Complete Collection

100+
React Infographics



Table Of Contents

1.0 React Project Structure

2.0 React developer roadmap

3.0 12 Essential React libraries

4.0 How useMemo works?

5.0 Difference between useMemo and useCallback

6.0 You don’t always need useEffect

7.0 Common conditional React mistake

Table Of Contents

8.0 How to properly useCallback

9.0 How to use useImperativeHandle hook?

10 Thinking in React components

11 Useful custom hook: useEffectAfterMount()

12 useEffect cheat sheet

13 React re-renders phases

14 Redux data flow

Table Of Contents

15 Avoid this useEffect mistake

16 How to create custom hooks

17 Introduction to useMemo

18 Using variants from base components

19 Prevent impossible states

20 What does render mean in React?

22 React hooks cheatsheets

Table Of Contents

23 Group related state, and avoid redundant state

24 Resetting a component’s state with a key

25 Don’t write and read refs during render

26 Building the structure of a Dropdown component using composition

27 How to use encapsulation in React components

28 How to create extensible components

29 useEffect vs useLayoutEffect

Table Of Contents

30 React render process

31 useDeferredValue hook

32 How to listen for ref changes in React

33 How to relocate state

34 Don’t use isLoading booleans as source of truth

35 Avoid infinite loops when using useEffect

36 Avoid fetching data in useEffect

Table Of Contents

37 Avoid prop-drilling, instead use composition

38 Avoid racing conditions in useEffect

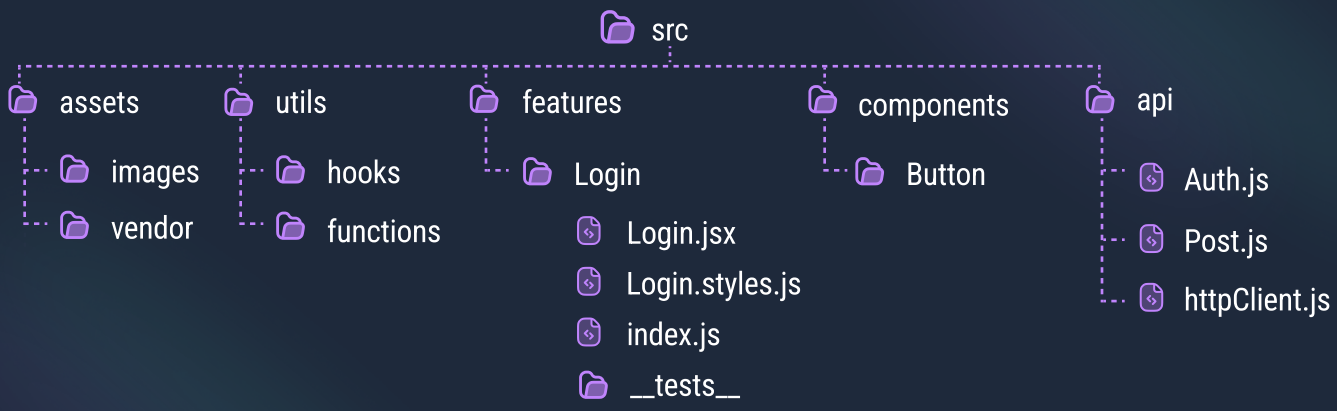
39 Optimize re-rendering

40 useEffect cheat sheet

41 Compound components pattern

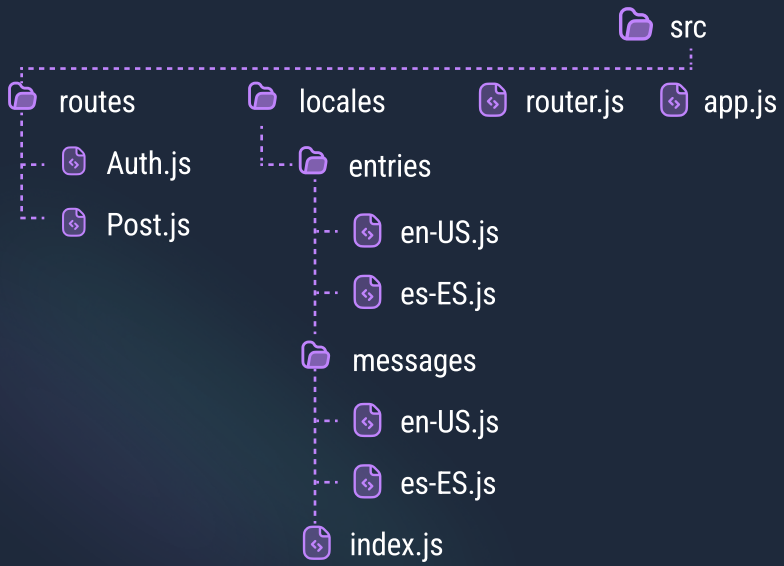
@_georgemoller

React Project Structure



@_georgemoller

React Project Structure





React Project Structure

@_georgemoller

- src
  - components
    - Button
      - Button.jsx — react component
      - index.jsx — util file to export everything from Button.tsx
      - Button.stories.jsx — stories from storybook live here
      - Button.styles.jsx — styles from styled-components library
      - tests
        - Button.test.jsx — where most of unit tests live

@_georgemoller



React Project Structure

- src
  - assets — assets folder contains all static files such as images, fonts etc.
  - utils — everything that is shared between components or features, e.g. functions, hooks, etc.
  - features — components that represent a feature should live here, alongside their tests, styles, etc.
  - components — truly generic, reusable components, usually atoms and molecules like buttons, dropdown.
  - api — API request declarations.
  - routes — routes configuration for each route in the app.
  - locales — holds internationalization config, entries and messages for each language.
  - router.js — entry point for routes, this file is a small wrapper around routes to delegate routes usage.
  - app.js — entry point of the app.

2023

React Developer Roadmap

@_georgemoller





Fundamentals



Hooks



Component Patterns



State Management



Performance Optimization



Test React Apps

@_georgemoller

2023

Fundamentals

1. Learn JSX and how it plays a crucial role in React.
2. Learn what is a component in React, and the difference between a component and an element.
3. Learn how to render components and elements conditionally.
4. Learn the syntax of a functional component.
5. Learn how to display a list of elements in React and the role of the key property.
6. Learn to split different UIs in React components.
7. Learn the concept of props in React, props, default props, prop types.
8. Learn the concept of state and how it differentiates from props.
9. Learn what are hooks, and the main rules of hooks.
10. Learn the useState and useEffect hook (most used hooks)

@_georgemoller

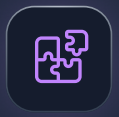


2023

Hooks

1. Learn the use cases for **useEffect**. When you should use it and when **not.**
2. Learn **useCallback** hook, and when to use it to avoid re-renders when passing functions as props.
3. Learn **useMemo** hook, and when to use it to avoid re-renders when passing values as props.
4. Learn **useRef** hook, and how it differentiates with **useState** hook.
5. Learn how you can listen for refs (useRef) changes.
6. Learn **useReducer** and when its a better approach than **useState.**
7. Learn how to write custom hooks, and the main purpose for decoupling logic from components and make it reusable.
8. Learn the concept of state and how it differentiates from props.
9. Learn how to properly fetch data inside **useEffect.**
10. Learn the caveats of fetching data in useEffect and how you can improve this by using libraries like react-query.

@_georgemoller



2023

Component Patterns

1. Learn the composition model of React and how composition can help you solve problems like prop-drilling.
2. Learn the compound component pattern.
3. Learn the Render Props pattern.
4. Learn the difference between controlled and uncontrolled components.
5. Learn how to use the useReducer hook for advanced state management.
6. Learn the Provider pattern.
7. Learn the HOC (higher order component) pattern.



2023

State Management

1. Learn react-query.
2. Learn Redux.
3. Learn XState.

> 💡 Note:
>
> You don't need to use either or all of them, however you'll reach a point where you might need to handle more complex state.
>
> Each of these serve different purposes, so it doesn't hurt to learn at least their use cases in case you need to use them in the future.

@_georgemoller



2023

Performance Optimization

1. Learn code splitting.
2. Learn how to avoid unnecessary renders with **useCallback, useMemo and React.memo.**
3. Learn how to profile components with the DevTools Profiler.
4. Learn how to virtualize long list with react-window or react-virtualized.
5. Learn how to debounce functions.

@_georgemoller



2023

Testing

1. Learn the basics of Jest
2. Learn react-testing-library

@_georgemoller



12 Essential React Libraries

Commonly Used and Production-Ready libraries.

1 React-query

A data-fetching library for web applications. If you use React you probably want to use a third party app to efficiently fetch data. This is my personal preference.

2 Day.js

If you handle dates in your app, chances are that you need a library for this. Day.js is a fast (2kB) alternative to Moment.js with the same modern API.

3 Tailwind CSS

In my opinion, this is the best utility-first CSS framework available. The developer experience it offers is simply incredible, and when it comes to writing styles, I have found no better companion.

4 Sentry

It is an essential tool to track and fix errors, crashes, and exceptions that occur in applications. It automatically captures and reports errors, providing detailed information about the issue.

5 classnames

A nifty little npm package for conditionally joining classNames together. Works great with tailwind.

6 framer-motion

Simply the best animation library for React.

@_georgemoller



12 Essential React Libraries

Commonly Used and Production-Ready libraries.

7 React-window

React Window is a lightweight library for rendering large lists or data sets efficiently in React applications.

When you have a long list of items (e.g., a list of thousands of elements) that you need to display on a web page, rendering them all at once can lead to performance issues, causing slow loading times and high memory usage.

8 Highcharts

Highcharts: Known for its ease of use and comprehensive documentation. It offers a wide range of charts and supports both free and commercial licenses.

9 React-table

Headless UI for building powerful tables & datagrids, with no styling!.

