {
  for (let listener of listeners) {
    listener();
```

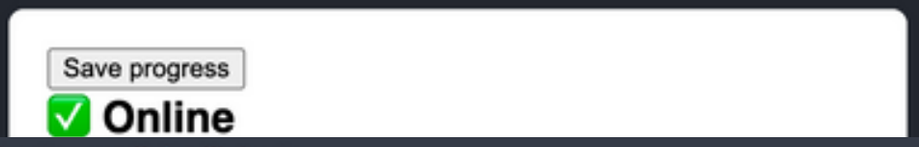
- Subscribing to a browser API

```jsx
import { useSyncExternalStore } from 'react';

function ChatIndicator() {
  const isOnline = useSyncExternalStore(subscribe, getSnapshot);
  // ...
}

function getSnapshot() {
  return navigator.onLine;
}

function subscribe(callback) {
  window.addEventListener('online', callback);
  window.addEventListener('offline', callback);
  return () => {
    window.removeEventListener('online', callback);
    window.removeEventListener('offline', callback);
  };
}
```

App.js useOnlineStatus.js

Reset Fo

```jsx
function StatusBar() {
  const isOnline = useOnlineStatus();
  return <h1>{isOnline ? '✅ Online' : '❌ Disconnected'}</h
}

function SaveButton() {
  const isOnline = useOnlineStatus();

  function handleSaveClick() {
    console.log('✅ Progress saved');
  }

  return (
    <button disabled={!isOnline} onClick={handleSaveClick}>
      {isOnline ? 'Save progress' : 'Reconnecting...'}
    </button>
```



App.js useOnlineStatus.js

```jsx
export function useOnlineStatus() {
  const isOnline = useSyncExternalStore(subscribe,
                                        getSnapshot);
  return isOnline;
}

function getSnapshot() {
  return navigator.onLine;
}

function subscribe(callback) {
  window.addEventListener('online', callback);
  window.addEventListener('offline', callback);
  return () => {
    window.removeEventListener('online', callback);
    window.removeEventListener('offline', callback);
  };
```

- Adding support for server rendering

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

export function useOnlineStatus() {
  const isOnline = useSyncExternalStore(subscribe, getSnapshot, getServerSnapshot);
  return isOnline;
}

function getSnapshot() {
  return navigator.onLine;
}

function getServerSnapshot() {
  return true; // Always show "Online" for server-generated HTML
}

function subscribe(callback) {
  // ...
}
```

The react-dom/server APIs let you render React components to HTML on the server. These APIs are only used on the server at the top level of your app to generate the initial HTML.

The getServerSnapshot function is similar to getSnapshot, but it runs only in two situations:

- It runs on the server when generating the HTML.
- It runs on the client during hydration, i.e. when React takes the server HTML and makes it interactive.

This lets you provide the initial snapshot value which will be used before the app becomes interactive. If there is no meaningful initial value for the server rendering, omit this argument to force rendering on the client.

15. useTransition

useTransition is a React Hook that lets the user update the state without blocking the UI.

```
const [isPending, startTransition] = useTransition()
```

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

function TabContainer() {
  const [isPending, startTransition] = useTransition();
  // ...
}
```

Parameters:

useTransition does not take any parameters.

Returns:

useTransition returns an array with exactly two items:

- The isPending flag that tells you whether there is a pending transition.
- The startTransition function that lets you mark a state update as a transition.
- startTransition function
- The startTransition function returned by useTransition lets you mark a state update as a transition.

```
function TabContainer() {
  const [isPending, startTransition] = useTransition();
  const [tab, setTab] = useState('about');

  function selectTab(nextTab) {
    startTransition(() => {
      setTab(nextTab);
    });
  }
  // ...
}
```

Parameters

scope: A function that updates some state by calling one or more set functions. React immediately calls scope with no parameters and marks all state updates scheduled synchronously during the scope function call as transitions. They will be non-blocking and will not display unwanted loading indicators.

Returns

startTransition does not return anything.

Caveats

useTransition is a Hook, so it can only be called inside components or custom Hooks. If you need to start a transition somewhere else (for example, from a data library), call the standalone startTransition instead.

You can wrap an update into a transition only if you have access to the set function of that state.

If you want to start a transition in response to some prop or a custom Hook value, try useDeferredValue instead.

The function you pass to startTransition must be synchronous. React immediately executes this function, marking all state updates that happen while it executes as transitions. If you try to perform more state updates later (for example, in a timeout), they won't be marked as transitions.

A state update marked as a transition will be interrupted by other state updates. For example, if you update a chart component inside a transition, but then start typing into an input while the chart is in the middle of a re-render, React will restart the rendering work on the chart component after handling the input update.

Transition updates can't be used to control text inputs.

If there are multiple ongoing transitions, React currently batches them together. This is a limitation that will likely be removed in a future release.

Usage:

- Marking a state update as a non-blocking transition

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

function TabContainer() {
  const [isPending, startTransition] = useTransition();
  // ...
}
```

```
function TabContainer() {
  const [isPending, startTransition] = useTransition();
  const [tab, setTab] = useState('about');

  function selectTab(nextTab) {
    startTransition(() => {
      setTab(nextTab);
    });
  }
  // ...
}
```

Transitions let you keep the user interface updates responsive even on slow devices.

- Updating the parent component in a transition

```
export default function TabButton({ children, isActive, onClick }) {
  const [isPending, startTransition] = useTransition();
  if (isActive) {
    return <b>{children}</b>
  }
  return (
    <button onClick={() => {
      startTransition(() => {
        onClick();
      });
    }}>
      {children}
    </button>
```

App.js TabButton.js AboutTab.js PostsTab.js ContactTab.js Reset Fork

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

export default function TabButton({ children, isActive, onC
  const [isPending, startTransition] = useTransition();
  if (isActive) {
    return <b>{children}</b>
  }
  return (
    <button onClick={() => {
      startTransition(() => {
        onClick();
      });
    }}>
      {children}
    </button>
  );
```

About **Posts (slow)** Contact

- Post #1
- Post #2
- Post #3
- Post #4
- Post #5
- Post #6

Console (14)

```
[ARTIFICIALLY SLOW] Rendering 500 <SlowPost />
[ARTIFICIALLY SLOW] Rendering 500 <SlowPost />
[ARTIFICIALLY SLOW] Rendering 500 <SlowPost />
[ARTIFICIALLY SLOW] Rendering 500 <SlowPost />
[ARTIFICIALLY SLOW] Rendering 500 <SlowPost />
```

- Displaying a pending visual state during the transition

```
function TabButton({ children, isActive, onClick }) {
  const [isPending, startTransition] = useTransition();
  // ...
  if (isPending) {
    return <b className="pending">{children}</b>;
  }
  // ...
```

App.js TabButton.js AboutTab.js PostsTab.js ContactTab.js Reset F

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

export default function TabButton({ children, isActive, onC
  const [isPending, startTransition] = useTransition();
  if (isActive) {
    return <b>{children}</b>
  }
  if (isPending) {
    return <b className="pending">{children}</b>;
  }
  return (
```

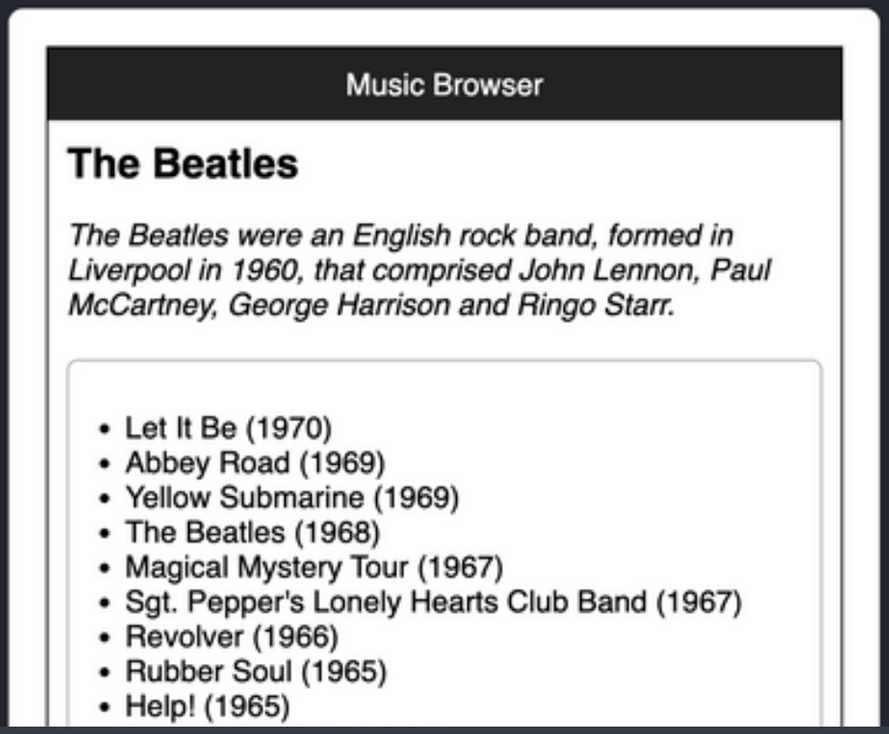
About Posts (slow) Contact

Welcome to my profile!

- Building a Suspense-enabled router

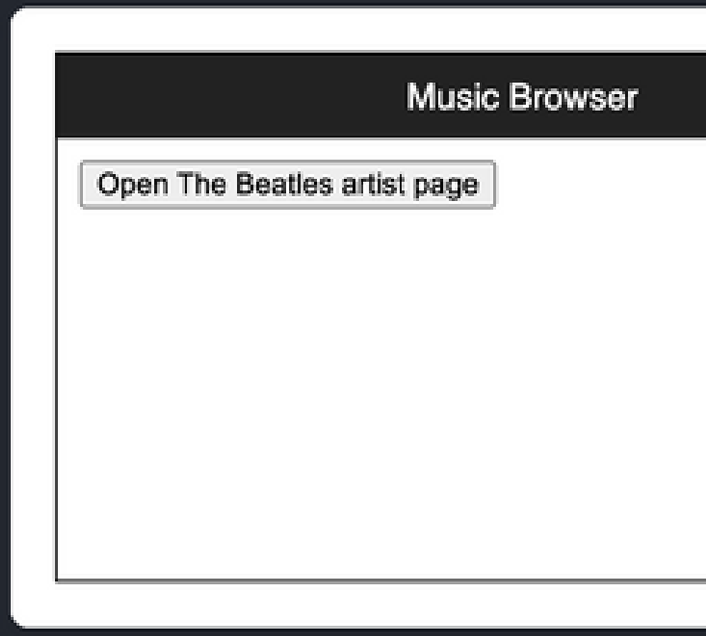
App.js Layout.js IndexPage.js ArtistPage.js

```
export default function App() {
  return (
    <Suspense fallback={<BigSpinner />}>
      <Router />
    </Suspense>
  );
}

function Router() {
  const [page, setPage] = useState('/');
  const [isPending, startTransition] = useTransition();

  function navigate(url) {
    startTransition(() => {
      setPage(url);
    });
```



App.js Layout.js IndexPage.js ArtistPage.js

```
export default function ArtistPage({ artist }) {
  return (
    <>
      <h1>{artist.name}</h1>
      <Biography artistId={artist.id} />
      <Suspense fallback={<AlbumsGlimmer />}>
        <Panel>
          <Albums artistId={artist.id} />
        </Panel>
      </Suspense>
    </>
  );
}

function AlbumsGlimmer() {
  return (
```



16. Custom Hooks: useWindowResize

```
export const useWindowResize = () => {
  const [width, setWidth] = useState(window.innerWidth);
  const [height, setHeight] = useState(window.innerHeight);

  const listener = () => {
    setWidth(window.innerWidth);
    setHeight(window.innerHeight);
  };

  useEffect(() => {
    window.addEventListener("resize", listener);
    return () => {
      window.removeEventListener("resize", listener);
    };
  }, []);

  return {
    width,
    height,
  };
};
example:
const windowSize = useWindowResize();
```

