



React Docs ENG



React: is a JavaScript library for building user interfaces. Learn what React is all about on [our homepage](#) or [in the tutorial](#).

Table of Contents

MAIN CONCEPTS

- [1. Hello World](#)
- [2. Introducing JSX](#)
- [3. Rendering Elements](#)
- [4. Components and Props](#)
- [5. State and Lifecycle](#)
- [6. Handling Events](#)
- [7. Conditional Rendering](#)
- [8. Lists and Keys](#)
- [9. Forms](#)
- [10. Lifting State Up](#)
- [11. Composition vs Inheritance](#)
- [12. Thinking In React](#)

ADVANCED GUIDES

- [1. Accessibility](#)
- [2. Code-Splitting](#)
- [3. Context](#)
- [4. Error Boundaries](#)
- [5. Forwarding Refs](#)
- [6. Fragments](#)
- [7. Higher-Order Components](#)
- [8. Integrating with Other Libraries](#)
- [9. JSX In Depth](#)
- [10. Optimizing Performance](#)
- [11. Portals](#)
- [12. Profiler](#)
- [13. React Without ES6](#)
- [14. React Without JSX](#)
- [15. Reconciliation](#)
- [16. Refs and the DOM](#)
- [17. Render Props](#)
- [18. Static Type Checking](#)

- 19. [Strict Mode](#)
- 20. [Typechecking With PropTypes](#)
- 21. [Uncontrolled Components](#)
- 22. [Web Components](#)

API REFERENCE

- 1. [React](#)
 - [React.Component](#)
- 2. [ReactDOM](#)
- 3. [ReactDOMClient](#)
- 4. [ReactDOMServer](#)
- 5. [DOM Elements](#)
- 6. [SyntheticEvent](#)
- 7. [Test Utilities](#)
- 8. [Test Renderer](#)
- 9. [JS Environment Requirements](#)
- 10. [Glossary](#)

HOOKS

- 1. [Introducing Hooks](#)
- 2. [Hooks at a Glance](#)
- 3. [Using the State Hook](#)
- 4. [Using the Effect Hook](#)
- 5. [Rules of Hooks](#)
- 6. [Building Your Own Hooks](#)
- 7. [Hooks API Reference](#)
- 8. [Hooks FAQ](#)

I. MAIN CONCEPTS

1. Hello World

The smallest React example looks like this:

```
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<h1>Hello, world!</h1>);
```

It displays a heading saying "Hello, world!" on the page.

Click the link above to open an online editor. Feel free to make some changes, and see how they affect the output. Most pages in this guide will have editable examples like this one.

How to Read This Guide

In this guide, we will examine the building blocks of React apps: elements and components. Once you master them, you can create complex apps from small reusable pieces.

Tip

This guide is designed for people who prefer **learning concepts step by step**. If you prefer to learn by doing, check out our [practical tutorial](#). You might find this guide and the tutorial complementary to each other.

This is the first chapter in a step-by-step guide about main React concepts. You can find a list of all its chapters in the navigation sidebar. If you're reading this from a mobile device, you can access the navigation by pressing the button in the bottom right corner of your screen.

Every chapter in this guide builds on the knowledge introduced in earlier chapters. **You can learn most of React by reading the “Main Concepts” guide chapters in the order they appear in the sidebar.** For example, [“Introducing JSX”](#) is the next chapter after this one.

Knowledge Level Assumptions

React is a JavaScript library, and so we'll assume you have a basic understanding of the JavaScript language. **If you don't feel very confident, we recommend [going through a JavaScript tutorial](#) to check your knowledge level** and enable you to follow along this guide without getting lost. It might take you between 30 minutes and an hour, but as a result you won't have to feel like you're learning both React and JavaScript at the same time.

Note

This guide occasionally uses some newer JavaScript syntax in the examples. If you haven't worked with JavaScript in the last few years, [these three points](#) should get you most of the way.

2. Introducing JSX

Consider this variable declaration:

```
const element = <h1>Hello, world!</h1>;
```

This funny tag syntax is neither a string nor HTML.

It is called JSX, and it is a syntax extension to JavaScript. We recommend using it with React to describe what the UI should look like. JSX may remind you of a template language, but it comes with the full power of JavaScript.

JSX produces React “elements”. We will explore rendering them to the DOM in the [next section](#). Below, you can find the basics of JSX necessary to get you started.

Why JSX?

React embraces the fact that rendering logic is inherently coupled with other UI logic: how events are handled, how the state changes over time, and how the data is prepared for display.

Instead of artificially separating *technologies* by putting markup and logic in separate files, React [separates concerns](#) with loosely coupled units called “components” that contain both. We will come back to components in a [further section](#), but if you're not yet comfortable putting markup in JS, [this talk](#) might convince you otherwise.

React [doesn't require](#) using JSX, but most people find it helpful as a visual aid when working with UI inside the JavaScript code. It also allows React to show more useful error and warning messages.

With that out of the way, let's get started!

Embedding Expressions in JSX

In the example below, we declare a variable called `name` and then use it inside JSX by wrapping it in curly braces:

```
const name = 'Josh Perez';
const element = <h1>Hello, {name}</h1>;
```

You can put any valid JavaScript expression inside the curly braces in JSX. For example, `2 + 2`, `user.firstName`, or `formatName(user)` are all valid JavaScript expressions.