10 React-hook-form + Yup

React Hook Form is a library that simplifies the process of managing and validating forms.

11 Yup

Yup is a JavaScript library used for data schema validation, primarily focused on validating objects and data structures.

@_georgemoller

12 Essential React Libraries

Commonly Used and Production-Ready libraries.

12 React-datepicker

You probably don't want to build a datepicker or calendar from scratch. React-datepicker is a simple and reusable Datepicker component for React. It is highly customizable too!

How useMemo() works in React?

useMemo will only recompute a value when one of the dependencies has changed.



This will be recalculated only when users or occupation props change.

```
import React from 'react'
import searchUsersByOccupation from 'utils/users'

const UsersList = ({ users, occupation }) => {

  const usersByOccupation = React.useMemo(() => {
    return searchUsersByOccupation(users, occupation)
  }, [users, occupation]);

  /*...*/
};
```

@_georgemoller

Difference between useMemo() and useCallback() in React?

@_georgemoller

useCallback()

useCallback will remember a **function** between renders.

Used to fix performance issues, e.g. when **changing** a function can cause expensive components to re-render.

```
const SomeComponent = () => {
  // Whenever SomeComponent re-renders, onItemClick fn reference
  // stays the same, and ExpensiveComponent won't re-render
  const onItemClick = React.useCallback(() => {
    /* do something */
  }, [])
  <ExpensiveComponent onItemClick={onItemClick} />
};

const ExpensiveComponent = React.memo(({onItemClick}) => {
  /*...*/
})
```

useMemo()

useMemo will remember a **value** between renders.

Used to avoid re-calculating expensive operations.

```
// expensive function that takes some time to calculate
import searchUsersByOccupation from 'utils/users'

const UsersList = ({ users, occupation }) => {
  // 💡 Hey React, remember this value between renders
  // Only re-calculate when user's list or occupation changes
  const usersByOccupation = React.useMemo(() => {
    return searchUsersByOccupation(users, occupation)
  }, [users, occupation]);
};
```

You don't always need useEffect()

@_georgemoller

```
function Dashboard({metrics}) {
  // 🔴 Avoid creating a state variable for this
  const [filteredMetrics, setFilteredMetrics] = useState([])

  useEffect(() => {
    const currentFilteredMetrics = filterMetrics(metrics)
    setFilteredMetrics(currentFilteredMetrics)
  }, [metrics]);
  // ...
}
```

When you need to calculate something based on props or state change, ***don't*** put it inside useEffect.

```
function Dashboard({metrics}) {
  // 🟢 Instead, calculate it during rendering phase
  const filteredMetrics = filterMetrics(metrics)
}
```

✓ *Less code* ✓ *Less bugs* ✓ *Faster code*

Instead, calculate it during rendering phase

@_georgemoller

Common React mistake

When using conditional rendering, and using &&, ***don't*** put numbers on the left side of &&.

If the left side is 0, then the whole expression gets that value (0).

Instead do:

`users.length > 0.`

```
const Userslist = ({users}) => {
  return (
    <div>
      {/* 🔴 If users.length equals 0, react will render 0 */}
      {users.length && users.map((user) => <p>{user.name}</p>)}
    </div>
  );
};
```

How to properly useCallback()

@_georgemoller

👎 Bad use case

```
function MyComponent() {
  // 🔴 Button component is light and its re-rendering
  // doesn't create performance issues
  const onClick = React.useCallback(() => {
    // Handle click event
  }, []);

  return <Button onClick={onClick} />;
}

function Button({ onClick }) {
  return <button onClick={onClick}>Click me!</button>;
}
```

👍 Good use case

```
// We use React.memo so that given the same props
// react will skip rendering the component again
const MyExpensiveComponent = React.memo(({onClick}) => {
  /*...*/
})

function MyComponent() {
  // 🟢 When MyComponent re-renders, onClick function
  // remains the same and doesn't break the memoization
  // of MyExpensiveComponent
  const onClick = React.useCallback(() => {
    // Handle click event
  }, []);

  return <MyExpensiveComponent onClick={onClick} />;
}
```

useImperativeHandle()

Use ***useImperativeHandle*** to tell React to expose your own custom object as the value of a ref to the parent component.

```
function MyInput(props, ref) {
  const realInputRef = React.useRef(null);

  React.useImperativeHandle(ref, () => {
    return {
      focus() {
        realInputRef.current.focus();
      },
    };
  });

  return (
    <div>
      <input {...props} ref={realInputRef} />
    </div>
  );
}

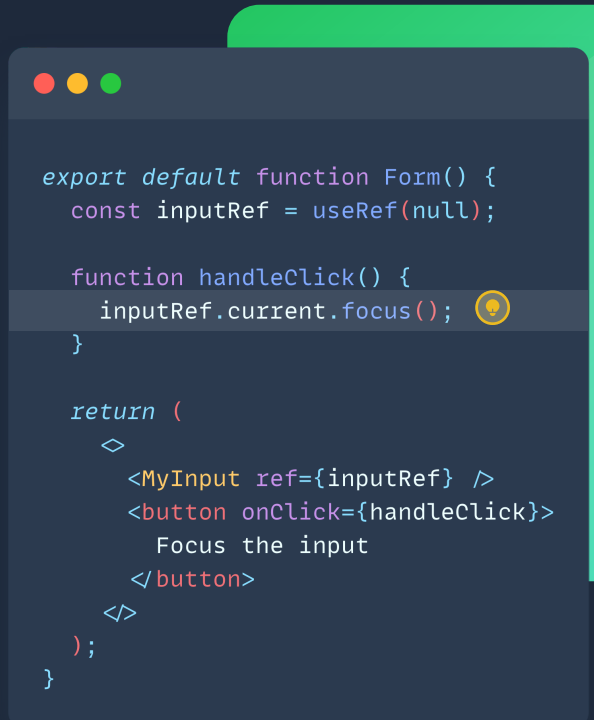
export default React.forwardRef(MyInput);
```

@_georgemoller

useImperativeHandle()

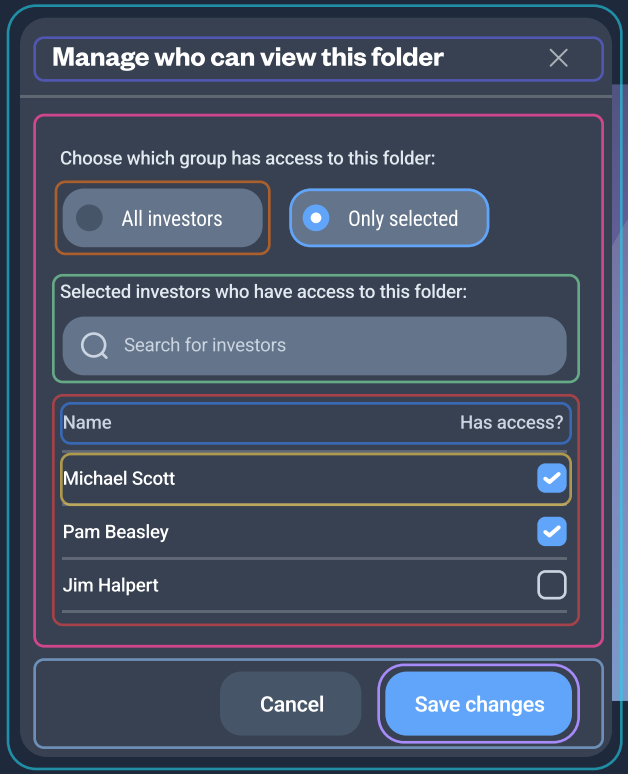
- *inputRef.current* points to the object returned by the useImperativeHandle function.

  This is why we have access to the *focus* function, which uses the *realInputRef*.



Thinking in React components

- <Modal
- <Modal.Header
- <Modal.Body
- <Modal.Footer
- <Button
- <RadioButton
- <SearchField
- <List
- <ListHeader
- <ListItem



Useful custom hook: useEffectAfterMount()

useEffect runs once after component mounts and runs everytime the dependencies change.

useEffectAfterMount will skip running after component mounts, and it will only run after dependencies change.

```
function useEffectAfterMount(cb, dependencies) {
  const hasMounted = React.useRef(false);
  React.useEffect(() => {
    if (hasMounted.current) {
      return cb();
    }
    hasMounted.current = true;
  }, [...dependencies, cb]);
}
```

@_georgemoller

useEffect cheatsheet

```
useEffect(() => {
  /* ... */
})
```

Syncs with all data, runs after every render

```
useEffect(() => {
  /* ... */
}, [state, props])
```

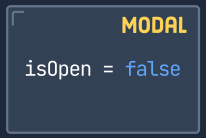
Syncs with state & props, runs after any of these changes

```
useEffect(() => {
  /* ... */
}, [])
```

Syncs with no data, runs only once after first render

@_georgemoller

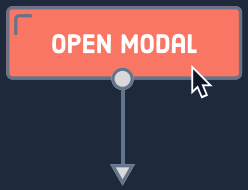
MODAL COMPONENT



React: re-renders phases when state updates

TRIGGER PHASE

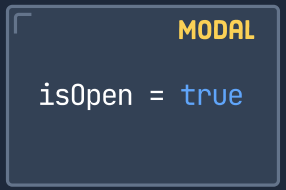
USER CLICKS ON OPEN MODAL BUTTON



`setIsOpen(true)`

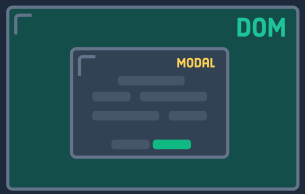
RENDER PHASE

(REACT CALLS MODAL COMPONENT WITH `isOpen = true`)