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

import reducer, { initialState } from "./reducer";
import useActions from "./actions";

const Store = React.createContext();

export function StoreProvider(props) {
  const { children } = props;

  const [state, dispatch] = useReducer(reducer, initialState);
  const actions = useActions(state, dispatch);
  const value = { state, dispatch, actions };

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

export const useStore = () => useContext(Store);
```

```
export default function useActions(_, dispatch) {
  const setTheme = (theme) => dispatch({ type: SET_THEME, payload: theme });

  const setCount = (count) => dispatch({ type: SET_COUNT, payload: count });

  const setVisible = (visible) => dispatch({ type: SET_VISIBLE, payload: visible });

  return {
    setTheme,
    setCount,
    setVisible
  };
}
```

```
const Header = ({ scrolled, history }) => {
  const {
    state: { theme, count },
    actions: { setTheme },
  } = useStore();
```

```
export default (reducer, actions, initialState) => {
  const Context = createContext();

  const Provider = ({children}) => {
    const [state, dispatch] = useReducer(reducer, initialState);

    const boundActions = {};
    for (let key in actions) {
      boundActions[key] = actions[key](dispatch);
    }

    return (
      <Context.Provider value={{state, ...boundActions}
        {children}
      </Context.Provider>
    );
  };

  return {Context, Provider};
};
```

```
export const { Context, Provider } = createDataContext(
  reducer,
  {
    fetchTexts,
    generatePDF,
    fetchAdminModalFields,
    checkLOA,
    checkStatus,
    verifyEmail,
    setUserDetails,
    fetchProducts,
    fetchOrders,
    setFormValues,
    fetchUserType,
    setAuthType,
    signIn,
    signUp,
    signOut,
    toggleModal,
    updateUser,
    checkAuth,
    fetchFormData,
    fetchTableData,
    setAuthType,
    deleteUser,
    setUserType,
    toggleLoading,
    checkVerification,
  },
  {
    texts: null,
```

```
const generatePDF = (dispatch) => async ({ child, file, type }) => {
  try {
    const email = Cookie.get("email");
    console.log(`${type}/${email}`, type, child);
    console.log(url);
    await firebase.storage().ref().child(child).put(file, "application/pdf")
    const url = await firebase
      .storage()
      .ref(`${type}/${email}`)
      .getDownloadURL();
    let obj = {};
    obj[type] = url;
    await db.collection("dev-users").doc(email).update(obj);
  } catch (e) {
    throw new Error(e);
  }
};
```

```
const {
  state: { country_code, texts, loading },
} = useContext(Context);
```

```
const fetchTexts = (dispatch) => async () => {
  try {
    const fetchUrl = "1acdf5a8-7165-40ab-a0e8-22917fd76808";
    const { data } = await Api(fetchUrl);
    generateTranslations(data, dispatch);
  } catch (e) {
    generateTranslations(mockTranslations, dispatch);
  }

  setCountryIP(dispatch);
};
```

```
const instance = axios.create({
  baseURL: "https://sheet.best/api/sheets/",
});

instance.interceptors.request.use(
  async (config) => {
      config.headers['Content-Type'] = 'application/json';
      return config;
  },
  (err) => {
    return Promise.reject(err);
  }
);

export default instance;
```

Built-in React Components

17. <Fragment> :

It is alternatively written as <>...</>, lets the user group multiple JSX nodes together.

<Fragment>, often used via <>...</> syntax, lets user group elements without a wrapper node.

```
<>
  <OneChild />
  <AnotherChild />
</>
```

Props

optional key: Fragments declared with the explicit <Fragment> syntax may have keys.

Caveats

If you want to pass key to a Fragment, you can’t use the <>...</> syntax. You have to explicitly import Fragment from 'react' and render <Fragment key={yourKey}>...</Fragment>.

React does not reset state when you go from rendering <><Child /></> to [<Child />] or back, or when you go from rendering <><Child /></> to <Child /> and back. This only works a single level deep: for example, going from <><><Child /></></> to <Child /> resets the state. See the precise semantics here.

Rendering a list of Fragments

Here's a situation where you need to write `Fragment` explicitly instead of using the `<></>` syntax. When you render multiple elements in a loop, you need to assign a `key` to each element. If the elements within the loop are Fragments, you need to use the normal JSX element syntax in order to provide the `key` attribute:

```
function Blog() {
  return posts.map(post =>
    <Fragment key={post.id}>
      <PostTitle title={post.title} />
      <PostBody body={post.body} />
    </Fragment>
  );
}
```

Assigning multiple elements to a variable

Like any other element, you can assign Fragment elements to variables, pass them as props, and so on:

```
function CloseDialog() {
  const buttons = (
    <>
      <OKButton />
      <CancelButton />
    </>
  );
  return (
    <AlertDialog buttons={buttons}>
      Are you sure you want to leave this page?
    </AlertDialog>
  );
}
```

18. <Profiler> :

<Profiler> lets user measure rendering performance of a React tree programmatically.

```
<Profiler id="App" onRender={onRender}>
  <App />
</Profiler>
```

- id: A string identifying the part of the UI you are measuring.
- onRender: An onRender callback that React calls every time components within the profiled tree update. It receives information about what was rendered and how much time it took.

```
function onRender(id, phase, actualDuration, baseDuration, startTime, commitTime) {
  // Aggregate or log render timings...
}

 <App>
   <Profiler id="Sidebar" onRender={onRender}>
     <Sidebar />
   </Profiler>
   <Profiler id="Content" onRender={onRender}>
     <Content>
       <Profiler id="Editor" onRender={onRender}>
         <Editor />
       </Profiler>
       <Preview />
     </Content>
   </Profiler>
 </App>
```

Although <Profiler> is a lightweight component, it should be used only when necessary. Each use adds some CPU and memory overhead to an application.

19. <StrictMode>

<StrictMode> lets user find common bugs in the components early during development.

```
<StrictMode>
  <App />
</StrictMode>
```

```
import { StrictMode } from 'react';
import { createRoot } from 'react-dom/client';

const root = createRoot(document.getElementById('root'));
root.render(
  <StrictMode>
    <App />
  </StrictMode>
);
```

Strict Mode enables the following development-only behaviors:

- Your components will re-render an extra time to find bugs caused by impure rendering.
- Your components will re-run Effects an extra time to find bugs caused by missing Effect cleanup.
- Your components will be checked for usage of deprecated APIs.

Usage:

- Enabling Strict Mode for entire app

```jsx
import { StrictMode } from 'react';
import { createRoot } from 'react-dom/client';

const root = createRoot(document.getElementById('root'));
root.render(
  <StrictMode>
    <App />
  </StrictMode>
);
```

Enabling strict mode for a part of the app

You can also enable Strict Mode for any part of your application:

```jsx
import { StrictMode } from 'react';

function App() {
  return (
    <>
      <Header />
      <StrictMode>
        <main>
          <Sidebar />
          <Content />
        </main>
      </StrictMode>
      <Footer />
    </>
  );
}
```

Fixing deprecation warnings enabled by Strict Mode

React warns if some component anywhere inside a `<StrictMode>` tree uses one of these deprecated APIs:

- `findDOMNode`. See alternatives.
- `UNSAFE_` class lifecycle methods like `UNSAFE_componentWillMount`. See alternatives.
- Legacy context (`childContextTypes`, `contextTypes`, and `getChildContext`). See alternatives.
- Legacy string refs (`this.refs`). See alternatives.

```jsx
export default function StoryTray({ stories }) {
  const items = stories;
  items.push({ id: 'create', label: 'Create Story' });
  return (
    <ul>
      {items.map(story => (
        <li key={story.id}>
          {story.label}
        </li>
      ))}
    </ul>
  );
}
```

Fixing bugs found by double rendering in development

index.js App.js StoryTray.js

```
export default function StoryTray({ stories }) {
  const items = stories;
  items.push({ id: 'create', label: 'Create Story' });
  return (
    <ul>
      {items.map(story => (
        <li key={story.id}>
          {story.label}
        </li>
      ))}
    </ul>
  );
}

```

Console (1)

```
Warning: Encountered two children with th
Keys should be unique so that components
identity across updates. Non-unique keys
be duplicated and/or omitted — the behavi
could change in a future version.
    at ul
```

Strict Mode *always* calls your rendering function twice, so you can see the mistake right away (“C
appears twice). This lets you notice such mistakes early in the process. When you fix your compone
in Strict Mode, you *also* fix many possible future production bugs like the hover functionality from

index.js App.js StoryTray.js

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

export default function StoryTray({ stories }) {
  const [isHover, setIsHover] = useState(false);
  const items = stories.slice(); // Clone the array
  items.push({ id: 'create', label: 'Create Story' });
  return (
    <ul
      onPointerEnter={() => setIsHover(true)}
      onPointerLeave={() => setIsHover(false)}
```

20. <Suspense>

<Suspense> lets the user display a fallback until its children have finished loading.

Usage:

Displaying a fallback while content is loading

```
<Suspense fallback={<Loading />}>
  <Albums />
</Suspense>
```

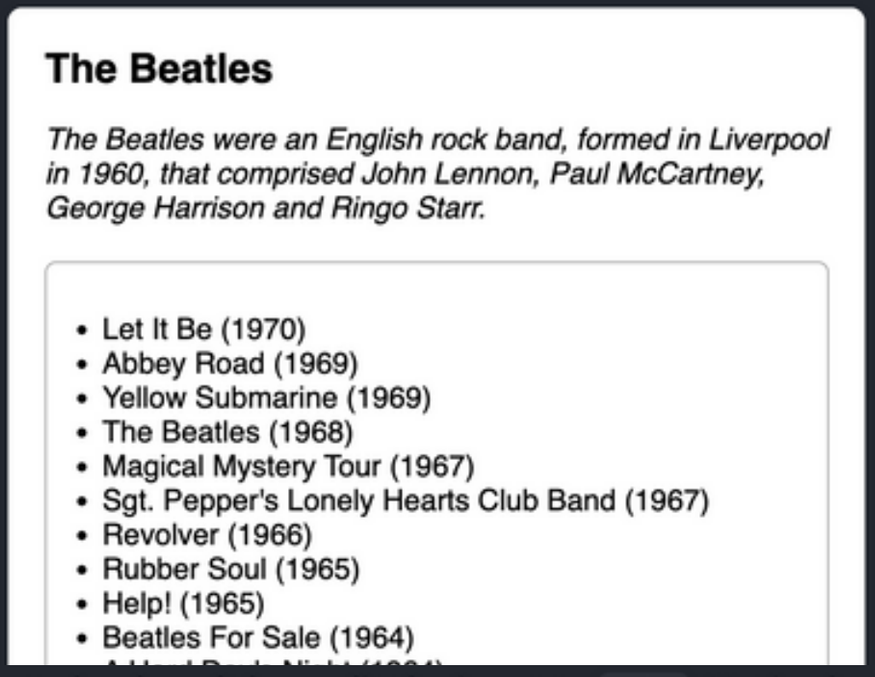
Only Suspense-enabled data sources will activate the Suspense component. They include:

- Data fetching with Suspense-enabled frameworks like Relay and Next.js
- Lazy-loading component code with lazy

```
export default function ArtistPage({ artist }) {
  return (
    <>
      <h1>{artist.name}</h1>
      <Suspense fallback={<Loading />}>
        <Biography artistId={artist.id} />
        <Panel>
          <Albums artistId={artist.id} />
        </Panel>
      </Suspense>
    </>
  );
}
```