In the example below, we embed the result of calling a JavaScript function, `formatName(user)`, into an `<h1>` element.

```
function formatName(user) {
  return user.firstName + ' ' + user.lastName;
}

const user = {
  firstName: 'Harper',
  lastName: 'Perez'
};

const element = <h1> Hello, {formatName(user)}! </h1>;
```

We split JSX over multiple lines for readability. While it isn't required, when doing this, we also recommend wrapping it in parentheses to avoid the pitfalls of automatic semicolon insertion.

JSX is an Expression Too

After compilation, JSX expressions become regular JavaScript function calls and evaluate to JavaScript objects.

This means that you can use JSX inside of `if` statements and `for` loops, assign it to variables, accept it as arguments, and return it from functions:

```
function getGreeting(user) {
  if (user) {
    return <h1>Hello, {formatName(user)}!</h1>;
  }
  return <h1>Hello, Stranger.</h1>;
}
```

Specifying Attributes with JSX

You may use quotes to specify string literals as attributes:

```
const element = <a href="https://www.reactjs.org"> link </a>;
```

You may also use curly braces to embed a JavaScript expression in an attribute:

```
const element = <img src={user.avatarUrl}></img>;
```

Don't put quotes around curly braces when embedding a JavaScript expression in an attribute. You should either use quotes (for string values) or curly braces (for expressions), but not both in the same attribute.

Warning:

Since JSX is closer to JavaScript than to HTML, React DOM uses `camelCase` property naming convention instead of HTML attribute names.

For example, `class` becomes `className` in JSX, and `tabindex` becomes `tabIndex`.

Specifying Children with JSX

If a tag is empty, you may close it immediately with `</>`, like XML:

```
const element = <img src={user.avatarUrl} />;
```

JSX tags may contain children:

```
const element = (
  <div>
```

```
<h1>Hello!</h1>
<h2>Good to see you here.</h2>
</div>
);
```

JSX Prevents Injection Attacks

It is safe to embed user input in JSX:

```
const title = response.potentiallyMaliciousInput;
// This is safe:
const element = <h1>{title}</h1>;
```

By default, React DOM escapes any values embedded in JSX before rendering them. Thus it ensures that you can never inject anything that's not explicitly written in your application. Everything is converted to a string before being rendered. This helps prevent XSS (cross-site-scripting) attacks.

JSX Represents Objects

Babel compiles JSX down to `React.createElement()` calls.

These two examples are identical:

```
const element = <h1 className="greeting">Hello, world!</h1>;

const element = React.createElement('h1', { className: 'greeting' }, 'Hello, world!');
```

`React.createElement()` performs a few checks to help you write bug-free code but essentially it creates an object like this:

```
// Note: this structure is simplified
const element = {
  type: 'h1',
  props: {
    className: 'greeting',
    children: 'Hello, world!'
  }
};
```

These objects are called “React elements”. You can think of them as descriptions of what you want to see on the screen. React reads these objects and uses them to construct the DOM and keep it up to date.

We will explore rendering React elements to the DOM in the next section.

Tip:

We recommend using the “Babel” language definition for your editor of choice so that both ES6 and JSX code is properly highlighted.

3. Rendering Elements

Elements are the smallest building blocks of React apps.

An element describes what you want to see on the screen:

```
const element = <h1>Hello, world</h1>;
```

Unlike browser DOM elements, React elements are plain objects, and are cheap to create. React DOM takes care of updating the DOM to match the React elements.

Note:

One might confuse elements with a more widely known concept of “components”. We will introduce components in the [next section](#). Elements are what components are “made of”, and we encourage you to read this section before jumping ahead.

Rendering an Element into the DOM

Let's say there is a `<div>` somewhere in your HTML file:

```
<div id="root"></div>
```

We call this a “root” DOM node because everything inside it will be managed by React DOM.

Applications built with just React usually have a single root DOM node. If you are integrating React into an existing app, you may have as many isolated root DOM nodes as you like.

To render a React element, first pass the DOM element to `ReactDOM.createRoot()`, then pass the React element to `root.render()`:

```
const root = ReactDOM.createRoot(document.getElementById('root'));
const element = <h1>Hello, world</h1>;
root.render(element);
```

It displays “Hello, world” on the page.

Updating the Rendered Element

React elements are [immutable](#). Once you create an element, you can't change its children or attributes. An element is like a single frame in a movie: it represents the UI at a certain point in time.

With our knowledge so far, the only way to update the UI is to create a new element, and pass it to `root.render()`.

Consider this ticking clock example:

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

function tick() {
  const element = (
    <div>
      <h1>Hello, world!</h1> <h2>It is {new Date().toLocaleTimeString()}</h2>
    </div>
  );
  root.render(element);
}

setInterval(tick, 1000);
```

It calls `root.render()` every second from a `setInterval()` callback.

Note:

In practice, most React apps only call `root.render()` once. In the next sections we will learn how such code gets encapsulated into [stateful components](#).

We recommend that you don't skip topics because they build on each other.

React Only Updates What's Necessary

React DOM compares the element and its children to the previous one, and only applies the DOM updates necessary to bring the DOM to the desired state.

You can verify by inspecting the [last example](#) with the browser tools:

Even though we create an element describing the whole UI tree on every tick, only the text node whose contents have changed gets updated by React DOM.

In our experience, thinking about how the UI should look at any given moment, rather than how to change it over time, eliminates a whole class of bugs.

4. Components and Props

Components let you split the UI into independent, reusable pieces, and think about each piece in isolation. This page provides an introduction to the idea of components. You can find a [detailed component API reference here](#).

Conceptually, components are like JavaScript functions. They accept arbitrary inputs (called “props”) and return React elements describing what should appear on the screen.

Function and Class Components

The simplest way to define a component is to write a JavaScript function:

```
function Welcome(props) {  
  return <h1>Hello, {props.name}</h1>;  
}
```

This function is a valid React component because it accepts a single “props” (which stands for properties) object argument with data and returns a React element. We call such components “function components” because they are literally JavaScript functions.

You can also use an [ES6 class](#) to define a component:

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

The above two components are equivalent from React’s point of view.

Function and Class components both have some additional features that we will discuss in the [next sections](#).

Rendering a Component

Previously, we only encountered React elements that represent DOM tags:

```
const element = <div />;
```

However, elements can also represent user-defined components:

```
const element = <Welcome name="Sara" />;
```

When React sees an element representing a user-defined component, it passes JSX attributes and children to this component as a single object. We call this object “props”.

For example, this code renders “Hello, Sara” on the page:

```
function Welcome(props) {  
  return <h1>Hello, {props.name}</h1>;  
}  
  
const root = ReactDOM.createRoot(document.getElementById('root'));  
const element = <Welcome name="Sara" />;  
root.render(element);
```

Let’s recap what happens in this example:

1. We call `root.render()` with the `<Welcome name="Sara" />` element.
2. React calls the `Welcome` component with `{name: 'Sara'}` as the props.

3. Our `Welcome` component returns a `<h1>Hello, Sara</h1>` element as the result.
4. React DOM efficiently updates the DOM to match `<h1>Hello, Sara</h1>`.

Note: Always start component names with a capital letter.

React treats components starting with lowercase letters as DOM tags. For example, `<div />` represents an HTML div tag, but `<Welcome />` represents a component and requires `Welcome` to be in scope.

To learn more about the reasoning behind this convention, please read [JSX In Depth](#).

Composing Components

Components can refer to other components in their output. This lets us use the same component abstraction for any level of detail. A button, a form, a dialog, a screen: in React apps, all those are commonly expressed as components.

For example, we can create an `App` component that renders `Welcome` many times:

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}

function App() {
  return (
    <div>
      <Welcome name="Sara" /> <Welcome name="Cahal" /> <Welcome name="Edite" />
    </div>
  );
}
```

Typically, new React apps have a single `App` component at the very top. However, if you integrate React into an existing app, you might start bottom-up with a small component like `Button` and gradually work your way to the top of the view hierarchy.

Extracting Components

Don't be afraid to split components into smaller components.

For example, consider this `Comment` component:

```
function Comment(props) {
  return (
    <div className="Comment">
      <div className="UserInfo">
        <img className="Avatar" src={props.author.avatarUrl} alt={props.author.name} />
        <div className="UserInfo-name">{props.author.name}</div>
      </div>
      <div className="Comment-text">{props.text}</div>
      <div className="Comment-date">{formatDate(props.date)}</div>
    </div>
  );
}
```

It accepts `author` (an object), `text` (a string), and `date` (a date) as props, and describes a comment on a social media website.

This component can be tricky to change because of all the nesting, and it is also hard to reuse individual parts of it. Let's extract a few components from it.

First, we will extract `Avatar`:

```
function Avatar(props) {
  return <img className="Avatar" src={props.user.avatarUrl} alt={props.user.name} />;
}
```

The `Avatar` doesn't need to know that it is being rendered inside a `Comment`. This is why we have given its prop a more generic name: `user` rather than `author`.

We recommend naming props from the component's own point of view rather than the context in which it is being used.

We can now simplify `Comment` a tiny bit:

```
function Comment(props) {
  return (
    <div className="Comment">
      <div className="UserInfo">
        <Avatar user={props.author} />
        <div className="UserInfo-name">{props.author.name}</div>
      </div>
      <div className="Comment-text">{props.text}</div>
      <div className="Comment-date">{formatDate(props.date)}</div>
    </div>
  );
}
```

Next, we will extract a `UserInfo` component that renders an `Avatar` next to the user's name:

```
function UserInfo(props) {
  return (
    <div className="UserInfo">
      <Avatar user={props.user} />
      <div className="UserInfo-name">{props.user.name}</div>
    </div>
  );
}
```

This lets us simplify `Comment` even further:

```
function Comment(props) {
  return (
    <div className="Comment">
      <UserInfo user={props.author} />
      <div className="Comment-text">{props.text}</div>
      <div className="Comment-date">{formatDate(props.date)}</div>
    </div>
  );
}
```

Extracting components might seem like grunt work at first, but having a palette of reusable components pays off in larger apps. A good rule of thumb is that if a part of your UI is used several times (`Button`, `Panel`, `Avatar`), or is complex enough on its own (`App`, `FeedStory`, `Comment`), it is a good candidate to be extracted to a separate component.

Props are Read-Only

Whether you declare a component as a function or a class, it must never modify its own props. Consider this `sum` function:

```
function sum(a, b) {
  return a + b;
}
```

Such functions are called “pure” because they do not attempt to change their inputs, and always return the same result for the same inputs.

In contrast, this function is impure because it changes its own input:

```
function withdraw(account, amount) {
  account.total -= amount;
}
```

React is pretty flexible but it has a single strict rule:

All React components must act like pure functions with respect to their props.

Of course, application UIs are dynamic and change over time. In the [next section](#), we will introduce a new concept of “state”. State allows React components to change their output over time in response to user actions, network responses, and anything else, without violating this rule.

5. State and Lifecycle

This page introduces the concept of state and lifecycle in a React component. You can find a [detailed component API reference here](#).