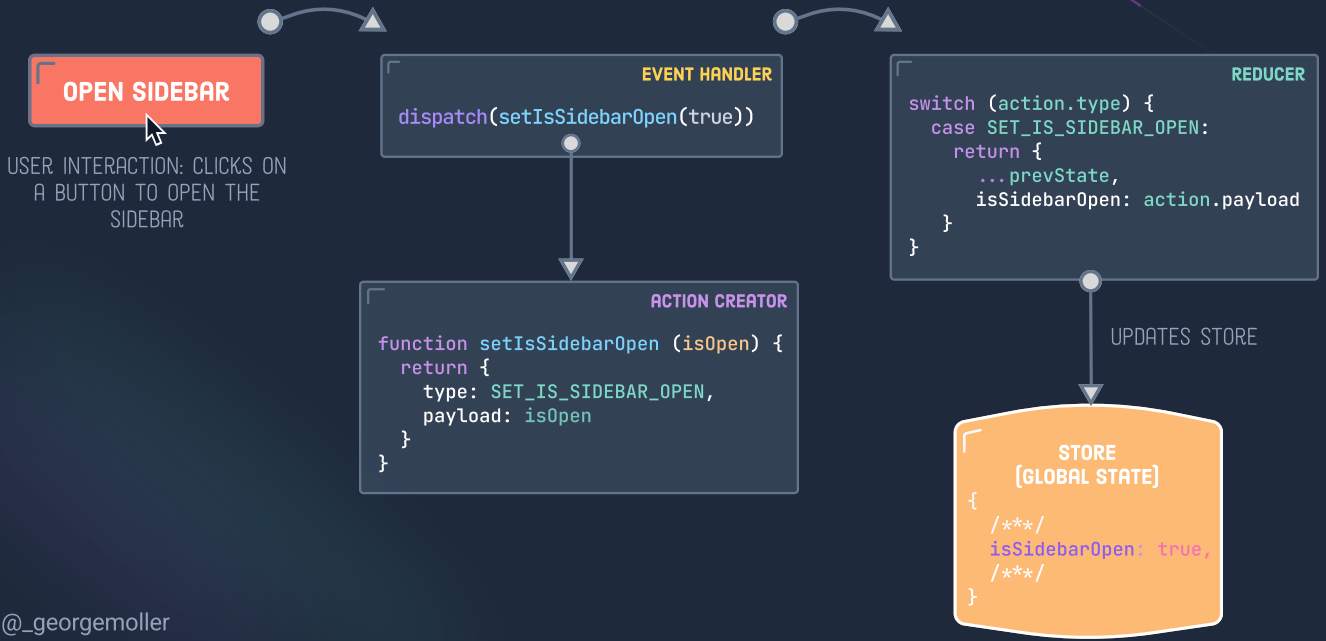


COMMIT PHASE

(REACT MODIFIES THE DOM)



Redux data flow



Avoid this useEffect() mistake

1. The Toggle updates its state first, and React updates the screen.
2. Then React runs the Effect, which calls the onChange function passed from a parent component.
3. Now the parent component will update its own state, starting **another** render pass.



```
function Toggle({ onChange }) {
  const [isOn, setIsOn] = useState(false);

  // 🔴 Avoid: The onChange handler runs too late
  useEffect(() => {
    onChange(isOn);
  }, [isOn, onChange])

  function handleClick() {
    setIsOn(!isOn);
  }
  // ...
}
```

@_georgemoller

Instead notify of the state change directly on the event handler.

① The Toggle updates its state first and notifies the parent component at the same time, in the event handler.



```
function Toggle({ onChange }) {
  const [isOn, setIsOn] = useState(false);

  function updateToggle(nextIsOn) {
    setIsOn(nextIsOn);
    onChange(nextIsOn);
  }

  function handleClick() {
    updateToggle(!isOn);
  }
  // ...
}
```

@_georgemoller

@_georgemoller

① Identify logic that can be reused

LOGIC THAT CAN BE REUSED

```
function Dashboard() {
  const [user, setUser] = useState(null);

  useEffect(() => {
    const getUser = async () => {
      const response = await fetch('user/by/id');
      const json = await response.json();

      setUser(json);
    };

    getUser();
  }, []);

  if (!user) {
    <p>Upss seems like you are not logged in</p>;
  }

  return <h1>{user.name}</h1>;
}
```

```
function Profile() {
  const [user, setUser] = useState(null);

  useEffect(() => {
    const getUser = async () => {
      const response = await fetch('user/by/id');
      const json = await response.json();

      setUser(json);
    };

    getUser()
  }, []);

  return (
    <>
      <h1>{user.name}</h1>
      <h2>{user.occupation}</h2>
    </>
  );
}
```

@_georgemoller

② Extract that logic into a custom hook

```
function useUser() {
  const [user, setUser] = useState(null);

  useEffect(() => {
    const getUser = async () => {
      const response = await fetch('user/by/id');
      const json = await response.json();

      setUser(json);
    };

    getUser();
  }, []);

  return { user };
}
```

@_georgemoller

③ Call the custom hook from components that use that logic.

```
function Dashboard() {
  const { user } = useUser();

  if (!user) {
    <p>Upss seems like you are not logged in</p>;
  }

  return <h1>{user.name}</h1>;
}
```

```
function Profile() {
  const { user } = useUser();

  return (
    <>
      <h1>{user.name}</h1>
      <h2>{user.occupation}</h2>
    </>
  );
}
```

Introduction to useMemo()

useMemo is a React hook used to skip **expensive** calculations by remembering the result of these between re-renders.

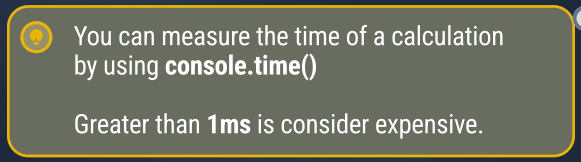
```
function TodoList({ todos, tab, theme }) {
  const visibleTodos = filterTodos(todos, tab);
  /*...*/
}
```

Every time React re-renders this component, **filterTodos()** function will re-run.

useMemo() to the rescue

If filterTodos() is an **expensive** function, we can optimize it by caching (remembering) the return value between re-renders.

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




```
console.time('filter todos');
const visibleTodos = filterTodos(todos, tab);
console.timeEnd('filter todos');

// ⏱ filter array: 1.25ms
```

When to useMemo()?

1. You have an **expensive** function on a component that often re-renders (due to props or state change)
2. You pass it as a prop to a component wrapped in **memo.**
3. The function is later used as a **dependency** of some Hook (e.g. another useMemo or useEffect).



useMemo() dependencies

Be careful of the dependencies you pass into the useMemo hook. If these always change between re-renders, it defeats the purpose of using useMemo().

```
function Dropdown({ allItems, text }) {
  // A new searchOptions object is created on each render
  const searchOptions = { matchMode: 'whole-word', text };

  /* 🔴 useMemo will recompute visibleItems
      when searchOptions change */
  const visibleItems = useMemo(() => {
    return searchItems(allItems, searchOptions);
  }, [allItems, searchOptions]);
```

```
function Dropdown({ allItems, text }) {
  // ✅ A new searchOptions object is created
  // only when text prop changes
  const searchOptions = useMemo(() => {
    return { matchMode: 'whole-word', text };
  }, [text]);

  const visibleItems = useMemo(() => {
    return searchItems(allItems, searchOptions);
  }, [allItems, searchOptions]);
```

Using variants from base components

```
function ButtonBase(props) {
  const { className, as = "button", ...restProps } = props;
  const Element = as;
  return (
    <Element
      {...restProps}
      className={cn(
        `add common classes here`,
        className
      )}
    />
  );
}
```

```
import ButtonBase from './ButtonBase';

const mapSolidVariant = {
  primary: `primary classes`,
  secondary: `secondary classes`,
  /* ... */
};

function ButtonSolid(props) {
  const { variant = "primary", ...restProps } = props;
  return (
    <ButtonBase
      {...restProps}
      className={cn(
        "custom button solid classes here",
        mapSolidVariant[variant]
      )}
    />
  );
}
```

Click me!

- *Close for modification*
  Core behavior and UI is isolated in <ButtonBase
- *Open for extension*
  New variants can be created by extending <ButtonBase

Using variants from base components

```
function ButtonBase(props) {
  const { className, as = "button", ...restProps } = props;
  const Element = as;
  return (
    <Element
      {...restProps}
      className={cn(
        `add common classes here`,
        className
      )}
    />
  );
}
```

```
import ButtonBase from './ButtonBase';

const mapOutlineVariant = {
  primary: `primary classes`,
  secondary: `secondary classes`,
  /* ... */
};

function ButtonOutline(props) {
  const { variant = "primary", ...restProps } = props;
  return (
    <ButtonBase
      {...restProps}
      className={cn(
        "custom button outline classes here",
        mapOutlineVariant[variant]
      )}
    />
  );
}
```

Click me!