Revealing nested content as it loads

```
export default function ArtistPage({ artist }) {
  return (
    <>
      <h1>{artist.name}</h1>
      <Suspense fallback={<BigSpinner />}>
        <Biography artistId={artist.id} />
        <Suspense fallback={<AlbumsGlimmer />}>
          <Panel>
            <Albums artistId={artist.id} />
          </Panel>
        </Suspense>
      </Suspense>
    </>
  );
}
```



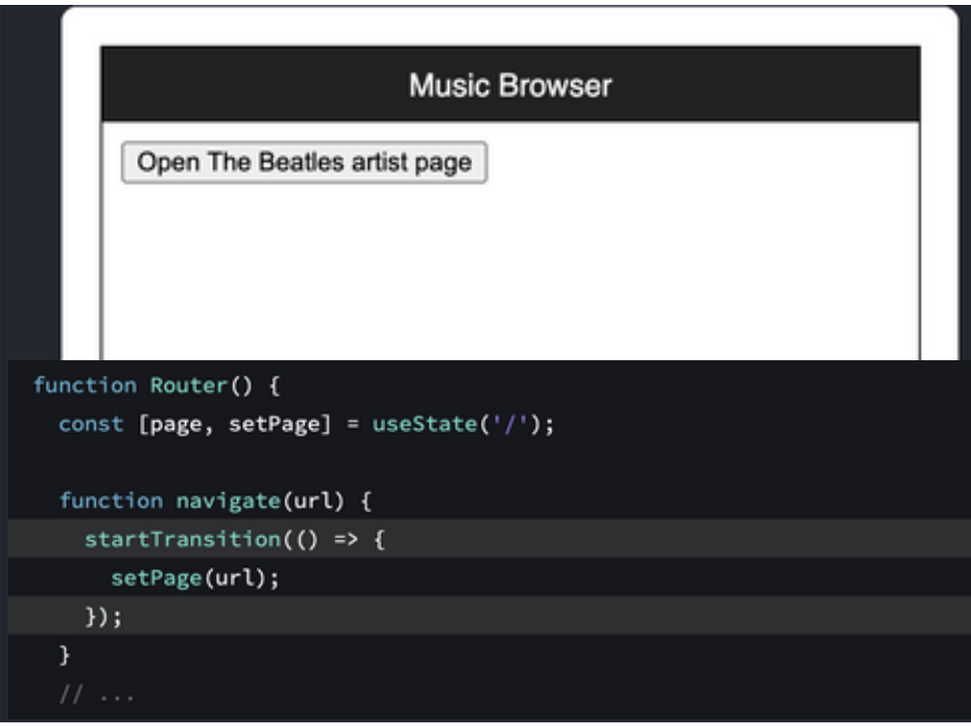
Enter `"a"` in the example below, wait for the results to load, and then edit the input to `"ab"`. Notice how instead of the Suspense fallback, you now see the dimmed stale result list until the new results have loade

App.js

```
export default function App() {
  const [query, setQuery] = useState('');
  const deferredQuery = useDeferredValue(query);
  const isStale = query !== deferredQuery;
  return (
    <>
      <label>
        Search albums:
        <input value={query} onChange={e => setQuery(e.targ
      </label>
      <Suspense fallback={<h2>Loading...</h2>}>
        <div style={{ opacity: isStale ? 0.5 : 1 }}>
          <SearchResults query={deferredQuery} />
        </div>
      </Suspense>
```

Preventing already revealed content from hiding:

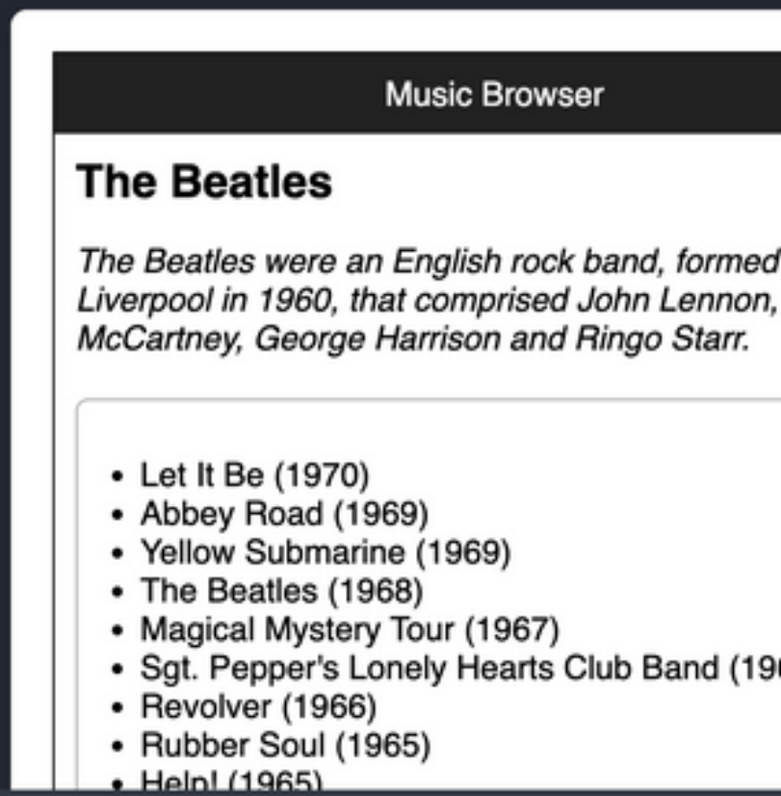
```jsx
export default function App() {
  return (
    <Suspense fallback={<BigSpinner />}>
      <Router />
    </Suspense>
  );
}

function Router() {
  const [page, setPage] = useState('/');

  function navigate(url) {
    startTransition(() => {
      setPage(url);
    });
```



- Indicating that a transition is happening

App.js Layout.js IndexPage.js ArtistPage.js

```jsx
7    return (
8      <Suspense fallback={<BigSpinner />}>
9        <Router />
10     </Suspense>
11   );
12 }
13
14 function Router() {
15   const [page, setPage] = useState('/');
16   const [isPending, startTransition] = useTransition();
17
18   function navigate(url) {
19     startTransition(() => {
20       setPage(url);
21     });
22   }
23
```



- Resetting Suspense boundaries on navigation

Resetting Suspense boundaries on navigation

During a transition, React will avoid hiding already revealed content. However, if you navigate to a route with different parameters, you might want to tell React it is *different* content. You can express this with a `key`:

```
<ProfilePage key={queryParams.id} />
```

```
<Suspense fallback={<Loading />}>
  <Chat />
</Suspense>

function Chat() {
  if (typeof window === 'undefined') {
    throw Error('Chat should only render on the client.');
  }
  // ...
}
```

Imagine you're navigating within a user's profile page, and something suspends. If that update is wrapped in a transition, it will not trigger the fallback for already visible content. That's the expected behavior.

However, now imagine you're navigating between two different user profiles. In that case, it makes sense to show the fallback. For example, one user's timeline is *different content* from another user's timeline. By specifying a `key`, you ensure that React treats different users' profiles as different components, and resets the Suspense boundaries during navigation. Suspense-integrated routers should do this automatically.

Providing a fallback for server errors and server-only content

If you use one of the streaming server rendering APIs (or a framework that relies on them), React will also use your `<Suspense>` boundaries to handle errors on the server. If a component throws an error on the server, React will not abort the server render. Instead, it will find the closest `<Suspense>` component above it and include its fallback (such as a spinner) into the generated server HTML. The user will see a spinner at first.

On the client, React will attempt to render the same component again. If it errors on the client too, React will throw the error and display the closest error boundary. However, if it does not error on the client, React will not display the error to the user since the content was eventually displayed successfully.

You can use this to opt out some components from rendering on the server. To do this, throw an error in the server environment and then wrap them in a `<Suspense>` boundary to replace their HTML with fallbacks:

21. createContext

It lets user create a context that components can provide or read.

```
const SomeContext = createContext(defaultValue)
```

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

const ThemeContext = createContext('light');

function App() {
  const [theme, setTheme] = useState('light');
  // ...
  return (
    <ThemeContext.Provider value={theme}>
      <Page />
    </ThemeContext.Provider>
  );
}

function Button() {
  // 🟡 Legacy way (not recommended)
  return (
    <ThemeContext.Consumer>
      {theme => (
        <button className={theme} />
      )}
    </ThemeContext.Consumer>
  );
}
function Button() {
  // ✅ Recommended way
  const theme = useContext(ThemeContext);
  return <button className={theme} />;
}
```

22. forwardRef

forwardRef lets the component expose a DOM node to parent component with a ref.

```
const SomeComponent = forwardRef(render)
```

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

const MyInput = forwardRef(function MyInput(props, ref) {
  const { label, ...otherProps } = props;
  return (
    <label>
      {label}
      <input {...otherProps} ref={ref} />
    </label>
  );
});

function Form() {
  const ref = useRef(null);

  function handleClick() {
    ref.current.focus();
  }

  return (
    <form>
      <MyInput label="Enter your name:" ref={ref} />
      <button type="button" onClick={handleClick}>
        Edit
      </button>
    </form>
  );
}
```

Exposing an imperative handle instead of a DOM node

Instead of exposing an entire DOM node, you can expose a custom object, called an *imperative handle*, with a more constrained set of methods. To do this, you'd need to define a separate ref to hold the DOM node:

```
const MyInput = forwardRef(function MyInput(props, ref) {
  const inputRef = useRef(null);

  // ...

  return <input {...props} ref={inputRef} />;
});
```

Pass the `ref` you received to `useImperativeHandle` and specify the value you want to expose to the `ref`:

```
import { forwardRef, useRef, useImperativeHandle } from 'react';

const MyInput = forwardRef(function MyInput(props, ref) {
  const inputRef = useRef(null);

  useImperativeHandle(ref, () => {
    return {
      focus() {
        inputRef.current.focus();
      },
      scrollIntoView() {
        inputRef.current.scrollIntoView();
      },
    };
  }, []);

  return <input {...props} ref={inputRef} />;
```

23. lazy

lazy lets user defer loading component's code until it is rendered for the first time.

```
const SomeComponent = lazy(load)
```

```
import MarkdownPreview from './MarkdownPreview.js';

import { lazy } from 'react';

const MarkdownPreview = lazy(() => import('./MarkdownPreview.js'));
```

```
<Suspense fallback={<Loading />}>
  <h2>Preview</h2>
  <MarkdownPreview />
</Suspense>
```

App.js Loading.js MarkdownPreview.js Reset Fork

```
3
4  const MarkdownPreview = lazy(() =>
5    delayForDemo(import('./MarkdownPreview.js')));
6
7  export default function MarkdownEditor() {
8    const [showPreview, setShowPreview] = useState(false);
9    const [markdown, setMarkdown] = useState('Hello, **world*
10   return (
11     <>
12       <textarea value={markdown} onChange={e => setMarkdown
13       <label>
14         <input type="checkbox" checked={showPreview} onChan
15         Show preview
16       </label>
17       <hr />
18       {showPreview && (
19         <Suspense fallback={<Loading />}>
```


```
function delayForDemo(promise) {
  return new Promise(resolve => {
    setTimeout(resolve, 2000);
  }).then(() => promise);
```

24. memo

memo lets the user skip re-rendering a component when its props are unchanged.