Consider the ticking clock example from [one of the previous sections](#). In [Rendering Elements](#), we have only learned one way to update the UI. We call `root.render()` to change the rendered output:

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

function tick() {
  const element = (
    <div>
      <h1>Hello, world!</h1>
      <h2>It is {new Date().toLocaleTimeString()}.</h2>
    </div>
  );
  root.render(element);
}

setInterval(tick, 1000);
```

In this section, we will learn how to make the `Clock` component truly reusable and encapsulated. It will set up its own timer and update itself every second.

We can start by encapsulating how the clock looks:

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

function Clock(props) {
  return (
    <div>
      <h1>Hello, world!</h1> <h2>It is {props.date.toLocaleTimeString()}.</h2>
    </div>
  );
}

function tick() {
  root.render(<Clock date={new Date()} />);
}

setInterval(tick, 1000);
```

However, it misses a crucial requirement: the fact that the `Clock` sets up a timer and updates the UI every second should be an implementation detail of the `Clock`.

Ideally we want to write this once and have the `Clock` update itself:

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

To implement this, we need to add “state” to the `Clock` component.

State is similar to props, but it is private and fully controlled by the component.

Converting a Function to a Class

You can convert a function component like `Clock` to a class in five steps:

1. Create an [ES6 class](#), with the same name, that extends `React.Component`.
2. Add a single empty method to it called `render()`.
3. Move the body of the function into the `render()` method.
4. Replace `props` with `this.props` in the `render()` body.

5. Delete the remaining empty function declaration.

```
class Clock extends React.Component {
  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.props.date.toLocaleTimeString()}</h2>
      </div>
    );
  }
}
```

`Clock` is now defined as a class rather than a function.

The `render` method will be called each time an update happens, but as long as we render `<Clock />` into the same DOM node, only a single instance of the `Clock` class will be used. This lets us use additional features such as local state and lifecycle methods.

Adding Local State to a Class

We will move the `date` from props to state in three steps:

1. Replace `this.props.date` with `this.state.date` in the `render()` method:

```
class Clock extends React.Component {
  render() {
    return (
      <div>
        <h1>Hello, world!</h1> <h2>It is {this.state.date.toLocaleTimeString()}</h2>
      </div>
    );
  }
}
```

1. Add a class constructor that assigns the initial `this.state`:

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = { date: new Date() };
  }

  render() {
    return (
      <div>
        <h1>Hello, world!</h1> <h2>It is {this.state.date.toLocaleTimeString()}</h2>
      </div>
    );
  }
}
```

Note how we pass `props` to the base constructor:

```
constructor(props) {
  super(props);
  this.state = { date: new Date() };
}
```

Class components should always call the base constructor with `props`.

1. Remove the `date` prop from the `<Clock />` element:

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

We will later add the timer code back to the component itself.

The result looks like this:

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = { date: new Date() };
  }
  render() {
    return (
      <div>
        <h1>Hello, world!</h1> <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}

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

Next, we'll make the `Clock` set up its own timer and update itself every second.

Adding Lifecycle Methods to a Class

In applications with many components, it's very important to free up resources taken by the components when they are destroyed.

We want to set up a timer whenever the `Clock` is rendered to the DOM for the first time. This is called “mounting” in React.

We also want to clear that timer whenever the DOM produced by the `Clock` is removed. This is called “unmounting” in React.

We can declare special methods on the component class to run some code when a component mounts and unmounts:

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = { date: new Date() };
  }

  componentDidMount() {}
  componentWillUnmount() {}
  render() {
    return (
      <div>
        <h1>Hello, world!</h1> <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

These methods are called “lifecycle methods”.

The `componentDidMount()` method runs after the component output has been rendered to the DOM. This is a good place to set up a timer:

```
componentDidMount() {
  this.timerID = setInterval(() => this.tick(), 1000);
}
```

Note how we save the timer ID right on `this` (`this.timerID`).

While `this.props` is set up by React itself and `this.state` has a special meaning, you are free to add additional fields to the class manually if you need to store something that doesn't participate in the data flow (like a timer ID).

We will tear down the timer in the `componentWillUnmount()` lifecycle method:

```
componentWillUnmount() {
  clearInterval(this.timerID);
}
```

Finally, we will implement a method called `tick()` that the `Clock` component will run every second.

It will use `this.setState()` to schedule updates to the component local state:

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = { date: new Date() };
  }

  componentDidMount() {
    this.timerID = setInterval(() => this.tick(), 1000);
  }

  componentWillUnmount() {
    clearInterval(this.timerID);
  }

  tick() {
    this.setState({ date: new Date() });
  }

  render() {
    return (
      <div>
        <h1>Hello, world!</h1> <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}

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

Now the clock ticks every second.

Let's quickly recap what's going on and the order in which the methods are called:

1. When `<Clock />` is passed to `root.render()`, React calls the constructor of the `Clock` component. Since `Clock` needs to display the current time, it initializes `this.state` with an object including the current time. We will later update this state.
2. React then calls the `Clock` component's `render()` method. This is how React learns what should be displayed on the screen. React then updates the DOM to match the `Clock`'s render output.
3. When the `Clock` output is inserted in the DOM, React calls the `componentDidMount()` lifecycle method. Inside it, the `Clock` component asks the browser to set up a timer to call the component's `tick()` method once a second.
4. Every second the browser calls the `tick()` method. Inside it, the `Clock` component schedules a UI update by calling `setState()` with an object containing the current time. Thanks to the `setState()` call, React knows the state has changed, and calls the `render()` method again to learn what should be on the screen. This time, `this.state.date` in the `render()` method will be different, and so the render output will include the updated time. React updates the DOM accordingly.
5. If the `Clock` component is ever removed from the DOM, React calls the `componentWillUnmount()` lifecycle method so the timer is stopped.

Using State Correctly

There are three things you should know about `setState()`.

Do Not Modify State Directly

For example, this will not re-render a component:

```
// Wrong
this.state.comment = 'Hello';
```

Instead, use `setState()`:

```
// Correct
this.setState({comment: 'Hello'});
```

The only place where you can assign `this.state` is the constructor.

State Updates May Be Asynchronous

React may batch multiple `setState()` calls into a single update for performance.

Because `this.props` and `this.state` may be updated asynchronously, you should not rely on their values for calculating the next state.

For example, this code may fail to update the counter:

```
// Wrong
this.setState({
  counter: this.state.counter + this.props.increment,
});
```

To fix it, use a second form of `setState()` that accepts a function rather than an object. That function will receive the previous state as the first argument, and the props at the time the update is applied as the second argument:

```
// Correct
this.setState((state, props) => ({
  counter: state.counter + props.increment
}));
```

We used an arrow function above, but it also works with regular functions:

```
// Correct
this.setState(function(state, props) {
  return {
    counter: state.counter + props.increment
  };
});
```

State Updates are Merged

When you call `setState()`, React merges the object you provide into the current state.

For example, your state may contain several independent variables:

```
constructor(props) {
  super(props);
  this.state = {
    posts: [],
    comments: []
  };
}
```

Then you can update them independently with separate `setState()` calls:

```
componentDidMount() {
  fetchPosts().then((response) => {
    this.setState({
      posts: response.posts
    });
  });

  fetchComments().then((response) => {
    this.setState({
      comments: response.comments
    });
  });
}
```

The merging is shallow, so `this.setState({comments})` leaves `this.state.posts` intact, but completely replaces `this.state.comments`.

The Data Flows Down

Neither parent nor child components can know if a certain component is stateful or stateless, and they shouldn't care whether it is defined as a function or a class.

This is why state is often called local or encapsulated. It is not accessible to any component other than the one that owns and sets it.

A component may choose to pass its state down as props to its child components:

```
<FormattedDate date={this.state.date} />
```

The `FormattedDate` component would receive the `date` in its props and wouldn't know whether it came from the `Clock`'s state, from the `Clock`'s props, or was typed by hand:

```
function FormattedDate(props) {  
  return <h2>It is {props.date.toLocaleTimeString()}.</h2>;  
}
```

This is commonly called a “top-down” or “unidirectional” data flow. Any state is always owned by some specific component, and any data or UI derived from that state can only affect components “below” them in the tree.

If you imagine a component tree as a waterfall of props, each component's state is like an additional water source that joins it at an arbitrary point but also flows down.

To show that all components are truly isolated, we can create an `App` component that renders three `<Clock>`:

```
function App() {  
  return (  
    <div>  
      <Clock />  
      <Clock />  
      <Clock />  
    </div>  
  );  
}
```

Each `Clock` sets up its own timer and updates independently.

In React apps, whether a component is stateful or stateless is considered an implementation detail of the component that may change over time. You can use stateless components inside stateful components, and vice versa.

6. Handling Events

Handling events with React elements is very similar to handling events on DOM elements. There are some syntax differences:

- React events are named using camelCase, rather than lowercase.
- With JSX you pass a function as the event handler, rather than a string.

For example, the HTML:

```
<button onclick="activateLasers()">Activate Lasers</button>
```

is slightly different in React:

```
<button onClick={activateLasers}>Activate Lasers</button>;
```

Another difference is that you cannot return `false` to prevent default behavior in React. You must call `preventDefault` explicitly. For example, with plain HTML, to prevent the default form behavior of submitting, you can write:

```
<form onsubmit="console.log('You clicked submit.');
```

In React, this could instead be:

```
function Form() {
  function handleSubmit(e) {
    e.preventDefault();
    console.log('You clicked submit.');
```

Here, `e` is a synthetic event. React defines these synthetic events according to the [W3C spec](#), so you don't need to worry about cross-browser compatibility. React events do not work exactly the same as native events. See the [SyntheticEvent](#) reference guide to learn more.

When using React, you generally don't need to call `addEventListener` to add listeners to a DOM element after it is created. Instead, just provide a listener when the element is initially rendered.

When you define a component using an [ES6 class](#), a common pattern is for an event handler to be a method on the class. For example, this `Toggle` component renders a button that lets the user toggle between "ON" and "OFF" states:

```
class Toggle extends React.Component {
  constructor(props) {
    super(props);
    this.state = { isToggleOn: true };

    // This binding is necessary to make `this` work in the callback
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    this.setState((prevState) => ({ isToggleOn: !prevState.isToggleOn }));
  }
  render() {
    return (
      <button onClick={this.handleClick}> {this.state.isToggleOn ? 'ON' : 'OFF'}</button>
    );
  }
}
```

You have to be careful about the meaning of `this` in JSX callbacks. In JavaScript, class methods are not [bound](#) by default. If you forget to bind `this.handleClick` and pass it to `onClick`, `this` will be `undefined` when the function is actually called.

This is not React-specific behavior; it is a part of [how functions work in JavaScript](#). Generally, if you refer to a method without `()` after it, such as `onClick={this.handleClick}`, you should bind that method.

If calling `bind` annoys you, there are two ways you can get around this. You can use [public class fields syntax](#) to correctly bind callbacks:

```
class LoggingButton extends React.Component {
  // This syntax ensures `this` is bound within handleClick.
  handleClick = () => {
    console.log('this is:', this);
  };
  render() {
    return <button onClick={this.handleClick}> Click me</button>;
  }
}
```

This syntax is enabled by default in [Create React App](#).

If you aren't using class fields syntax, you can use an [arrow function](#) in the callback:

```
class LoggingButton extends React.Component {
  handleClick() {
    console.log('this is:', this);
  }

  render() {
    // This syntax ensures `this` is bound within handleClick
    return <button onClick={() => this.handleClick()}> Click me</button>;
  }
}
```

The problem with this syntax is that a different callback is created each time the `LoggingButton` renders. In most cases, this is fine. However, if this callback is passed as a prop to lower components, those components might do an extra re-rendering. We generally recommend binding in the constructor or using the class fields syntax, to avoid this sort of performance problem.

Passing Arguments to Event Handlers

Inside a loop, it is common to want to pass an extra parameter to an event handler. For example, if `id` is the row ID, either of the following would work:

```
<button onClick={(e) => this.deleteRow(id, e)}>Delete Row</button>
<button onClick={this.deleteRow.bind(this, id)}>Delete Row</button>
```

The above two lines are equivalent, and use [arrow functions](#) and [Function.prototype.bind](#) respectively.

In both cases, the `e` argument representing the React event will be passed as a second argument after the ID. With an arrow function, we have to pass it explicitly, but with `bind` any further arguments are automatically forwarded.

7. Conditional Rendering

In React, you can create distinct components that encapsulate behavior you need. Then, you can render only some of them, depending on the state of your application.

Conditional rendering in React works the same way conditions work in JavaScript. Use JavaScript operators like `if` or the [conditional operator](#) to create elements representing the current state, and let React update the UI to match them.

Consider these two components:

```
function UserGreeting(props) {
  return <h1>Welcome back!</h1>;
}

function GuestGreeting(props) {
  return <h1>Please sign up.</h1>;
}
```

We'll create a `Greeting` component that displays either of these components depending on whether a user is logged in:

```
function Greeting(props) {
  const isLoggedIn = props.isLoggedIn;
  if (isLoggedIn) {
    return <UserGreeting />;
  }
  return <GuestGreeting />;
}

const root = ReactDOM.createRoot(document.getElementById('root'));
// Try changing to isLoggedIn={true}:
root.render(<Greeting isLoggedIn={false} />);
```

This example renders a different greeting depending on the value of `isLoggedIn` prop.

Element Variables

You can use variables to store elements. This can help you conditionally render a part of the component while the rest of the output doesn't change.

Consider these two new components representing Logout and Login buttons:

```
function LoginButton(props) {
  return <button onClick={props.onClick}>Login</button>;
}

function LogoutButton(props) {
  return <button onClick={props.onClick}>Logout</button>;
}
```

In the example below, we will create a stateful component called `LoginControl`.

It will render either `<LoginButton />` or `<LogoutButton />` depending on its current state. It will also render a `<Greeting />` from the previous example:

```
class LoginControl extends React.Component {
  constructor(props) {
    super(props);
    this.handleLoginClick = this.handleLoginClick.bind(this);
    this.handleLogoutClick = this.handleLogoutClick.bind(this);
    this.state = { isLoggedIn: false };
  }

  handleLoginClick() {
    this.setState({ isLoggedIn: true });
  }

  handleLogoutClick() {
    this.setState({ isLoggedIn: false });
  }

  render() {
    const isLoggedIn = this.state.isLoggedIn;
    let button;
    if (isLoggedIn) {
      button = <LogoutButton onClick={this.handleLogoutClick} />;
    } else {
      button = <LoginButton onClick={this.handleLoginClick} />;
    }
    return (
      <div>
        <Greeting isLoggedIn={isLoggedIn} /> {button}
      </div>
    );
  }
}

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

While declaring a variable and using an `if` statement is a fine way to conditionally render a component, sometimes you might want to use a shorter syntax. There are a few ways to inline conditions in JSX, explained below.

Inline If with Logical `&&` Operator

You may embed expressions in JSX by wrapping them in curly braces. This includes the JavaScript logical `&&` operator. It can be handy for conditionally including an element:

```
function Mailbox(props) {
  const unreadMessages = props.unreadMessages;
  return (
    <div>
      <h1>Hello!</h1> {unreadMessages.length > 0 && <h2> You have {unreadMessages.length} unread messages. </h2>}
    </div>
  );
}

const messages = ['React', 'Re: React', 'Re:Re: React'];

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Mailbox unreadMessages={messages} />);
```

It works because in JavaScript, `true && expression` always evaluates to `expression`, and `false && expression` always evaluates to `false`.

Therefore, if the condition is `true`, the element right after `&&` will appear in the output. If it is `false`, React will ignore and skip it.

Note that returning a falsy expression will still cause the element after `&&` to be skipped but will return the falsy expression. In the example below, `<div>0</div>` will be returned by the render method.

```
render() {
  const count = 0;
  return <div> {count} && <h1>Messages: {count}</h1> </div>;
}
```

Inline If-Else with Conditional Operator

Another method for conditionally rendering elements inline is to use the JavaScript conditional operator `condition ? true : false`.