- *Close for modification*
  Core behavior and UI is isolated in <ButtonBase
- *Open for extension*
  New variants can be created by extending <ButtonBase

React

@_georgemoller

Prevent impossible states

```
const MyComponent = () => {
  const [isTyping, setIsTyping] = useState(false)
  const [isEmpty, setIsEmpty] = useState(false)
  /*...*/
}
```

isTyping = true	isEmpty = false
isTyping = false	isEmpty = true
isTyping = false	isEmpty = false
isTyping = true	isEmpty = true

4 possible combinations

1 imposible state

React

@_georgemoller

Prevent impossible states

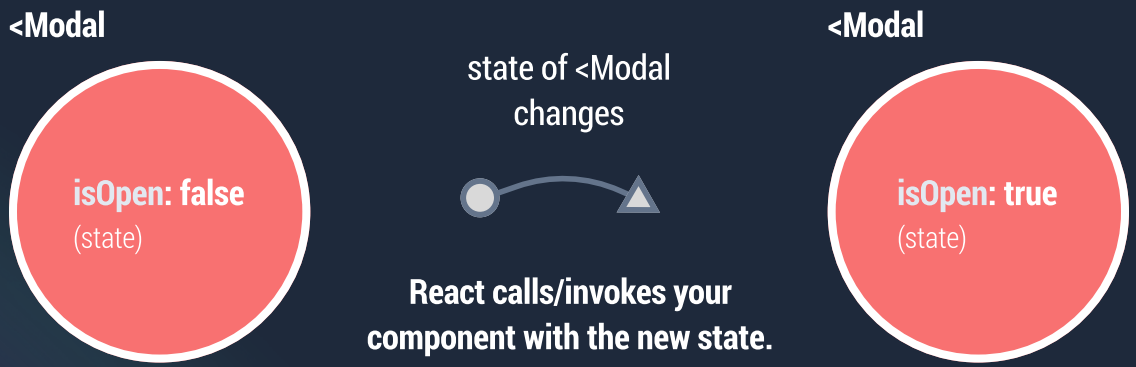
```
const MyComponent = () => {
  const [status, setStatus] = useState('empty')

  const isEmpty = status === 'empty'
  const isTyping = status === 'typing'
  /*...*/
}
```

You can remove impossible states, by combining these into a ***status*** variable that holds only one of these values: ‘empty’ or ‘typing’.

What does Render mean in React?

Render is just a fancy word for saying that React is calling one of your components.



React

@_georgemoller

React hooks cheatsheet

State

- **`useState()`**
  Declares a state variable.

```
const [age, setAge] = useState(42);
```

- **`useReducer()`**
  Declares a state variable managed with a reducer.

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

Context

- **`useContext()`**
  Reads and subscribes to a context.

```
const theme = useContext(ThemeContext);
```

React

@_georgemoller

React hooks cheatsheet

Effects

- `useEffect()`

  Synchronize external state.

```
useEffect(() => {
  const unsubscribe = socket.connect(props.userId);

  return () => {
    unsubscribe();
  }
}, [props.userId]);
```

- `useLayoutEffect()`

  Read layout DOM state.

```
useLayoutEffect(() => {
  // Read DOM layout
});
```

React

@_georgemoller

React hooks cheatsheet

Effects

- `useInsertionEffect()`

  Insert styles into the DOM.

```
useInsertionEffect(() => {
  // Insert styles
});
```

Subscribing

- `useSyncExternalStore()`

  Subscribe to external state.

```
const state = useSyncExternalStore(subscribe, getSnapshot);
```

React

@_georgemoller

React hooks cheatsheet

Memoization

- **`useCallback()`**

  Return a memoized callback.

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

- **`useMemo()`**

  Return a memoized value.

```
const value = useMemo(() => {
  return calculateValue(a, b);
}, [a, b]);
```

React

@_georgemoller

React hooks cheatsheet

Refs

- **`useRef()`**
  Declares a ref.

```
const inputRef = useRef(null);
```

- **`useImperativeHandle()`**
  Customize instance value exposed to parent refs.

```
useImperativeHandle(ref, () => {
  // ...
});
```

React

@_georgemoller

React hooks cheatsheet

Accessibility

`useId()`

Generate unique IDs across the server and client.

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

Debugging

`useDebugValue()`

Display a label for custom hooks.

```
useDebugValue('Custom Label');
```

React

@_georgemoller

React hooks cheatsheet

Events

useEvents()

Extract non-reactive effect logic into an event function.

```
useEvent(callback)
```

Transitions

useDeferredValue()

Defer to more urgent updates.

```
const deferredValue = useDeferredValue(value);
```

useTransition()

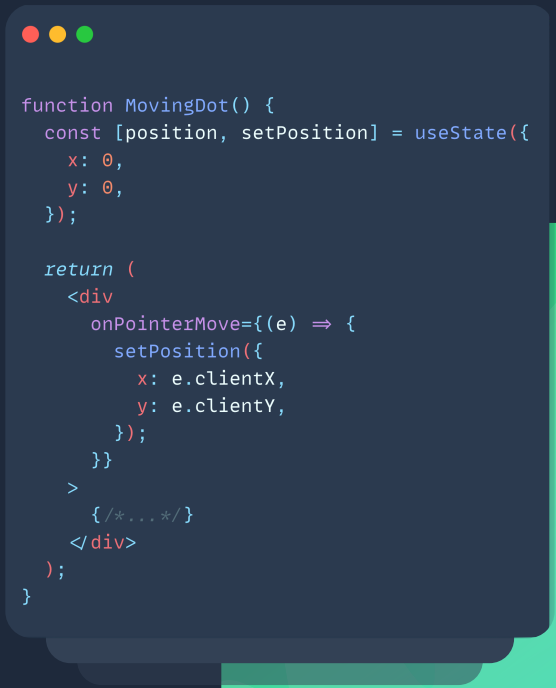
Lets you mark updates in the provided callback as transitions.

```
startTransition(() => {
  setCount(count + 1);
})
```

React state tips!

Group related state

If two or more state variables are likely to change together, then you should consider putting those variables into an object.



```
function MovingDot() {
  const [position, setPosition] = useState({
    x: 0,
    y: 0,
  });

  return (
    <div
      onPointerMove={(e) => {
        setPosition({
          x: e.clientX,
          y: e.clientY,
        });
      }}
    >
      {/*...*/}
    </div>
  );
}
```

@_georgemoller

React state tips!

@_georgemoller

Avoid redundant state

Calculations from props or state should not be put in the component's state.

```
export default function Form() {
  const [firstName, setFirstName] = useState('');
  const [lastName, setLastName] = useState('');
  const [fullName, setFullName] = useState('');

  function handleFirstNameChange(e) {
    setFirstName(e.target.value);
    setFullName(e.target.value + ' ' + lastName);
  }

  function handleLastNameChange(e) {
    setLastName(e.target.value);
    setFullName(firstName + ' ' + e.target.value);
  }

  /*...*/
}
```

```
export default function Form() {
  const [firstName, setFirstName] = useState('');
  const [lastName, setLastName] = useState('');

  const fullName = firstName + ' ' + lastName;

  function handleFirstNameChange(e) {
    setFirstName(e.target.value);
  }

  function handleLastNameChange(e) {
    setLastName(e.target.value);
  }

  /*...*/
}
```

React state tips!

Resetting a component's state with a key

Whenever you pass in a different key to a component you are resetting the component's state.



This function will change the <Form component key, resetting its state.

```
export default function App() {
  const [awesomeKey, setAwesomeKey] = useState(0);

  function resetComponentState() {
    setAwesomeKey(Math.random())
  }

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

@_georgemoller

React ref tips!

@_georgemoller

Don't write/read a ref.current during rendering.

Calculations from props or state should not be put in the component's state.

```
function MyComponent() {
  myRef.current = 123;

  return <h1>{myOtherRef.current}</h1>;
}
```

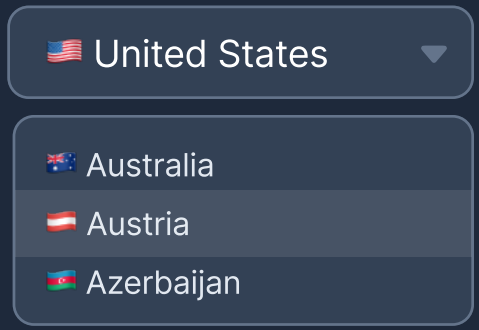
You can read or write refs from event handlers or effects instead.

```
function MyComponent() {
  useEffect(() => {
    myRef.current = 123;
  });

  function handleClick() {
    doSomething(myOtherRef.current);
  }
  //...
}
```

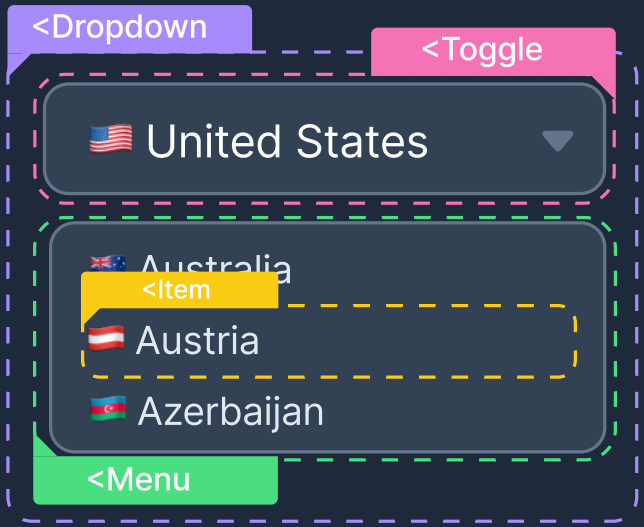
@_georgemoller

Build a dropdown UI using React Components



@_georgemoller

Build a dropdown UI using React Components



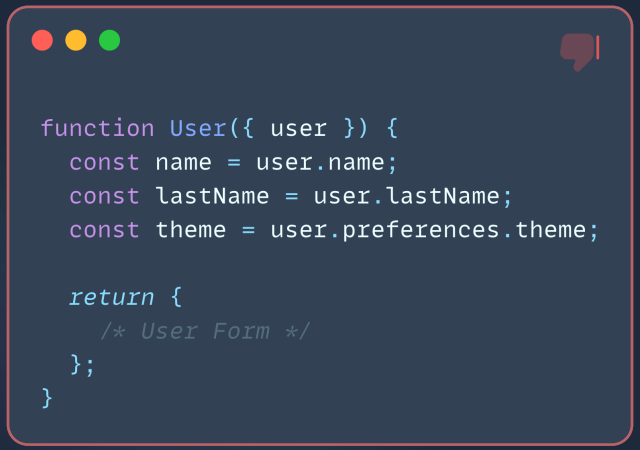
@_georgemoller

Build a dropdown UI using React Components

```
const SomeComponent = () => {
  return (
    <Dropdown>
      <Dropdown.Toggle>
         Select a country
      </Dropdown.Toggle>

      <Dropdown.Menu>
        <Dropdown.Item>🇦🇺 Australia</Dropdown.Item>
        <Dropdown.Item>🇦🇹 Austria</Dropdown.Item>
        <Dropdown.Item>🇦🇿 Azerbaijan</Dropdown.Item>
      </Dropdown.Menu>
    </Dropdown>
  );
};
```

How to use encapsulation in your React components.