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

const SomeComponent = memo(function SomeComponent(props) {
  // ...
});
```

Usage:

- Skipping re-rendering when props are unchanged

```
const Greeting = memo(function Greeting({ name }) {
  return <h1>Hello, {name}!</h1>;
});

export default Greeting;
```

- Updating a memoized component using state

```
const Greeting = memo(function Greeting({ name }) {
  console.log('Greeting was rendered at',
              new Date().toLocaleTimeString());
  const [greeting, setGreeting] = useState('Hello');
  return (
    <>
      <h3>{greeting}{name && ', '}{name}!</h3>
      <GreetingSelector value={greeting}
        onChange={setGreeting} />
    </>
  );
});
```

- Minimizing props changes

```
function Page() {
  const [name, setName] = useState('Taylor');
  const [age, setAge] = useState(42);

  const person = useMemo(
    () => ({ name, age }),
    [name, age]
  );

  return <Profile person={person} />;
}

const Profile = memo(function Profile({ person }) {
  // ...
});
```

Specifying a custom comparison function

In rare cases it may be infeasible to minimize the props changes of a memoized component. In that case, you can provide a custom comparison function, which React will use to compare the old and new props instead of using shallow equality. This function is passed as a second argument to memo. It should return true only if the new props would result in the same output as the old props; otherwise it should return false.

```
const Chart = memo(function Chart({ dataPoints }) {
  // ...
}, arePropsEqual);

function arePropsEqual(oldProps, newProps) {
  return (
    oldProps.dataPoints.length === newProps.dataPoints.length &&
    oldProps.dataPoints.every((oldPoint, index) => {
      const newPoint = newProps.dataPoints[index];
      return oldPoint.x === newPoint.x && oldPoint.y === newPoint.y;
    })
  );
}
```

If you do this, use the Performance panel in your browser developer tools to make sure that your comparison function is actually faster than re-rendering the component.

When you do performance measurements, make sure that React is running in the production mode.

25. startTransition

startTransition lets the user update the state without blocking the UI.

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

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

  function selectTab(nextTab) {
    startTransition(() => {
      setTab(nextTab);
    });
  }
  // ...
}
```

startTransition is very similar to useTransition, except that it does not provide the isPending flag to track whether a transition is ongoing. You can call startTransition when useTransition is not available. For example, startTransition works outside components, such as from a data library.

Usage

- Marking a state update as a non-blocking transition

```
function TabContainer() {
  const [tab, setTab] = useState('about');

  function selectTab(nextTab) {
    startTransition(() => {
      setTab(nextTab);
    });
  }
  // ...
}
```

Transitions let you keep the user interface updates responsive even on slow devices.

With a transition, your UI stays responsive in the middle of a re-render. For example, if the user clicks a tab but then change their mind and click another tab, they can do that without waiting for the first re-render to finish.

26. createPortal

createPortal lets the user render some children into a different part of the DOM.

```
import { createPortal } from 'react-dom';

// ...

<div>
  <p>This child is placed in the parent div.</p>
  {createPortal(
    <p>This child is placed in the document body.</p>,
    document.body
  )}
</div>
```

A portal only changes the physical placement of the DOM node. In every other way, the JSX you render into a portal acts as a child node of the React component that renders it. For example, the child can access the context provided by the parent tree, and events bubble up from children to parents according to the React tree.

Usage

- Rendering to a different part of the DOM

```
  return (
    <div style={{ border: '2px solid black' }}>
      <p>This child is placed in the parent div.</p>
      {createPortal(
        <p>This child is placed in the document body.</p>,
        document.body
      )}
    </div>
```

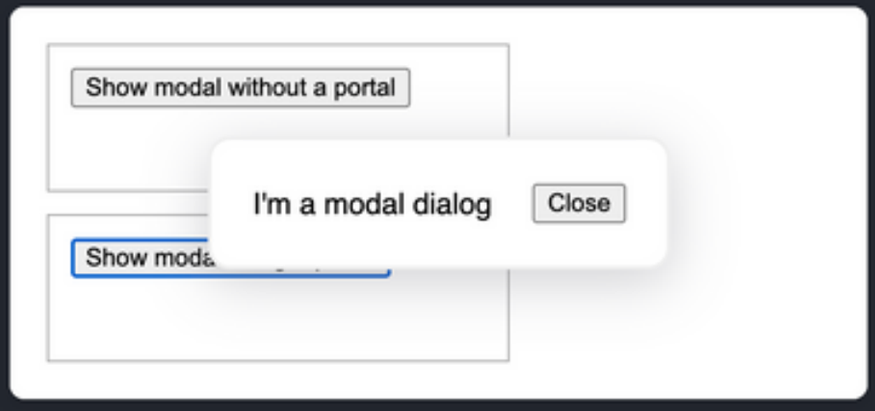
Rendering a modal dialog with a portal

You can use a portal to create a modal dialog that floats above the rest of the page, even if the component that summons the dialog is inside a container with `overflow: hidden` or other styles that interfere with the dialog.

In this example, the two containers have styles that disrupt the modal dialog, but the one rendered into a portal is unaffected because, in the DOM, the modal is not contained within the parent JSX elements.

App.js NoPortalExample.js PortalExample.js ModalContent.js Reset Fork

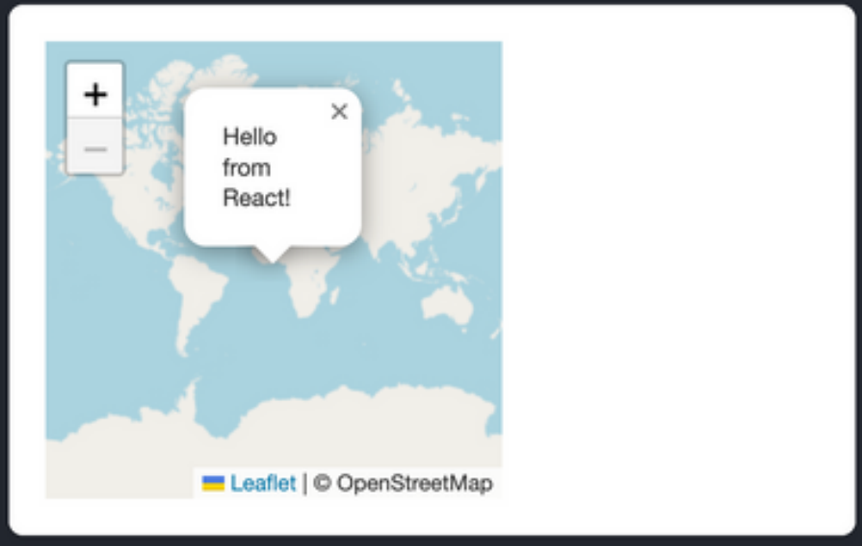
```
4
5  export default function PortalExample() {
6    const [showModal, setShowModal] = useState(false);
7    return (
8      <>
9        <button onClick={() => setShowModal(true)}>
10         Show modal using a portal
11       </button>
12       {showModal && createPortal(
13         <ModalContent onClose={() => setShowModal(false)} />
14         document.body
15       )}
16     </>
17   );
18 }
19
```



- Rendering React components into non-React DOM nodes

App.js map-widget.js Reset Fork

```
13       mapRef.current = map;
14       const popupDiv = addPopupToMapWidget(map);
15       setPopupContainer(popupDiv);
16     }
17   }, []);
18
19   return (
20     <div style={{ width: 250, height: 250 }}
21       ref={containerRef}>
22       {popupContainer !== null && createPortal(
23         <p>Hello from React!</p>,
24         popupContainer
25       )}
26     </div>
```



27. flushSync

flushSync lets the user force React to flush any updates inside the provided callback synchronously. This ensures that the DOM is updated immediately.

```
flushSync(callback)
```

```
import { flushSync } from 'react-dom';

flushSync(() => {
  setSomething(123);
});
```

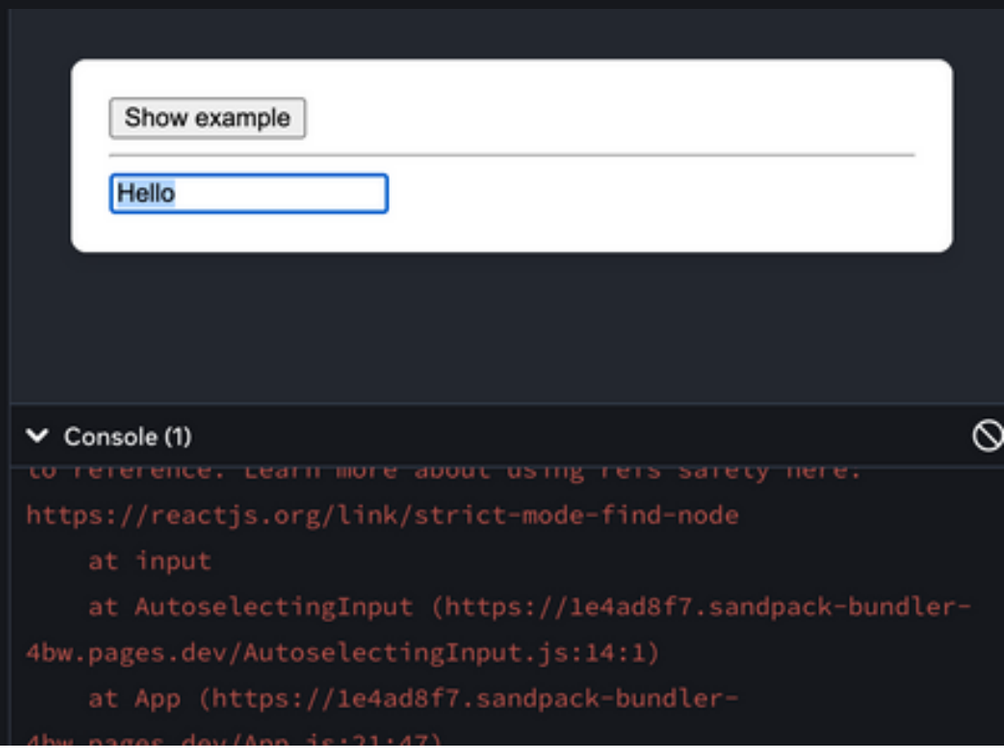
```
    const [isPrinting, setIsPrinting] = useState(false);

    useEffect(() => {
      function handleBeforePrint() {
        flushSync(() => {
          setIsPrinting(true);
        })
      }

      function handleAfterPrint() {
        setIsPrinting(false);
      }

      window.addEventListener('beforeprint', handleBeforePrin
      window.addEventListener('afterprint', handleAfterPrint)
      return () => {
        window.removeEventListener('beforeprint', handleBefor
      window.removeEventListener('afterprint',
                                 handleAfterPrint);
    }
  }, []);

  return (
    <>
      <h1>isPrinting: {isPrinting ? 'yes' : 'no'}</h1>
      <button onClick={() => window.print()}>
        Print
```

28. findDOMNode

findDOMNode finds the browser DOM node for a React class component instance.

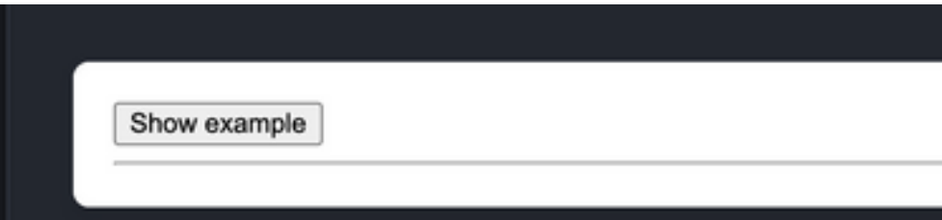
```
const domNode = findDOMNode(componentInstance)
```

```
import { Component } from 'react';
import { findDOMNode } from 'react-dom';

class AutoselectingInput extends Component {
  componentDidMount() {
    const input = findDOMNode(this);
    input.select()
  }

  render() {
    return <input defaultValue="Hello" />
  }
}

export default AutoselectingInput;
```



- Reading component's own DOM node from a ref

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

export default function AutoselectingInput() {
  const inputRef = useRef(null);

  useEffect(() => {
    const input = inputRef.current;
    input.select();
  }, []);

  return <input ref={inputRef} defaultValue="Hello" />
}
```



29. hydrate

hydrate lets the user display React components inside a browser DOM node whose HTML content was previously generated by react-dom/server in React 17 and below.