In the example below, we use it to conditionally render a small block of text.

```
render() {
  const isLoggedIn = this.state.isLoggedIn;
  return (
    <div>
      The user is <b>{isLoggedIn ? 'currently' : 'not'}</b> logged in.
    </div>
  );
}
```

It can also be used for larger expressions although it is less obvious what's going on:

```
render() {
  const isLoggedIn = this.state.isLoggedIn;
  return (
    <div>
      {isLoggedIn ? (
        <LogoutButton onClick={this.handleLogoutClick} />
      ) : (
        <LoginButton onClick={this.handleLoginClick} />
      )}
    </div>
  );
}
```

Just like in JavaScript, it is up to you to choose an appropriate style based on what you and your team consider more readable. Also remember that whenever conditions become too complex, it might be a good time to [extract a component](#).

Preventing Component from Rendering

In rare cases you might want a component to hide itself even though it was rendered by another component. To do this return `null` instead of its render output.

In the example below, the `<WarningBanner />` is rendered depending on the value of the prop called `warn`. If the value of the prop is `false`, then the component does not render:

```
function WarningBanner(props) {
  if (!props.warn) {
    return null;
  }
  return <div className="warning"> Warning!</div>;
}

class Page extends React.Component {
  constructor(props) {
```

```

super(props);
this.state = { showWarning: true };
this.handleClick = this.handleClick.bind(this);
}

handleToggleClick() {
  this.setState((state) => ({
    showWarning: !state.showWarning
  }));
}

render() {
  return (
    <div>
      <WarningBanner warn={this.state.showWarning} />
      <button onClick={this.handleClick}> {this.state.showWarning ? 'Hide' : 'Show'}</button>
    </div>
  );
}
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Page />);

```

Returning `null` from a component's `render` method does not affect the firing of the component's lifecycle methods. For instance `componentDidUpdate` will still be called.

8. Lists and Keys

First, let's review how you transform lists in JavaScript.

Given the code below, we use the `map()` function to take an array of `numbers` and double their values. We assign the new array returned by `map()` to the variable `doubled` and log it:

```

const numbers = [1, 2, 3, 4, 5];
const doubled = numbers.map((number) => number * 2);
console.log(doubled); // (5) [2, 4, 6, 8, 10]

```

This code logs `[2, 4, 6, 8, 10]` to the console.

In React, transforming arrays into lists of elements is nearly identical.

Rendering Multiple Components

You can build collections of elements and include them in JSX using curly braces `{ }`.

Below, we loop through the `numbers` array using the JavaScript `map()` function. We return a `` element for each item. Finally, we assign the resulting array of elements to `listItems`:

```

const numbers = [1, 2, 3, 4, 5];
const listItems = numbers.map((number) => <li>{number}</li>);

```

Then, we can include the entire `listItems` array inside a `` element:

```

<ul>{listItems}</ul>

```

This code displays a bullet list of numbers between 1 and 5.

Basic List Component

Usually you would render lists inside a component.

We can refactor the previous example into a component that accepts an array of `numbers` and outputs a list of elements.

```

function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) => <li>{number}</li>);
  return <ul>{listItems}</ul>;
}

```

```

}

const numbers = [1, 2, 3, 4, 5];
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<NumberList numbers={numbers} />);

```

When you run this code, you'll be given a warning that a key should be provided for list items. A "key" is a special string attribute you need to include when creating lists of elements. We'll discuss why it's important in the next section.

Let's assign a `key` to our list items inside `numbers.map()` and fix the missing key issue.

```

function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) => <li key={number.toString()}> {number}</li>);
  return <ul>{listItems}</ul>;
}

```

Keys

Keys help React identify which items have changed, are added, or are removed. Keys should be given to the elements inside the array to give the elements a stable identity:

```

const numbers = [1, 2, 3, 4, 5];
const listItems = numbers.map((number) => <li key={number.toString()}> {number}</li>);

```

The best way to pick a key is to use a string that uniquely identifies a list item among its siblings. Most often you would use IDs from your data as keys:

```

const todoItems = todos.map((todo) => <li key={todo.id}> {todo.text}</li>);

```

When you don't have stable IDs for rendered items, you may use the item index as a key as a last resort:

```

const todoItems = todos.map((todo, index) => (
  // Only do this if items have no stable IDs
  <li key={index}> {todo.text}</li>
));

```

We don't recommend using indexes for keys if the order of items may change. This can negatively impact performance and may cause issues with component state. Check out Robin Pokorny's article for an [in-depth explanation on the negative impacts of using an index as a key](#). If you choose not to assign an explicit key to list items then React will default to using indexes as keys.

Here is an [in-depth explanation about why keys are necessary](#) if you're interested in learning more.

Extracting Components with Keys

Keys only make sense in the context of the surrounding array.

For example, if you [extract](#) a `Listitem` component, you should keep the key on the `<Listitem />` elements in the array rather than on the `` element in the `Listitem` itself.

Example: Incorrect Key Usage

```

function ListItem(props) {
  const value = props.value;
  return (
    // Wrong! There is no need to specify the key here:
    <li key={value.toString()}> {value}</li>
  );
}

function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) => (
    // Wrong! The key should have been specified here:

```

```

    <ListItem value={number} />
  ));
  return <ul> {listItems}</ul>;
}

```

Example: Correct Key Usage

```

function ListItem(props) {
  // Correct! There is no need to specify the key here:
  return <li>{props.value}</li>;
}

function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) => (
    // Correct! Key should be specified inside the array.
    <ListItem key={number.toString()} value={number} />
  ));
  return <ul> {listItems}</ul>;
}

```

A good rule of thumb is that elements inside the `map()` call need keys.