```
function User({ user }) {
  const name = user.name;
  const lastName = user.lastName;
  const theme = user.preferences.theme;

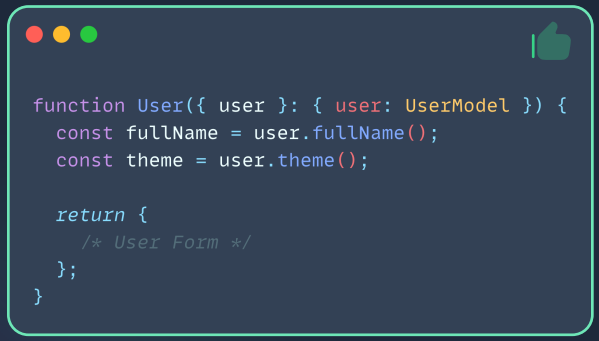
  return {
    /* User Form */
  };
}
```

User component knows too much about the user.

e.g. address has other properties that are not being consumed by <User so no need to expose them.

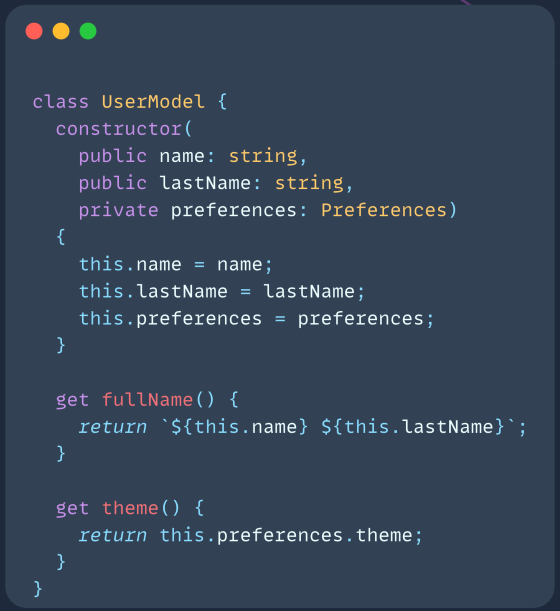
@_georgemoller

How to use encapsulation in your React components.



```
function User({ user }: { user: UserModel }) {
  const fullName = user.fullName();
  const theme = user.theme();

  return {
    /* User Form */
  };
}
```



```
class UserModel {
  constructor(
    public name: string,
    public lastName: string,
    private preferences: Preferences)
  {
    this.name = name;
    this.lastName = lastName;
    this.preferences = preferences;
  }

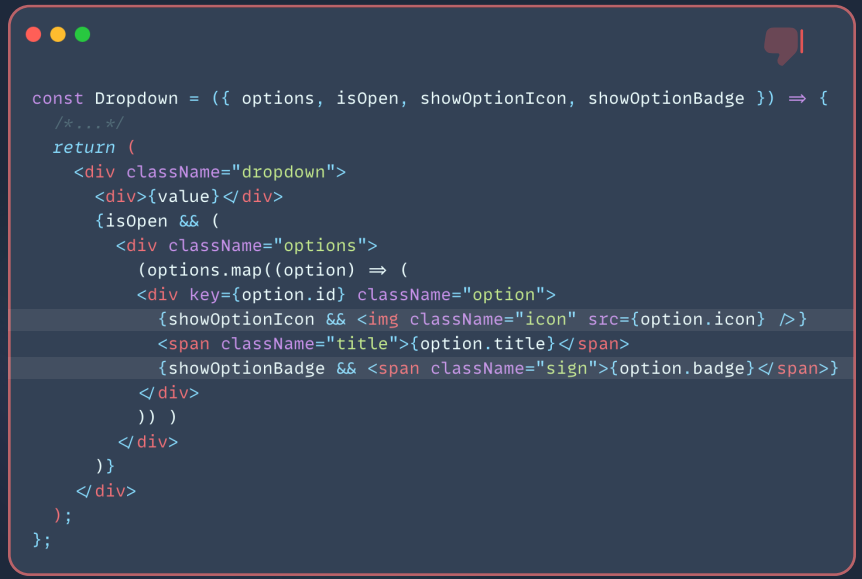
  get fullName() {
    return `${this.name} ${this.lastName}`;
  }

  get theme() {
    return this.preferences.theme;
  }
}
```

UserModel encapsulation can prevent unwanted access to sensitive data (preferences). You can go one step further and also hide information through access modifiers and reduce unwanted changes on properties.

How to create extensible React components.

@_georgemoller



```
const Dropdown = ({ options, isOpen, showOptionIcon, showOptionBadge }) => {
  /*...*/
  return (
    <div className="dropdown">
      <div>{value}</div>
      {isOpen && (
        <div className="options">
          (options.map((option) => (
          <div key={option.id} className="option">
            {showOptionIcon && <img className="icon" src={option.icon} />}
            <span className="title">{option.title}</span>
            {showOptionBadge && <span className="sign">{option.badge}</span>}
          </div>
          )) )
        </div>
      )}
    </div>
  );
};
```

Dropdown isn't closed for modification as we need to change it in order to extend it, in other words: it isn't open for extension.

How to create extensible React components.

@_georgemoller

```
const Dropdown = ({ options, isOpen, toggleDropdown }) => {
  /*...*/
  return (
    <div className="dropdown">
      <div>{value}</div>
      {isOpen && (
        <div className="options">
          {React.Children.map(children, (child) =>
            React.cloneElement(child, { toggleDropdown })
          )}
        </div>
      )}
    </div>
  );
};
```

Now dropdown doesn't care about how options are rendered.

It's up to the component's consumer to determine how to show the options.

How to create extensible React components.

@_georgemoller



```
// Somewhere in the dropdown component
const Option = (props) => (
  <div className="option">
    {props.children}
  </div>
);

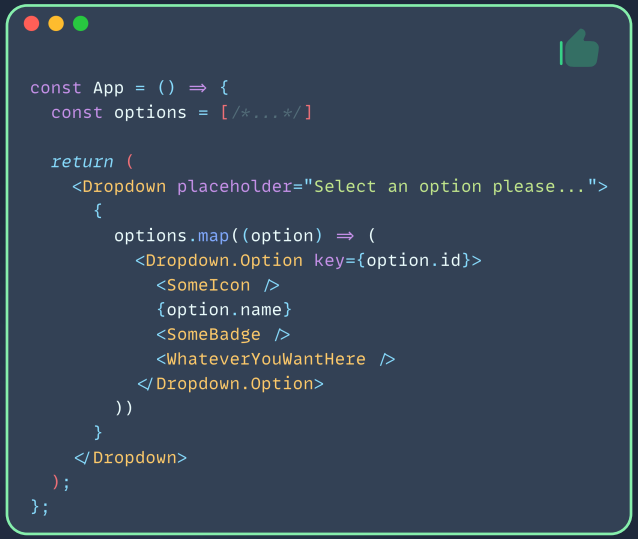
// ...

Dropdown.Option = Option
```

Dropdown can expose some style conditions on how options should be displayed, but is up to the consumer to determine its content.

How to create extensible React components.

@_georgemoller



```
const App = () => {
  const options = [/*...*/]

  return (
    <Dropdown placeholder="Select an option please...">
      {
        options.map((option) => (
          <Dropdown.Option key={option.id}>
            <SomeIcon />
            {option.name}
            <SomeBadge />
            <WhateverYouWantHere />
          </Dropdown.Option>
        ))
      }
    </Dropdown>
  );
};
```

Now Dropdown is closed for modification as we don't need to change it in order to extend it, in other words: it is open for extension.

REACT HOOKS.

useId

@_georgemoller

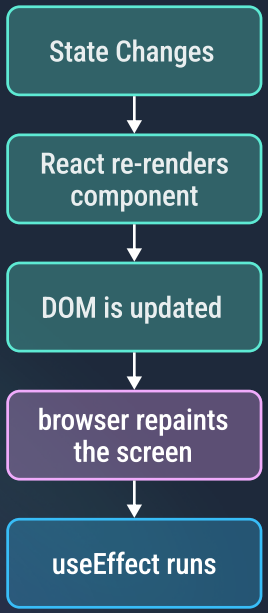
useId is a React Hook for generating unique IDs that can be passed to accessibility attributes.

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

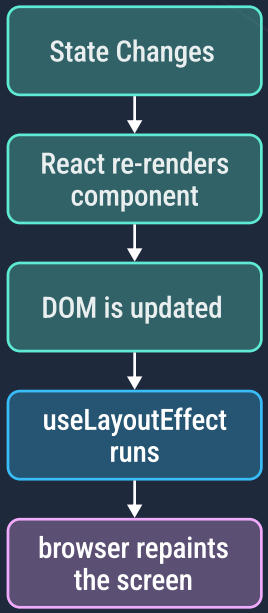
```jsx
import { useId } from 'react';