```
hydrate(reactNode, domNode, callback?)
```

Hydrating server-rendered HTML

In React, “hydration” is how React “attaches” to existing HTML that was already rendered by React in a server environment. During hydration, React will attempt to attach event listeners to the existing markup and take over rendering the app on the client.

In apps fully built with React, **you will usually only hydrate one “root”, once at startup for your entire app.**

index.html index.js App.js Reset

```
1  import './styles.css';
2  import { hydrate } from 'react-dom';
3  import App from './App.js';
4
5  hydrate(<App />, document.getElementById('root'));
6
```

Hello, world!

Console (1)

```
Warning: ReactDOM.hydrate is no longer supported in React
Use hydrateRoot instead. Until you switch to the new API,
app will behave as if it's running React 17. Learn more:
```

index.html index.js App.js

```
1  import './styles.css';
2  import { hydrate } from 'react-dom';
3  import App from './App.js';
4
5  hydrate(<App />, document.getElementById('root'));
6
```

Hello, world!

Console (1)

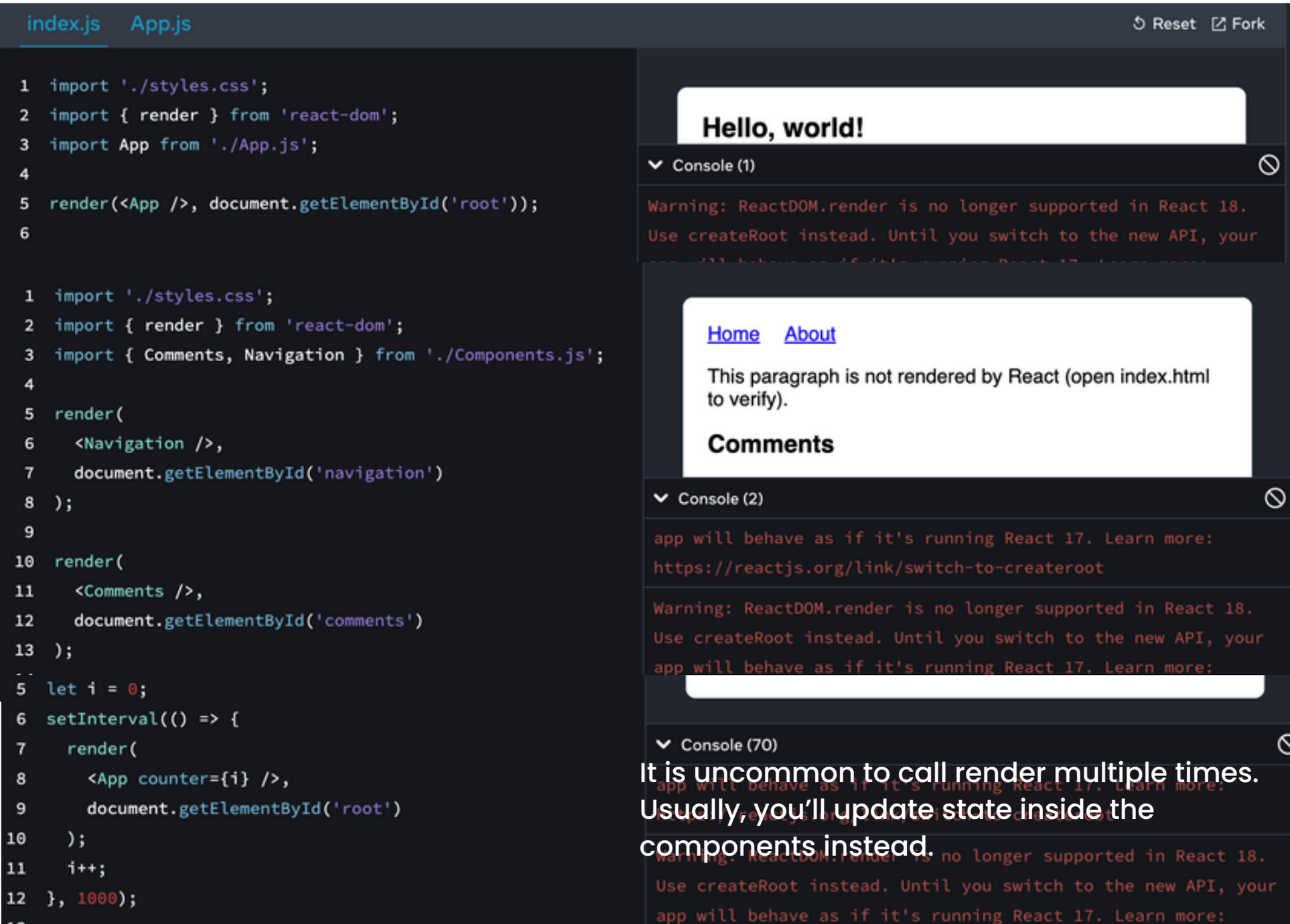
```
Warning: ReactDOM.hydrate is no longer su
Use hydrateRoot instead. Until you switch
app will behave as if it's running React
https://reactjs.org/link/switch-to-create
```

30. render

render renders a piece of JSX (“React node”) into a browser DOM node.

```
render(reactNode, domNode, callback?)
```

In React 18, render was replaced by createRoot. Using render in React 18 will warn that your app will behave as if it’s running React 17.



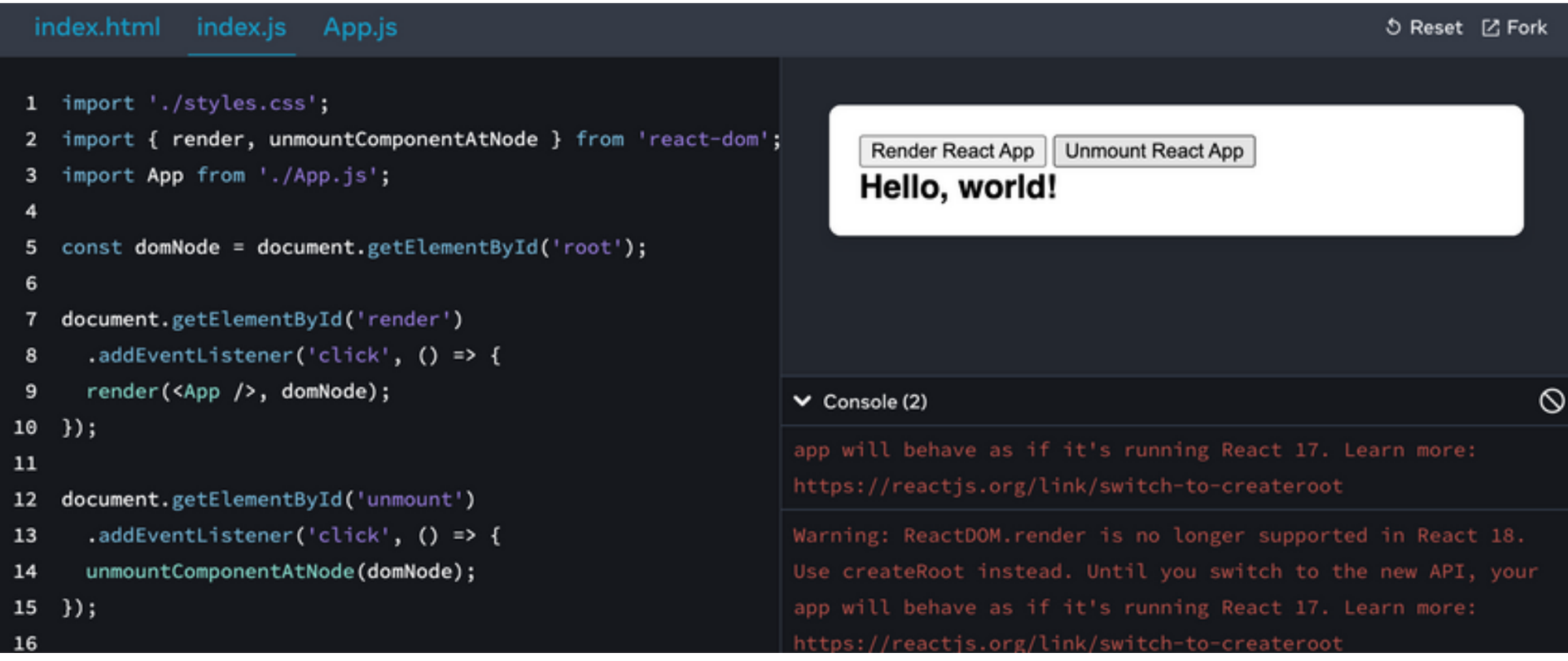
It is uncommon to call render multiple times. Usually, you’ll update state inside the components instead.

31. unmountComponentAtNode

In React 18, unmountComponentAtNode was replaced by root.unmount().

```
import { unmountComponentAtNode } from 'react-dom';

const domNode = document.getElementById('root');
render(<App />, domNode);

unmountComponentAtNode(domNode);
```

- Removing a React app from a DOM element



Client React DOM APIs

32. createRoot

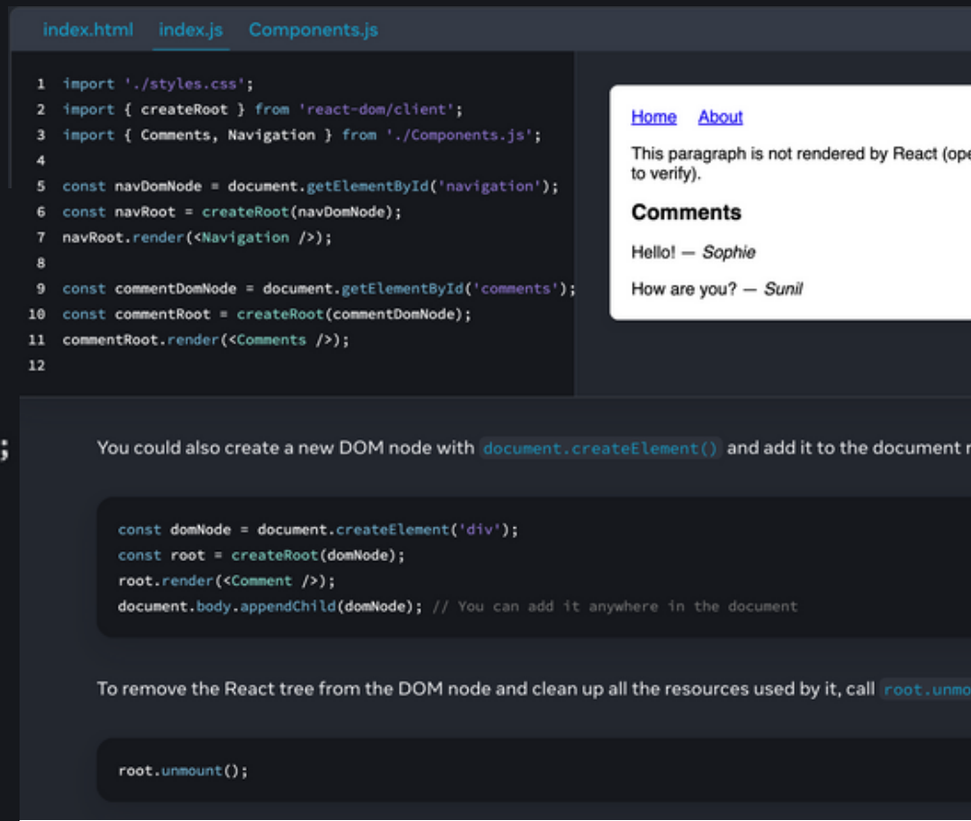
The react-dom/client APIs let the user render React components on the client (in the browser). These APIs are typically used at the top level of the app to initialize your React tree. A framework may call them for the user. Most of the components don’t need to import or use them.

Browser support

React supports all popular browsers, including Internet Explorer 9 and above. Some polyfills are required for older browsers such as IE 9 and IE 10.

```
import { createRoot } from 'react-dom/client';

const domNode = document.getElementById('root');
const root = createRoot(domNode);

root.render(<App />);
```

```
import { createRoot } from 'react-dom/client';
import './styles.css';
import App from './App.js';

const root = createRoot(document.getElementById('root'));

let i = 0;
setInterval(() => {
  root.render(<App counter={i} />);
  i++;
}, 1000);
```



33. hydrateRoot

hydrateRoot lets user display React components inside a browser DOM node whose HTML content was previously generated by react-dom/server.

```
const root = hydrateRoot(domNode, reactNode, options?)
```

```
import { hydrateRoot } from 'react-dom/client';

const domNode = document.getElementById('root');
const root = hydrateRoot(domNode, reactNode);
```

```
import './styles.css';
import { hydrateRoot } from 'react-dom/client';
import App from './App.js';

hydrateRoot(document.getElementById('root'), <App />);
```

```
export default function App() {
  return (
    <h1 suppressHydrationWarning={true}>
      Current Date: {new Date().toLocaleDateString()}
    </h1>
  );
}
```

Current Date: 06/05/2023

Current Date: 06/05/2023

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

export default function App() {
  const [isClient, setIsClient] = useState(false);

  useEffect(() => {
    setIsClient(true);
  }, []);

  return (
    <h1>
      {isClient ? 'Is Client' : 'Is Server'}
```

Is Client

Server React DOM APIs

The react-dom/server APIs let user render React components to HTML on the server. These APIs are only used on the server at the top level of your app to generate the initial HTML. A framework may call them for you. Most of your components don't need to import or use them.

Server APIs for Node.js Streams

34. renderToNodeStream

renderToNodeStream renders a React tree to a Node.js Readable Stream.

```
 const stream = renderToNodeStream(reactNode)

import { renderToNodeStream } from 'react-dom/server';

const stream = renderToNodeStream(<App />);
stream.pipe(response);

 import { renderToNodeStream } from 'react-dom/server';

 // The route handler syntax depends on your backend framework
 app.use('/', (request, response) => {
   const stream = renderToNodeStream(<App />);
   stream.pipe(response);
 });
```

The stream will produce the initial non-interactive HTML output of your React components. On the client, you will need to call hydrateRoot to hydrate that server-generated HTML and make it interactive.

35. renderToPipeableStream

renderToPipeableStream renders a React tree to a pipeable Node.js Stream.

```
const { pipe, abort } = renderToPipeableStream(reactNode, options?)
```

```
import { renderToPipeableStream } from 'react-dom/server';

const { pipe } = renderToPipeableStream(<App />, {
  bootstrapScripts: ['/main.js'],
  onShellReady() {
    response.setHeader('content-type', 'text/html');
    pipe(response);
  }
});
```

```
import { renderToPipeableStream } from 'react-dom/server';

// The route handler syntax depends on your backend framework
app.use('/', (request, response) => {
  const { pipe } = renderToPipeableStream(<App />, {
    bootstrapScripts: ['/main.js'],
    onShellReady() {
      response.setHeader('content-type', 'text/html');
      pipe(response);
    }
  });
});
```

```
<!DOCTYPE html>
<html>
  <!-- ... HTML from your components ... -->
</html>
<script src="/main.js" async=""></script>
```

Server APIs for Web Streams

36. renderToReadableStream

renderToReadableStream renders a React tree to a Readable Web Stream.

```
import { renderToReadableStream } from 'react-dom/server';

async function handler(request) {
  const stream = await renderToReadableStream(<App />, {
    bootstrapScripts: ['/main.js']
  });
  return new Response(stream, {
    headers: { 'content-type': 'text/html' },
  });
}
```

- Streaming more content as it loads

Streaming allows the user to start seeing the content even before all the data has loaded on the server.

```
function ProfilePage() {
  return (
    <ProfileLayout>
      <ProfileCover />
      <Sidebar>
        <Friends />
        <Photos />
      </Sidebar>
      <Posts />
    </ProfileLayout>
  );
}
```

Server APIs for non-streaming environments

37. renderToStaticMarkup

renderToStaticMarkup renders a non-interactive React tree to an HTML string.

```
const html = renderToStaticMarkup(reactNode)
```

```
import { renderToStaticMarkup } from 'react-dom/server';

const html = renderToStaticMarkup(<Page />);
```

Rendering a non-interactive React tree as HTML to a string

```
import { renderToStaticMarkup } from 'react-dom/server';

// The route handler syntax depends on your backend framework
app.use('/', (request, response) => {
  const html = renderToStaticMarkup(<Page />);
  response.send(html);
});
```

This will produce the initial non-interactive HTML output of your React components.

38. renderToStaticNodeStream

renderToStaticNodeStream renders a non-interactive React tree to a Node.js Readable Stream.

```
const stream = renderToStaticNodeStream(reactNode)

import { renderToStaticNodeStream } from 'react-dom/server';

const stream = renderToStaticNodeStream(<Page />);
stream.pipe(response);
import { renderToStaticNodeStream } from 'react-dom/server';

// The route handler syntax depends on your backend framework
app.use('/', (request, response) => {
  const stream = renderToStaticNodeStream(<Page />);
  stream.pipe(response);
});
```

The stream will produce the initial non-interactive HTML output of the React components.

39. renderToString

renderToString renders a React tree to an HTML string.

```
const html = renderToString(reactNode)
```

```
import { renderToString } from 'react-dom/server';

const html = renderToString(<App />);
```

```
import { renderToString } from 'react-dom/server';

// The route handler syntax depends on your backend framework
app.use('/', (request, response) => {
  const html = renderToString(<App />);
  response.send(html);
});
```

Removing `renderToString` from the client code

Sometimes, `renderToString` is used on the client to convert some component to HTML.

```
// ▶ Unnecessary: using renderToString on the client
import { renderToString } from 'react-dom/server';

const html = renderToString(<MyIcon />);
console.log(html); // For example, "<svg>...</svg>"
```

Importing `react-dom/server` **on the client** unnecessarily increases your bundle size and should be avoided. If you need to render some component to HTML in the browser, use `createRoot` and read HTML from the DOM:

```
import { createRoot } from 'react-dom/client';
import { flushSync } from 'react-dom';

const div = document.createElement('div');
const root = createRoot(div);
flushSync(() => {
  root.render(<MyIcon />);
});
console.log(div.innerHTML); // For example, "<svg>...</svg>"
```

The `flushSync` call is necessary so that the DOM is updated before reading its `innerHTML` property.

Legacy React APIs

These APIs are exported from the react package, but they are not recommended for use in newly written code.

40. Children

Children lets us manipulate and transform the JSX user received as the children prop.

```
const mappedChildren = Children.map(children, child =>
  <div className="Row">
    {child}
  </div>
);

import { Children } from 'react';

function RowList({ children }) {
  return (
    <>
      <h1>Total rows: {Children.count(children)}</h1>
      ...
    </>
  );
}
```

```
function RowList({ children }) {
  return (
    <div className="RowList">
      {Children.map(children, child =>
        <div className="Row">
          {child}
        </div>
      )}
```

41. cloneElement

cloneElement lets user create a new React element using another element as a starting point.

```
const clonedElement = cloneElement(element, props, ...children)
```

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

// ...
const clonedElement = cloneElement(
  <Row title="Cabbage">
    Hello
  </Row>,
  { isHighlighted: true },
  'Goodbye'
);

console.log(clonedElement); // <Row title="Cabbage">Goodbye</Row>
```

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

 // ...
 const clonedElement = cloneElement(
   <Row title="Cabbage" />,
   { isHighlighted: true }
 );

export default function List({ children }) {
  const [selectedIndex, setSelectedIndex] = useState(0);
  return (
    <div className="List">
      {Children.map(children, (child, index) =>
        cloneElement(child, {
          isHighlighted: index === selectedIndex
        })
      )}
```

42. Component

Component is the base class for the React components defined as JavaScript classes. Class components are still supported by React, but we don't recommend using them in new code.

```
class Greeting extends Component {
  render() {
    return <h1>Hello, {this.props.name}!</h1>;
  }
}
```

```
class Counter extends Component {
  constructor(props) {
    super(props);
    this.state = { counter: 0 };
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    // ...
  }
```

f you use modern JavaScript syntax, constructors are rarely needed. Instead, you can rewrite this code above
sing the public class field syntax which is supported both by modern browsers and tools like Babel:

```
class Counter extends Component {
  state = { counter: 0 };

  handleClick = () => {
    // ...
  }
```

43. createElement

createElement lets user create a React element. It serves as an alternative to writing JSX.

```
const element = createElement(type, props, ...children)
```

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

function Greeting({ name }) {
  return createElement(
    'h1',
    { className: 'greeting' },
    'Hello'
  );
}
```

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

function Greeting({ name }) {
  return createElement(
    'h1',
    { className: 'greeting' },
    'Hello ',
    createElement('i', null, name),
    '. Welcome!'
```

App.js

Download Reset Fork

```
1  import { createElement } from 'react';
2
3  function Greeting({ name }) {
4    return createElement(
5      'h1',
6      { className: 'greeting' },
7      'Hello ',
8      createElement('i', null, name),
9      '. Welcome!'
10   );
```

Hello *Taylor*. Welcome!

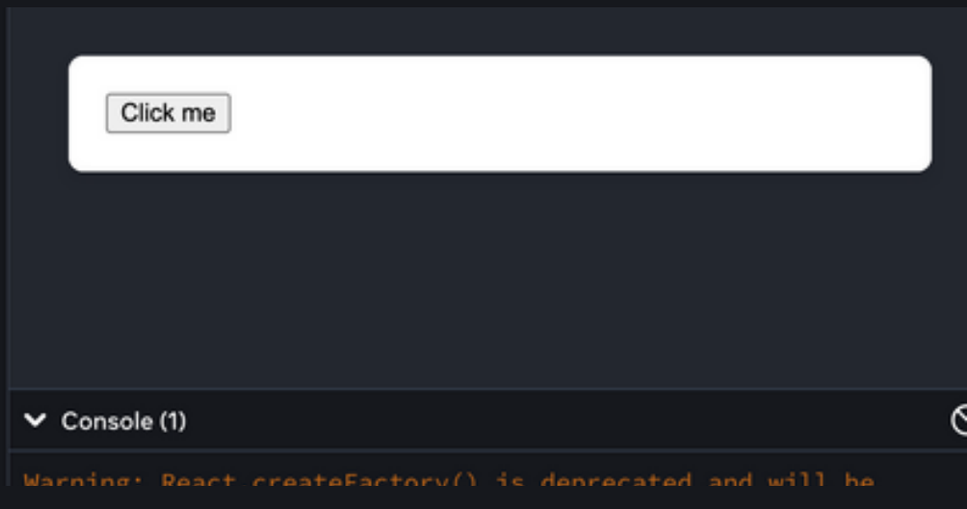
44. createFactory

createFactory lets user create a function that produces React elements of a given type.

```
const factory = createFactory(type)
```

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

const button = createFactory('button');

export default function App() {
  return button({
    onClick: () => {
      alert('Clicked!')
    }
  }, 'Click me');
}
```

```
1  import { createFactory } from 'react';
2
3  const button = createFactory('button');
4
5  export default function App() {
6    return button({
7      onClick: () => {
8        alert('Clicked!')
9      }
10   }, 'Click me');
```



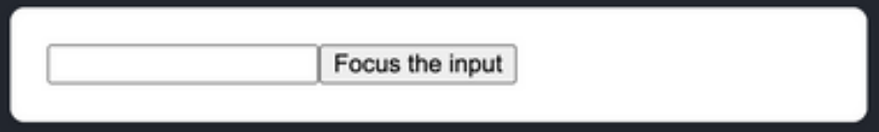

```
import { createElement } from 'react';

export default function App() {
  return createElement('button', {
    onClick: () => {
      alert('Clicked!')
    }
  }, 'Click me');
}
```

45. createRef

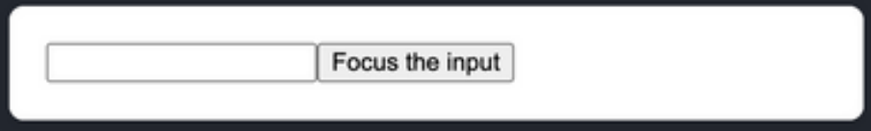
createRef creates a ref object which can contain arbitrary value.