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>
  );
}
```

useEffect



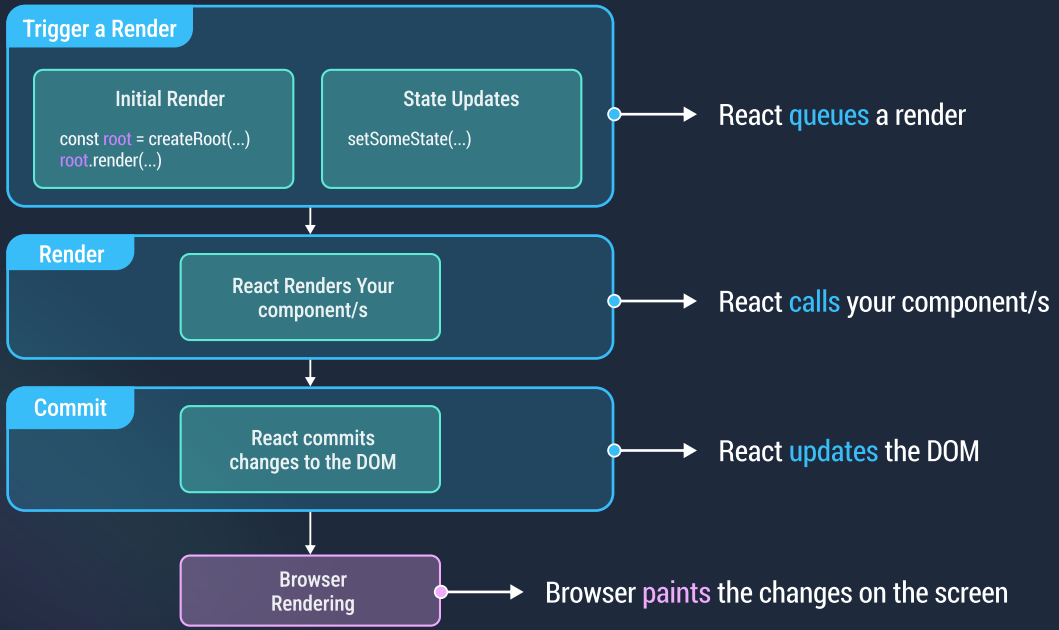
useLayoutEffect



@_georgemoller

React Render Process

@_georgemoller



@_georgemoller

useDeferredValue()
hook

@_georgemoller

REACT HOOKS.

useDeferredValue

useDeferredValue is a React hook that **delays the update of a state value**

Let's suppose we have a SlowList component that:

1. Receives a *search* value.
2. Performs an expensive computation.
3. Filters a country list by the *search* term.

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

const countryList = ['United States', 'Senegal', 'Canada'];
const SlowList = memo(({ search }) => {
  // → Simulate expensive compmutation starts
  const now = performance.now();
  while (performance.now() - now < 40) {}
  // → Simulate expensive compmutation ends

  return countryList.filter((el) => el.indexOf(search) >= 0);
});
```

REACT HOOKS.

useDeferredValue

The Problem

On every key stroke on the search input, the UI feels slow, because we are performing the expensive computation inside the SlowList component everytime the search value changes.

The user doesn't see the input's value update immediately.

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

export default function App() {
  const [search, setSearch] = useState('');

  const onSearchChange = (event) => {
    setSearch(event.target.value);
  };

  return (
    <>
      <input onChange={onSearchChange} value={search} />
      <SlowList search={search} />
    </>
  );
}
```

@_georgemoller

REACT HOOKS.

useDeferredValue

The Solution

useDeferredValue provides a deferred version of the value that we pass in to it.

How it works?

1. React first re-renders with the new search (e.g. "Un") but the old deferredSearch (still "U").

2. In the background, React tries to re-render with both search and deferredSearch updated to "Un".

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

export default function App() {
  const [search, setSearch] = useState('');
  const deferredSearch = useDeferredValue(search)

  const onSearchChange = (event) => {
    setSearch(event.target.value);
  };

  return (
    <>
      <input onChange={onSearchChange} value={search} />
      <SlowList search={deferredSearch} />
    </>
  );
}
```

@_georgemoller

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

export default function App() {
  const [search, setSearch] = useState('');
  const deferredSearch = useDeferredValue(search)

  const onSearchChange = (event) => {
    setSearch(event.target.value);
  };

  console.log(search, deferredSearch)
  return (
    <>
      <input onChange={onSearchChange} value={search} />
      <SlowList search={deferredSearch} />
    </>
  );
}
```

Info

If we type into the input again, React will **abandon the current render and restart with the new value**.

This is how it “skips” expensive calculation unless it’s the last deferred value.

search	deferredSearch
s	“ “
se	s
sen	se
sene	sen
seneg	sene
senega	seneg
senegal	senega
senegal	senegal

How to listen for ref changes in React

@_georgemoller

Don't do this

```
function SomeComponent() {
  const [height, setHeight] = useState(0);
  const refElement = React.useRef(null);

  React.useEffect(() => {
    // 🚨 An object ref doesn't notify us about
    // changes to the current ref value.
    if (refElement.current) {
      setHeight(refElement.current.offsetHeight);
    }
  }, [refElement.current]);

  return (
    <>
      <h1 ref={refElement}>Hello, world</h1>
      <h2>The above header is {Math.round(height)}px tall</h2>
    </>
  );
}
```

Do this instead

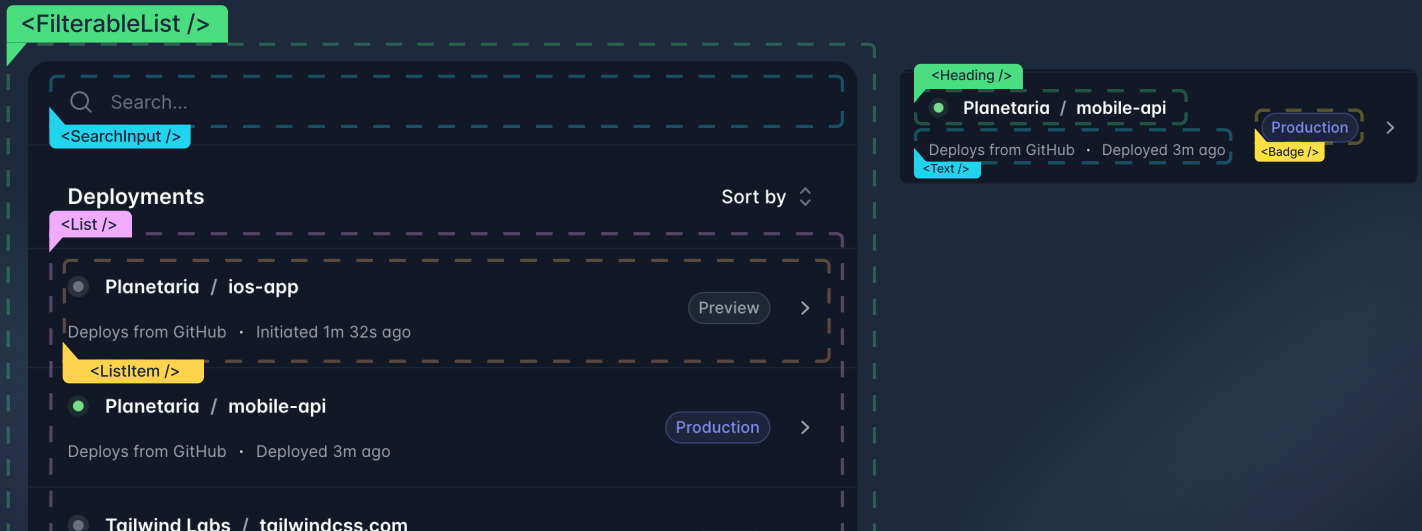
```
function SomeComponent() {
  const [height, setHeight] = useState(0);

  // ✅ Using a callback ref ensures that even if a child component
  // displays the measured node later (e.g. in response to a click),
  // we still get notified about it in the parent component and can
  // update the measurements.
  const refElement = useCallback(node => {
    if (node !== null) {
      setHeight(node.getBoundingClientRect().height);
    }
  }, []);

  return (
    <>
      <h1 ref={refElement}>Hello, world</h1>
      <h2>The above header is {Math.round(height)}px tall</h2>
    </>
  );
}
```

How to build UIs with React Components?

@_georgemoller





Avoid this common mistake

1. Whenever **searchTerm** changes, **<Dashboard** component will re-render.

2. <Dashboard doesn't use searchTerm, and re-rendering it might be expensive.

```
function App() {
  const [searchTerm, setSearchTerm] = React.useState('');
  const user = { name: 'George' };

  const onChange = (newSearchTerm) => {
    setSearchTerm(newSearchTerm);
  };

  return (
    <div>
      <Sidebar>
        <SearchField
          user={user}
          searchTerm={searchTerm}
          onChange={onChange}
        />
      </Sidebar>

      {/* Dashboard is an expensive/slow component */}
      <Dashboard user={user} />
    </div>
  );
```

Don't do this

@_georgemoller



Don’t use React.memo & React.useMemo to solve this.

```
function App() {
  const [searchTerm, setSearchTerm] = React.useState('');
  const user = React.useMemo(() => ({ name: 'George' }));

  const onChange = (newSearchTerm) => {
    setSearchTerm(newSearchTerm);
  };

  return /*...*/;
}

const Dashboard = React.memo(
  () => {
    return; /*...*/
  },
  (prev, next) => prev.user === next.user
);
```

Don’t do this

@_georgemoller



Solution: Relocate state.

1. Move the searchTerm state down to the <SearchField component.

2. When searchTerm changes, React doesn't bother on calculating what changed in the <App component, but instead only re calculates the <SearchField component.

```
function App() {
  const user = { name: 'George' };

  return (
    <div>
      <Sidebar>
        <SearchField user={user} />
      </Sidebar>

      {/* Dashboard is an expensive/slow component */}
      <Dashboard user={user} />
    </div>
  );
}

function SearchField() {
  const [searchTerm, setSearchTerm] = React.useState('');

  const onChange = (newSearchTerm) => {
    setSearchTerm(newSearchTerm);
  };

  return; /*...*/
}
```

@_georgemoller

Don’t use isLoading booleans as source of truth.

Why? to avoid **impossible states.**

If an error happens when fetching users, we would have an impossible state for a fraction of ms:

`isLoading = true` `error = ‘Upps...’`

```
const App = () => {
  const [isLoading, setIsLoading] = React.useState(false);
  const [error, setError] = React.useState('');

  const fetchUsers = async () => {
    try {
      setIsLoading(true);
      fetch(/*...*/);
    } catch (error) {
      setError('Upps, an error occurred');
    } finally {
      setIsLoading(false);
    }
  };

  React.useEffect(() => {
    fetchUsers();
  }, []);

  if (isLoading) {
    return <p>Is Loading...</p>;
  }

  if (error) {
    return <p>{error}</p>;
  }

  return <p>Great we have some users data</p>;
};
```

Use boolean variables that are derived from a source of truth:

Our source of truth is `**status**`.

We can create boolean variables derived from that like so:

```
isLoading = status === 'pending' || status === 'idle'
```

```
isSuccess = status === 'successful'
```

```
isRejected = status === 'rejected'
```

```
const App = () => {
  const [status, setStatus] = React.useState('idle')

  const fetchUsers = async () => {
    try {
      setStatus('pending');
      fetch(/*...*/);
      setStatus('successful')
    } catch (error) {
      setStatus('rejected');
    }
  };

  React.useEffect(() => {
    fetchUsers();
  }, []);

  if (status === 'pending' || status === 'idle') {
    return <p>Is Loading...</p>;
  }

  if (status === 'rejected') {
    return <p>Upps, an error occurred</p>;
  }

  return <p>Great we have some users data</p>;
};
```

@_georgemoller

@_georgemoller

Avoid infinite loops when using useEffect()

1. On first render we setViews to the current state (0) + userViews (5). This triggers a **re-render on <App**.

2. State changed so React re-render App, then a **new object reference for user is created**, which triggers the sideEffect inside useEffect.

3. Views gets updated, triggering a new re-render which creates a new object reference on user, triggering the side effect again...

```
export default function App() {
  const [views, setViews] = React.useState(0);
  const user = { name: 'George', userViews: 5 };

  React.useEffect(() => {
    setViews((currState) => currState + user.userViews);
  }, [user]);

  return <p>The count is: {views}</p>;
}
```

Solution: wrap the object with useMemo()

1. Wrap the user object with **useMemo()**.
2. This will prevent React from creating new object references on re-renders.
3. Given that no new references for the user object are created, the side effect doesn't run again.

```
export default function App() {
  const [views, setViews] = React.useState(0);
  const user = React.useMemo(() => {
    return { name: 'George', userViews: 5 };
  }, []);

  React.useEffect(() => {
    setViews((currState) => currState + user.userViews);
  }, [user]);

  return <p>The count is: {views}</p>;
}
```

@_georgemoller



Avoid fetching data inside useEffect

1. When components unmount and remount, data would need to be fetched again.
2. Fetching directly in Effects can create "network waterfalls" where each component fetches data sequentially.
3. Fetching directly in Effects often doesn't allow for data preloading or caching.
4. Effects don't run on the server, resulting in server-rendered HTML with no data.

```
import { fetchUser } from './api.js';

export default function App({userId}) {
  const [user, setUser] = React.useState(null);

  React.useEffect(() => {
    fetchUser(userId).then(result => {
        setUser(result);
    });
  }, [userId]);
  // ...
}
```



Instead use a framework, or library that handles data fetching mechanism

e.g. react-query

1. React query includes built-in caching capabilities.
2. It provides abstractions that manage loading states, error states, and data synchronization between components.
3. React query optimizes data fetching by deduplicating identical queries and batching them together.
4. Potentially help you save on bandwidth and increase memory performance.

```
import { fetchUser } from './api.js';

import { useQuery } from 'react-query'

export default function App({userId}) {
  const { data } = useQuery(['user', userId], fetchUser)
  // ...
}
```



Reset a component's state with the key prop

Keys in Components specify a component's identity.

They answer to the question: ***Is this the "same" or "different" component between re-renders?.***

When the key changes, React re-creates the Form component (and all of its children) from scratch, so its state gets reset.

```
export default function App() {
  const [user, setUser] = useState(null);

  // Get the user...

  return (
    <>
      <button onClick={handleReset}>Reset</button>
      <Form key={user.id} user={user} />
    </>
  );
}

function UserForm({user}) {
  const [name, setName] = useState(user.name);

  return (
      <input
        value={name}
        onChange={e => setName(e.target.value)}
      />
  );
}
```



Caveat

You should avoid setting it on top level components (ones that have multiple expensive children), as it might be the cause of performance issues.



```
export default function App() {
  const [user, setUser] = React.useState(null)
  return <DashboardApp key={user.id} user={user} />
}

// 🧨 Expensive root component
function DashboardApp() {
  // This component is probably a root
  // component that renders many children
}
```

@_georgemoller



Avoid prop-drilling* in React. Use Composition instead.

- Doesn't provide much room for reusability.
- Difficult to read and debug.
- It's not flexible.
- Encourages prop-drilling.

```jsx
export default function App() {
  return (
    <div>
      <MainContent user={user} />
    </div>
  );
}

function MainContent({user}) {
  return (
    <div>
      <Sidebar user={user} />
      <CenterContent user={user} />
      <RightContent user={user} />
    </div>
  );
}
```

```jsx
function CenterContent({ user }) {
  return (
    <div>
      {' '}
      <Posts user={user} />{' '}
    </div>
  );
}
function Posts({ user }) {
  return (
    <div>
      {' '}
      <PostItem user={user} />{' '}
    </div>
  );
}
```

*__Prop-driling__ occurs when a parent component passes data down to its children and then those children pass the same data down to their own children, so on and so forth.

@_georgemoller



Avoid prop-drilling in React. Use Composition instead.

By using React's composition model:

- It facilitates modular and reusable code.
- It improves code organization and maintainability.
- It enables flexibility in your UI.
- Avoid prop-drilling.

```jsx
export default function App() {
  return (
    <div>
      <MainContent
        sidebar={<Sidebar user={user} />}
        centerContent={<CenterContent user={user} />}
        rightContent={<RightContent user={user} />}
      />
    </div>
  );
}
```

@_georgemoller



Avoid Racing Conditions in useEffect()

Follow this possible scenario:

1. First call with user.id = 1.
2. Second call with user.id = 2.
3. First fetch call with user.id = 1 takes longer to finish than second fetch call with user.id = 2
4. Last setUser will be the first one, which is wrong, because the last user should be the last/second call with user.id = 2

```
import { fetchUserById } from 'api/user';

export default function User({ user }) {
  const [user, setUser] = useState(null);

  useEffect(() => {
    const fetcUser = async () => {
      const newUser = await fetchUserById(user.id);
      setUser(newUser);
    };

    fetchUser();
  }, [user.id]);

  if (!user) {
    return null;
  }

  return <div>{user.name}</div>;
}
```

@_georgemoller



Avoid Racing Conditions in useEffect()

Use AbortController to abort older requests:

1. We initialize an AbortController at the start of the effect,
2. We pass the abort controller signal to the signal option argument.
3. This request will fail, if the abortController.abort() function is called.
4. React runs the clean-up function (which aborts the request) before executing the next effect
5. Whenever a new user is passed, react runs the cleanup function, and the older fetch request will be aborted (if still in progress)



```
import { fetchUserById } from 'api/user';

export default function User({ user }) {
  const [user, setUser] = useState(null);

  useEffect(() => {
    const abortController = new AbortController();
    const fetcUser = async () => {
      try {
        const newUser = await fetch(`/user/endpoint/${user.id}/`, {
          signal: abortController.signal,
        });
        setUser(newUser);
      } catch (error) {
        // Aborting a request throws an error
      }
    };

    fetchUser();

    return () => {
      abortController.abort();
    };
  }, [user.id]);

  // rest of the code
}
```



Optimizing re-rendering

In this example, whenever the input value changes, we set a new state for the first name, making the entire **<App component render again**.

Let's assume PageContent is not memoized, this will make the <PageContent to re-render as well, which sounds expensive as it holds the entire page content.

```
function App() {
  const [firstName, setFirstName] = useState('');
  return (
    <>
      <form>
        <input
          value={firstName}
          onChange={e => setFirstName(e.target.value)} />
      </form>
      <PageContent />
    </>
  );
}
```



Optimizing re-rendering

This can be easily solved by **relocating** the firstName state.

Since <PageContent /> doesn't rely on the input state, you can move the input state into its own component.



```
function App() {
  return (
    <>
      <SignupForm />
      <PageContent />
    </>
  );
}

function SignupForm() {
  const [firstName, setFirstName] = useState('');
  return (
    <form>
      <input
        value={firstName}
        onChange={e => setFirstName(e.target.value)}
      />
    </form>
  );
```

@_georgemoller



Don't use useEffect to execute callback functions from parent components



```
const Dropdown = ({ onOpen, onClose }) => {
  const [isOpen, setIsOpen] = useState(false)

  useEffect(() => {
    if (isOpen) {
      onOpen()
    } else {
      onClose()
    }
  }, [isOpen])

  return (
    <button onClick={() => setIsOpen((currState) => !currState)}>
      Toggle dropdown
    </button>
  )
}
```



Instead use the event handler to execute this effect



```
const Dropdown = ({ onOpen, onClose }) => {
  const [isOpen, setIsOpen] = useState(false);

  const toggleView = () => {
    const currIsOpen = !isOpen;
    setIsOpen(currIsOpen);
    if (currIsOpen) {
      onOpen();
    } else {
      onClose();
    }
  }

  return <button onClick={toggleView}>Toggle dropdown</button>
};
```

REACT

useEffect cheat sheet

```
useEffect(() => {
  /* This will run once, after the component mounts */
}, [])

useEffect(() => {
  /* This will run after every component update */
  /* This can be either by change in props or state */
})
```

```
useEffect(() => {
  /* This will run once, after the component mounts */
  return () => {
    /* This will run only after the component unmounts */
  }
}, [])

useEffect(() => {
  /*
    This will run once, after the component mounts,
    and every time somePropOrState changes
  */
}, [somePropOrState])
```

REACT HOOKS TIP

useMemo

George Moller
_georgemoller

useMemo will only recompute the memoized value when one of the dependencies has changed.