```
class MyInput extends Component {
  inputRef = createRef();
  // ...
}
```

```
export default class Form extends Component {
  inputRef = createRef();

  handleClick = () => {
    this.inputRef.current.focus();
  }

  render() {
    return (
      <>
        <input ref={this.inputRef} />
        <button onClick={this.handleClick}>
          Focus the input
        </button>
      </>
    );
  }
```




```
export default function Form() {
  const inputRef = useRef(null);

  function handleClick() {
    inputRef.current.focus();
  }

  return (
    <>
      <input ref={inputRef} />
      <button onClick={handleClick}>
        Focus the input
      </button>
    </>
  );
}
```



46. isValidElement

isValidElement checks whether a value is a React element.

```
const isElement = isValidElement(value)
```

```
import { isValidElement, createElement } from 'react';

// ✅ React elements
console.log(isValidElement(<p />)); // true
console.log(isValidElement(createElement('p'))); // true

// ❌ Not React elements
console.log(isValidElement(25)); // false
console.log(isValidElement('Hello')); // false
console.log(isValidElement({ age: 42 })); // false
```

```
import { isValidElement, createElement } from 'react';

// ✅ JSX tags are React elements
console.log(isValidElement(<p />)); // true
console.log(isValidElement(<MyComponent />)); // true

// ✅ Values returned by createElement are React elements
console.log(isValidElement(createElement('p'))); // true
console.log(isValidElement(createElement(MyComponent))); // true
```

Any other values, such as strings, numbers, or arbitrary objects and arrays, are not React elements.

For them, `isValidElement` returns `false`:

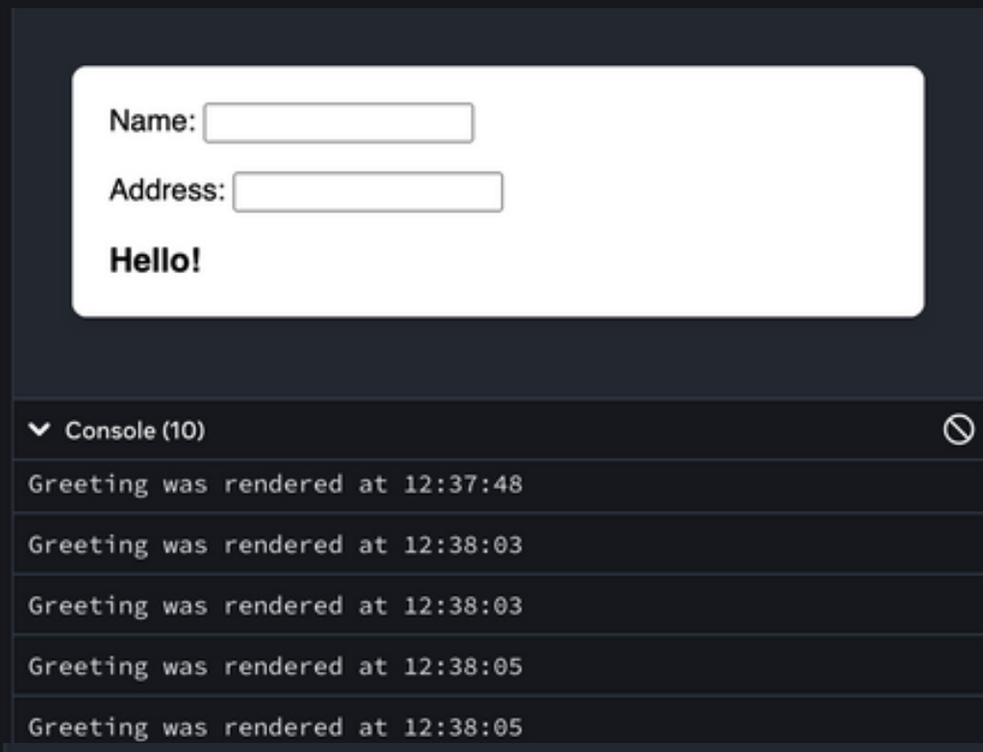
```
// ❌ These are *not* React elements
console.log(isValidElement(null)); // false
console.log(isValidElement(25)); // false
```

47. PureComponent

PureComponent is similar to Component but it skips re-renders for same props and state. Class components are still supported by React, but we don't recommend using them in new code.

```jsx
class Greeting extends PureComponent {
  render() {
    return <h1>Hello, {this.props.name}!</h1>;
  }
}
```

```jsx
class Greeting extends PureComponent {
  render() {
    console.log("Greeting was rendered at",
                new Date().toLocaleTimeString());
    return <h3>Hello{this.props.name && ', '}
      {this.props.name}!</h3>;
  }
}

export default function MyApp() {
  const [name, setName] = useState('');
  const [address, setAddress] = useState('');
  return (
    <>
      <label>
        Name{': '}
        <input value={name} onChange={e =>
```




```jsx
const Greeting = memo(function Greeting({ name }) {
  console.log("Greeting was rendered at", new Date().toLoca
  return <h3>Hello{name && ', '}{name}!</h3>;
});

export default function MyApp() {
  const [name, setName] = useState('');
  const [address, setAddress] = useState('');
  return (
    <>
      <label>
```

48. Throttle & Debounce

```html
<html>
  <body>
    <label>Search Here : </label>
    <input type="text" id="search-bar" />
  </body>
</html>
```

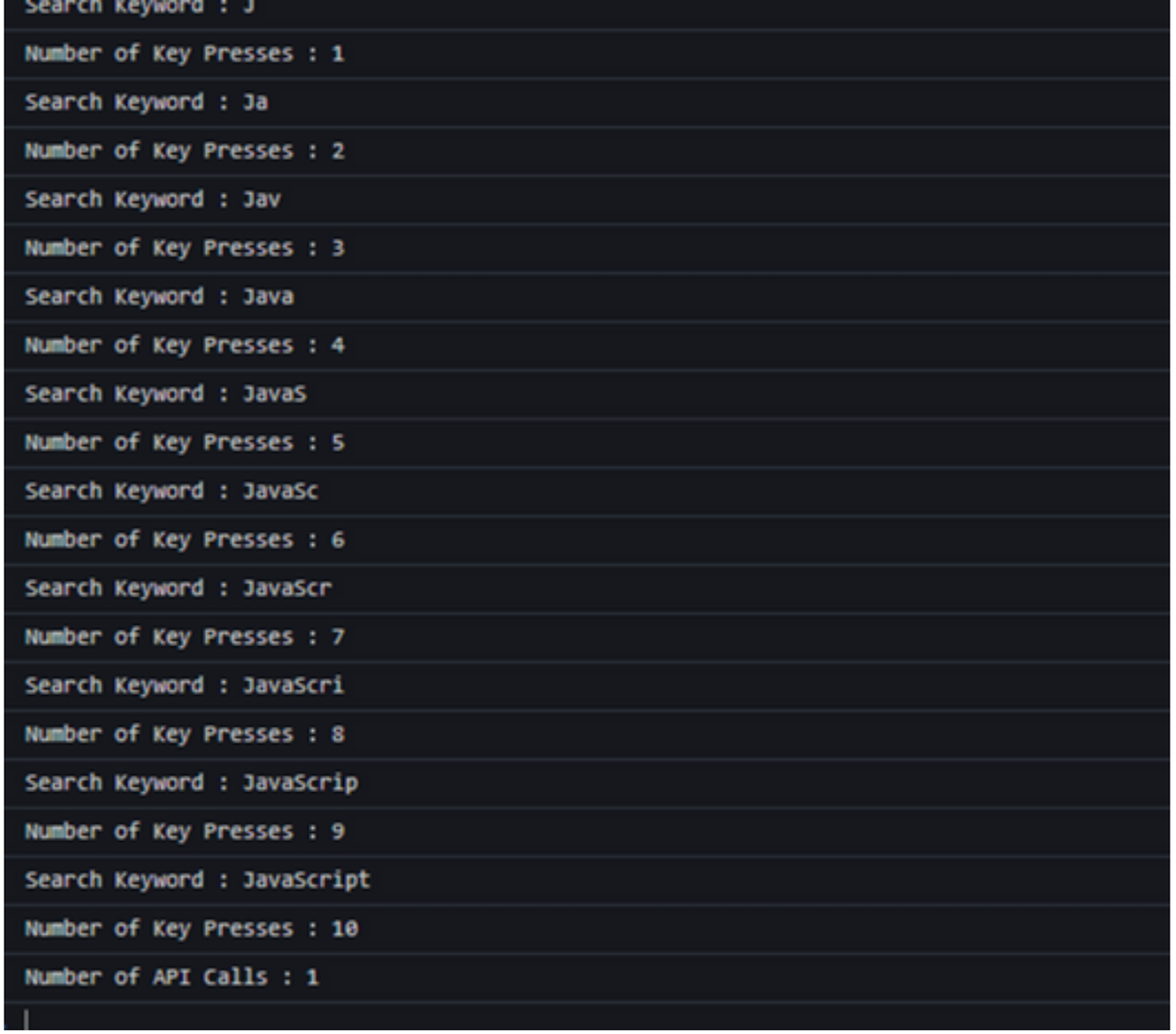
```
var searchBarDom = document.getElementById('search-bar');
var numberOfKeyPresses = 0;
var numberOfApiCalls = 0;

function getSearchResult() {
  numberOfApiCalls += 1;
  console.log('Number of API Calls : ' + numberOfApiCalls);
}

searchBarDom.addEventListener('input', function (e) {
  numberOfKeyPresses += 1;
  console.log('Search Keyword : ' + e.target.value);
  console.log('Number of Key Presses : ' + numberOfKeyPresses );
  getSearchResult();
});
```

```
Search Keyword : J
Number of Key Presses : 1
Number of API Calls : 1
Search Keyword : Ja
Number of Key Presses : 2
Number of API Calls : 2
Search Keyword : Jav
Number of Key Presses : 3
Number of API Calls : 3
Search Keyword : Java
Number of Key Presses : 4
Number of API Calls : 4
Search Keyword : JavaS
Number of Key Presses : 5
Number of API Calls : 5
Search Keyword : JavaSc
Number of Key Presses : 6
Number of API Calls : 6
Search Keyword : JavaScr
Number of Key Presses : 7
Number of API Calls : 7
Search Keyword : JavaScri
Number of Key Presses : 8
Number of API Calls : 8
Search Keyword : JavaScrip
Number of Key Presses : 9
Number of API Calls : 9
Search Keyword : JavaScript
Number of Key Presses : 10
```

The concept of debouncing is pretty straightforward. It delays the function invocation by a defined period of time to avoid unnecessary invocations. So, the function will only be invoked if no event is triggered within that time. If the user triggers a new event during that time, the time will be reset.

Let's consider the previous example again. The issue with the scenario was unnecessary API calls. So, if we define a debounce function with a delay of one second, it will hold back the API call until one second passes without any user events. If the user presses a key within that second, the debounce function will reset the delay and wait for another second to make the API call.

```
function debounce(callback, delay = 1000) {
  var time;

  return (...args) => {
    clearTimeout(time);
    time = setTimeout(() => {
      callback(...args);
    }, delay);
  };
}
```



Although the user has typed the word JavaScript, only a single API call has been invoked. So, debouncing has blocked nine unnecessary API calls from the previous example.

```
var searchBarDom = document.getElementById('search-bar');
var numberOfKeyPresses = 0;
var numberOfApiCalls = 0;

const getSearchResult = debounce(() => {
  numberOfApiCalls += 1;
  console.log('Number of API Calls : ' + numberOfApiCalls);
}, 1000);

searchBarDom.addEventListener('input', function (e) {
  numberOfKeyPresses += 1;
  console.log('Search Keyword : ' + e.target.value);
  console.log('Number of Key Presses : ' + numberOfKeyPresses);
  getSearchResult();
});

function debounce(callback, delay = 1000) {
  var time;
  return (...args) => {
    clearTimeout(time);
    time = setTimeout(() => {
      callback(...args);
    }, delay);
  };
}
```

Throttle is another technique to minimize unnecessary function invocations when using event listeners. However, throttle works a bit differently from debouncing. Instead of delaying, it invokes the callback function at regular intervals as long as the event trigger is active.

For example, assume we have defined the delay as one second in a throttle function. First, it will invoke the callback function immediately. Then, it will use the delay time as the waiting time and invoke the callback function every second until the event trigger becomes inactive.

```
function throttle(callback, delay = 1000) {
  let shouldWait = false;

  return (...args) => {
    if (shouldWait) return;

    callback(...args);
    shouldWait = true;
    setTimeout(() => {
      shouldWait = fal
    }, delay);
  };
}
```

```
var searchBarDom = document.getElementById('search-bar');
var numberOfKeyPresses = 0;
var numberOfApiCalls = 0;

const getSearchResult = throttle(() => {
  numberOfApiCalls += 1;
  console.log('Number of API Calls : ' + numberOfApiCalls);
}, 1000);

searchBarDom.addEventListener('input', function (e) {
  numberOfKeyPresses += 1;
  console.log('Search Keyword : ' + e.target.value);
  console.log('Number of Key Presses : ' + numberOfKeyPresses);
  getSearchResult();
});


function throttle(callback, delay = 1000) {
  let shouldWait = false;

  return (...args) => {
    if (shouldWait) return;

    callback(...args);
    shouldWait = true;
    setTimeout(() => {
      shouldWait = false;
    }, delay);
  };
}
```

```
Console was cleared
Search Keyword : J
Number of Key Presses : 1
Number of API Calls : 1
Search Keyword : Ja
Number of Key Presses : 2
Search Keyword : Jav
Number of Key Presses : 3
Search Keyword : Java
Number of Key Presses : 4
Search Keyword : JavaS
Number of Key Presses : 5
Number of API Calls : 2
Search Keyword : JavaSc
Number of Key Presses : 6
Search Keyword : JavaScr
Number of Key Presses : 7
Search Keyword : JavaScri
Number of Key Presses : 8
Search Keyword : JavaScrip
Number of Key Presses : 9
Search Keyword : JavaScript
Number of Key Presses : 10
Number of API Calls : 3
```

As you can see, only three API calls were made in this example when I typed the word JavaScript. The first call is the initial call, and the other two were made after 5 and 10 key presses, respectively

Difference between debounce and throttle:

Debounce monitors the time delay between user actions and only executes the callback function if the delay exceeds the time delay defined by the developer. So, continuous user actions can significantly delay the callback function’s execution if we use debounce.

On the other hand, throttle uses the time delay to execute the callback function at regular intervals until the event trigger is active. So, it does not delay the callback function execution for a significant period like debounce.

When to use what

Debounce is most suitable for control events like typing or button clicks. Throttle is most suitable for continuous user events like resizing and scrolling.

Algolia Autocomplete API Performance Optimization Via Debounce:

```
function Autocomplete({ hits, refine, delay, defaultQuery, selectedTags, setSelectedTags }) {
  const timerId = useRef(null);
  const inputRef = useRef();
  const dropdownRef = useRef();
  const [dropdownVisible, setDropdownVisible] = useState(false);
  const [selected, setSelected] = useState(-1);
  const [memoizedValue, setMemoizedValue] = useState(null);

  useEffect(() => {
    refine(defaultQuery);
    return () => {};
  }, []);

  function useOutsideAlerter(ref) {
    useEffect(() => {
      function handleClickOutside(event) {
        if (ref.current && !ref.current.contains(event.target) && event.target.id !== 'tags-input') {
          setDropdownVisible(false);
        }
      }

      document.addEventListener('mousedown', handleClickOutside);
      return () => {
        document.removeEventListener('mousedown', handleClickOutside);
      };
    }, [ref]);
  }

  useOutsideAlerter(dropdownRef);

  function onChangeDebounced(text) {      Sai Ashish, 15 months ago • Suggestions Dropdown
    clearTimeout(timerId.current);
    timerId.current = setTimeout(() => refine(text), delay);
  }

  function handleInputChange(e) {
    const text = e.target.value;
    onChangeDebounced(text);
    if (text) setDropdownVisible(true);
    else setDropdownVisible(false);
    setMemoizedValue(text);
  }

  function handleFocusChange() {
    if (inputRef.current.value && hits.length) setDropdownVisible(true);
  }
```

```
function handleInputChange(e) {      Sai Ashish, 15 months ago • Suggestions Dropdown
  const text = e.target.value;
  onChangeDebounced(text);
  if (text) setDropdownVisible(true);
  else setDropdownVisible(false);
  setMemoizedValue(text);
}

function handleFocusChange() {
  if (inputRef.current.value && hits.length) setDropdownVisible(true);
}

const check = dropdownVisible && inputRef.current.value && hits.length > 0;

function handleKeyUp(e) {
  let key = -1;
  if (inputRef.current.value.length) {
    if (e.code === 'ArrowUp') {
      if (selected === -1) key = hits.length - 1;
      else key = selected - 1;
      if (selected - 1 === -1) inputRef.current.value = memoizedValue;
    } else if (e.code === 'ArrowDown') {
      if (selected === hits.length) {
        key = 0;
        inputRef.current.value = memoizedValue;
      } else key = selected + 1;
      if (selected + 1 === hits.length) inputRef.current.value = memoizedValue;
    }
    setSelected(key);
    if (['ArrowUp', 'ArrowDown'].includes(e.code) && selected === -1) setMemoizedValue(inputRef.current.value);
    if (hits[key]) inputRef.current.value = hits[key].title;
    if (e.code === 'Enter') {
      handleInputChange(e);
      setDropdownVisible(false);
      e.target.blur();
    }
  }
}

function handleTagItemClick(e, tag) {
  e.preventDefault();
  e.stopPropagation();
  const item = tag.title ?? memoizedValue;
  inputRef.current.value = item;
  setDropdownVisible(false);
  onChangeDebounced(item);
}
```

```
  return (
    <InputParentContainer>
      <InputContainer hideBorderRadius={check}>
        <IconContainer>
          <LoadingIndicator />
        </IconContainer>
        <Input
          id='tags-input'
          autoComplete='off'
          ref={inputRef}
          onKeyUp={handleKeyUp}
          defaultValue={defaultQuery}
          placeholder='Search by show, influencer, store'
          onChange={handleInputChange}
          onFocus={handleFocusChange}
        />
        <Filter selectedTags={selectedTags} setSelectedTags={setSelectedTags} />
      </InputContainer>
      {check && (
        <CustomDropdown ref={dropdownRef}>
          <TagsList large>
            {hits.slice(0, 4).map((item, key) => (
              <TagItem
                dropdown
                onClick={(e) => handleTagItemClick(e, item)}
                selected={item.title === hits[selected]?.title && selected === key}
                key={item.objectID}
                role='listitem'>
                <CustomHighlight hit={item} attribute='title' />
                <Row>
                  <ImgTag src={item.user.photoURL} alr='img' />
                  <Tagline>
                    <CustomHighlight hit={item} attribute='user.displayName' />
                  </Tagline>
                </Row>
              </TagItem>
            ))}{' '}
          </TagsList>
        </CustomDropdown>
      )}
    </InputParentContainer>
  );
}
```

49. Cohesion, Hoisting & Closure

Cohesion is when elements of a system are grouped together by some criteria. Low cohesion means that app elements don’t have clear boundaries. This may look like a mess.

Hoisting is JavaScript's default behavior of moving declarations to the top.

JavaScript Declarations are Hoisted
In JavaScript, a variable can be declared after it has been used.
In other words; a variable can be used before it has been declared.
JavaScript only hoists declarations, not initializations.

Variables declared with const, just like let, are hoisted to the top of their global, local, or block scope – but without a default initialization.
var variables, as you've seen earlier, are hoisted with a default value of **undefined** so they can be accessed before declaration without errors.
Accessing a variable declared with const before the line of declaration will throw a **cannot access variable before initialization** error.

```
x = 5; // Assign 5 to x

elem = document.getElementById("demo");
elem.innerHTML = x;

var x; // Declare x
```

Variables defined with `let` and `const` are hoisted to the top of the block, but not *initialized*.

Meaning: The block of code is aware of the variable, but it cannot be used until it has been declared.

Using a `let` variable before it is declared will result in a `ReferenceError`.

The variable is in a "temporal dead zone" from the start of the block until it is declared:

Example

This will result in a `ReferenceError`:

```
carName = "Volvo";
let carName;
```

Try it Yourself »

Using a `const` variable before it is declared, is a syntax error, so the code will simply not run.

Example

This code will not run.

```
carName = "Volvo";
const carName;
```

Try it Yourself »

```
const add = (function () {
  let counter = 0;
  return function () {counter += 1; return counter}
})();

add();
add();
add();

// the counter is now 3
```

The variable add is assigned to the return value of a self-invoking function.

The self-invoking function only runs once. It sets the counter to zero (0), and returns a function expression.

This way add becomes a function. The "wonderful" part is that it can access the counter in the parent scope.

This is called a JavaScript closure. It makes it possible for a function to have "private" variables.

The counter is protected by the scope of the anonymous function, and can only be changed using the add function.

A closure is a function having access to the parent scope, even after the parent function has closed.

50. Coupling, Error Boundaries

https://gist.github.com/SaiAshish9/1fb0d98a6dfa00aac9ab68b13a6bae80

Tight coupling means classes and objects are dependent on one another. In general, tight coupling is usually not good because it reduces the flexibility and re-usability of the code while Loose coupling means reducing the dependencies of a class that uses the different class directly. JavaScript is a loosely typed language, meaning you don't have to specify what type of information will be stored in a variable in advance.

```
import {ErrorBoundary} from 'react-error-boundary'

function ErrorFallback({error, resetErrorBoundary}) {
  return (
    <div role="alert">
      <p>Something went wrong:</p>
      <pre>{error.message}</pre>
      <button onClick={resetErrorBoundary}>Try again</button>
    </div>
  )
}

const ui = (
  <ErrorBoundary
    FallbackComponent={ErrorFallback}
    onReset={() => {
      // reset the state of your app so the error doesn't happen again
    }}
  >
    <ComponentThatMayError />
  </ErrorBoundary>
)
```

```jsx
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError(error) {
    // Update state so the next render will show the fallback UI.
    return { hasError: true };
  }

  componentDidCatch(error, errorInfo) {
    // You can also log the error to an error reporting service
    logErrorToMyService(error, errorInfo);
  }

  render() {
    if (this.state.hasError) {
      // You can render any custom fallback UI
      return <h1>Something went wrong.</h1>;
    }

    return this.props.children;
  }
}
```

```jsx
<ErrorBoundary>
  <MyWidget />
</ErrorBoundary>
```

Error boundaries work like a JavaScript catch {} block, but for components. Only class components can be error boundaries. In practice, most of the time you'll want to declare an error boundary component once and use it throughout your application.