```
import searchUsersByOccupation from 'utils/functions'

function UsersList({ users, occupation }) {

  // 🔴 This will be recalculated on every render
  const usersByOccupation =
    searchUsersByOccupation(users, occupation)

  // ...
}
```

Don't do this

```
import searchUsersByOccupation from 'utils/functions'

function UsersList({ users, occupation }) {

  // 🟢 This will be recalculated only when
  // users or occupation change
  const usersByOccupation = React.useMemo(
    () => searchUsersByOccupation(users, occupation),
    [users, occupation])

  // ...
}
```

Do this

REACT HOOKS TIP

useCallback

George Moller
_georgemoller

useCallback hook returns the same function instance between renderings given the same dependency values.

```
function MyComponent() {
  // Bad use case because most likely Button component is light
  // and its re-rendering doesn't create performance issues.
  const handleClick = React.useCallback(() => {
    // handle the click event
  }, []);

  return <Button onClick={handleClick} />;
}

function Button({ onClick }) => {
  return <button onClick={onClick}>Click Me</button/>
}
```

Bad use case

```
// We wrap MyExpensiveComponent into React.memo()
// Given the same props React will skip rendering the component
const MyExpensiveComponent = React.memo(() => {
  /*...*/
})

export function MyParent() {
  // When MyParent component re-renders,
  // onItemClick function object remains the same
  // and doesn't break the memoization of MyExpensiveComponent.
  const onItemClick = React.useCallback(() => {
    /* handle click event */
  }, []);

  return (
    <MyExpensiveComponent
      onItemClick={onItemClick}
    />
  );
}
```

Good use case

REACT TIP

React.forwardRef

Technique for automatically passing a ref through a component to one of its children.

George Moller
_georgemoller

```
const FormInput = React.forwardRef((props, ref) => (
  <div>
    <label>{props.label}</label>
    <input ref={ref}>
      {props.children}
    </input>
  </div>
));

const ParentComponent = () => {
  // Create a ref directly to the input element
  const ref = React.createRef();

  return <FormInput ref={ref} /* ...props */ />;
}
```

Useful for components that render leaf elements such as buttons and inputs, this way you can manage focus, selection etc.

REACT TIP

Compound Components

CCs communicate the relationship between UI components and share implicit state by leveraging an explicit parent-child relationship.

George Moller
_georgemoller



REACT TIP

Compound Components

George Moller
_georgemoller

Context that holds the state to be shared with children.

Provide children (Option) with the ability to use and change parent state.

```
const OptionsContext = React.createContext();

const Options = ({ children, onChangeValue }) => {
  const [selectedValue, setSelectedValue] = React.useState("");

  React.useEffect(() => {
    if (selectedValue) {
      onChangeValue(selectedValue);
    }
  }, [selectedValue, onChangeValue]);

  return (
    <OptionsContext.Provider value={{ selectedValue, setSelectedValue }}>
      {children}
    </OptionsContext.Provider>
  );
};
```

REACT TIP

Compound Components

George Moller
_georgemoller

```
const OptionsContext = React.createContext();

const Option = ({ children, value }) => {
  const { selectedValue, setSelectedValue } = React.useContext(OptionsContext);

  return (
    <button onClick={() => setSelectedValue(value)}>
      {children} {selectedValue === value ? "I'm selected" : "not selected"}
    </button>
  );
};
```

Use of the current selected value to check if the option is selected.

Use the setSelectedValue function to update the selected value every time the button is clicked.


George Moller
_georgemoller

COMPOUND COMPONENTS PATTERN

INTENT

To provide a user friendly API that expresses a relationship between components.

STRUCTURE

```
/*...*/
<Dropdown value={currentValue} onChange={onDropdownChange}>
  <Dropdown.Item> I'm option one </Dropdown.Item>
  <Dropdown.Item> I'm option two </Dropdown.Item>
  <Dropdown.Item> I'm option three </Dropdown.Item>
</Dropdown>
/*...*/
```


George Moller
_georgemoller

COMPOUND COMPONENTS PATTERN

APPLICABILITY

Use it when:

1. You need to use custom elements as children.
2. You need flexibility with the positioning of your children.
3. You have deeply nested components and want to avoid prop drilling.*
4. You want your component to be expressed in a more declarative fashion.

* prop drilling: is the process by which you pass data from one part of the React Component tree to another by going through other parts that do not need the data but only help in passing it around.

George Moller
_georgemoller

IMPLEMENTATION

```
const Dropdown = ({ value, onChange, children }) => {
  return (
    <>
      <button>{value}</button>
      <ul>
        {React.Children.map(children, (child) => {
          return React.cloneElement(child, {
            onChange: onChange
          })
        })}
      </ul>
    </>
  )
}

const DropdownItem = ({ children, onChange }) => {
  return (
    <li onClick={onChange}>{children}</li>
  )
}

Dropdown.displayName = 'Dropdown'
Dropdown.Item = DropdownItem

export default Dropdown
```

USAGE

```
/*...*/
<Dropdown value={currentValue} onChange={onDropdownChange}>
  <Dropdown.Item> I'm option one </Dropdown.Item>
  <Dropdown.Item> I'm option two </Dropdown.Item>
  <Dropdown.Item> I'm option three </Dropdown.Item>
</Dropdown>
/*...*/
```

💡 TIP: use React.Context to share props.

George Moller
_georgemoller

```jsx
const DropdownContext = React.createContext();

const Dropdown = ({ value, onChange, children }) => {
  return (
    <DropdownContext.Provider value={onChange}>
      <button>{value}</button>
      <ul>{children}</ul>
    </DropdownContext.Provider>
  )
}

const DropdownItem = ({ children }) => {
  const onChange = React.useContext(DropdownContext);

  if (!onChange) {
    throw new Error(
      `DropdownItems can not be rendered outside Dropdown`
    );
  }

  return (
    <li onClick={onChange}>{children}</li>
  )
}

/*...*/
```

💡 PRO TIP: extract usage of context on its own custom hook.


George Moller
_georgemoller

```jsx
const DropdownContext = React.createContext();

function useDropdownContext() {
  const context = React.useContext(DropdownContext);
  if (!context) {
    throw new Error(
      `DropdownItems can not be rendered outside Dropdown`
    );
  }
  return context;
}

const Dropdown = ({ value, onChange, children }) => {
  /*...*/
}

const DropdownItem = ({ children }) => {
  const onChange = useDropdownContext()
  return (
    <li onClick={onChange}>{children}</li>
  )
}

/*...*/
```



Remove event listeners in useEffect cleanup function

@_georgemoller

If you don't remove this scroll event in the cleanup function, the event will always be there, even after the component unmounts.

```
React.useEffect(() => {
  document.addEventListener('scroll', showProgress);
}, []);
```

Remove this event listener in the useEffect cleanup function to make sure no unexpected behavior is introduced.

```
React.useEffect(() => {
  document.addEventListener('scroll', showProgress);

  return () => {
    document.removeEventListener('scroll', showProgress);
  };
}, []);
```

Prefer getByRole over getByTestId

@_georgemoller

Extra tests are needed to test default translations.

```
import { render, screen } from '@testing-library/react';

render(
  <div>
    <button data-testid="submit-button" type="submit">
      Submit
    </button>
  </div>
);

screen.getByTestId('submit-button');
```

You are implicitly testing that translations are getting applied correctly.

```
import { render, screen } from '@testing-library/react';

render(
  <div>
    <button type="submit">
      Submit
    </button>
  </div>
);

screen.getByRole('button', {name: /submit/i})
```



Prefer userEvent over fireEvent

@_georgemoller

fireEvent.change will simply trigger a single change event on the input

```
import React from 'react'
import {render, screen, fireEvent} from '@testing-library/react'

test('type on textarea', () => {
  render(<textarea />)
  fireEvent.change(screen.getByRole('textbox'),
   {target: {value: 'Hello,{enter}World!'}}
  )
  expect(screen.getByRole('textbox')).toHaveValue('Hello,\nWorld!
})
```

userEvent.type, will trigger keyDown, keyPress, and keyUp events for each character as well.

```
import React from 'react'
import {render, screen} from '@testing-library/react'
import userEvent from '@testing-library/user-event'

test('type on textarea', () => {
  render(<textarea />)

  userEvent.type(screen.getByRole('textbox'), 'Hello,{enter}World!')
  expect(screen.getByRole('textbox')).toHaveValue('Hello,\nWorld!')
})
```

closer to the user's actual interactions

Rely on integration tests to test interactions like this instead of unit tests.

Prefer accessible selectors over ByTestId

@_georgemoller

- More realistic test.
- More resilient test.
- More accessible app.

```
// ✔
import { render, screen } from '@testing-library/react';

render(<Tab />);
const selectedTab = getByRole('tab', { selected: true })

// JSX
function Tab() {
  return (
    <body>
      <div role="tablist">
        <button role="tab" aria-selected="true">
          Native
        </button>
        <button role="tab" aria-selected="false">
          React
        </button>
      </div>
    </body>
  );
}
```

@_georgemoller



How to initialize state from props

If you absolutely need to initialize state with props avoid using the same prop name as the state name.



```
function ColorPicker(props) {
  const [color, setColor] = React.useState(props.color);

  // some awesome color picker
}
```

Instead call it initial{sth} or default{sth}. This way the consumer knows that the state won't get updated if the prop changes.



```
function ColorPicker({initialColor}) {
  const [color, setColor] = React.useState(initialColor);

  // some awesome color picker
}
```