Keys Must Only Be Unique Among Siblings

Keys used within arrays should be unique among their siblings. However, they don't need to be globally unique. We can use the same keys when we produce two different arrays:

```

function Blog(props) {
  const sidebar = (
    <ul>
      {props.posts.map((post) => (
        <li key={post.id}> {post.title}</li>
      ))}
    </ul>
  );
  const content = props.posts.map((post) => (
    <div key={post.id}>
      <h3>{post.title}</h3> <p>{post.content}</p>
    </div>
  ));
  return (
    <div>
      {sidebar} <hr /> {content}
    </div>
  );
}

const posts = [
  { id: 1, title: 'Hello World', content: 'Welcome to learning React!' },
  { id: 2, title: 'Installation', content: 'You can install React from npm.' }
];

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Blog posts={posts} />);

```

Keys serve as a hint to React but they don't get passed to your components. If you need the same value in your component, pass it explicitly as a prop with a different name:

```

const content = posts.map((post) => <Post key={post.id} id={post.id} title={post.title} />);

```

With the example above, the `Post` component can read `props.id`, but not `props.key`.

Embedding map() in JSX

In the examples above we declared a separate `listItems` variable and included it in JSX:

```

function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) => {

```

```

    return <ListItem key={number.toString()} value={number} />
  });
  return <ul> {listItems}</ul>;
}

```

JSX allows embedding any expression in curly braces so we could inline the `map()` result:

```

function NumberList(props) {
  const numbers = props.numbers;
  return (
    <ul>
      {numbers.map((number) => (
        <ListItem key={number.toString()} value={number} />
      ))}
    </ul>
  );
}

```

Sometimes this results in clearer code, but this style can also be abused. Like in JavaScript, it is up to you to decide whether it is worth extracting a variable for readability. Keep in mind that if the `map()` body is too nested, it might be a good time to extract a component.

9. Forms

HTML form elements work a bit differently from other DOM elements in React, because form elements naturally keep some internal state. For example, this form in plain HTML accepts a single name:

```

<form>
  <label>
    Name:
    <input type="text" name="name" />
  </label>
  <input type="submit" value="Submit" />
</form>;

```

This form has the default HTML form behavior of browsing to a new page when the user submits the form. If you want this behavior in React, it just works. But in most cases, it's convenient to have a JavaScript function that handles the submission of the form and has access to the data that the user entered into the form. The standard way to achieve this is with a technique called “controlled components”.

Controlled Components

In HTML, form elements such as `<input>`, `<textarea>`, and `<select>` typically maintain their own state and update it based on user input. In React, mutable state is typically kept in the state property of components, and only updated with `setState()`.

We can combine the two by making the React state be the “single source of truth”. Then the React component that renders a form also controls what happens in that form on subsequent user input. An input form element whose value is controlled by React in this way is called a “controlled component”.

For example, if we want to make the previous example log the name when it is submitted, we can write the form as a controlled component:

```

class NameForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = { value: '' };
    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  handleChange(event) {
    this.setState({ value: event.target.value });
  }

  handleSubmit(event) {
    alert('A name was submitted: ' + this.state.value);
    event.preventDefault();
  }

  render() {

```

```

    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Name:
          <input type="text" value={this.state.value} onChange={this.handleChange} />
        </label>
        <input type="submit" value="Submit" />
      </form>
    );
  }
}

```

Since the `value` attribute is set on our form element, the displayed value will always be `this.state.value`, making the React state the source of truth. Since `handleChange` runs on every keystroke to update the React state, the displayed value will update as the user types.

With a controlled component, the input's value is always driven by the React state. While this means you have to type a bit more code, you can now pass the value to other UI elements too, or reset it from other event handlers.

The textarea Tag

In HTML, a `<textarea>` element defines its text by its children:

```

<textarea>
  Hello there, this is some text in a text area
</textarea>

```

In React, a `<textarea>` uses a `value` attribute instead. This way, a form using a `<textarea>` can be written very similarly to a form that uses a single-line input:

```

class EssayForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = { value: 'Please write an essay about your favorite DOM element.' };
    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  handleChange(event) {
    this.setState({ value: event.target.value });
  }
  handleSubmit(event) {
    alert('An essay was submitted: ' + this.state.value);
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Essay:
          <textarea value={this.state.value} onChange={this.handleChange} />
        </label>
        <input type="submit" value="Submit" />
      </form>
    );
  }
}

```

Notice that `this.state.value` is initialized in the constructor, so that the text area starts off with some text in it.

The select Tag

In HTML, `<select>` creates a drop-down list. For example, this HTML creates a drop-down list of flavors:

```

<select>
  <option value="grapefruit">Grapefruit</option>
  <option value="lime">Lime</option>
  <option selected value="coconut">
    Coconut
  </option>
</select>

```



```
<option value="mango">Mango</option>
</select>
```

Note that the Coconut option is initially selected, because of the `selected` attribute. React, instead of using this `selected` attribute, uses a `value` attribute on the root `select` tag. This is more convenient in a controlled component because you only need to update it in one place. For example:

```
class FlavorForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = { value: 'coconut' };
    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  handleChange(event) {
    this.setState({ value: event.target.value });
  }

  handleSubmit(event) {
    alert('Your favorite flavor is: ' + this.state.value);
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Pick your favorite flavor:
          <select value={this.state.value} onChange={this.handleChange}>
            <option value="grapefruit">Grapefruit</option> <option value="lime">Lime</option>
            <option value="coconut">Coconut</option> <option value="mango">Mango</option>
          </select>
        </label>
        <input type="submit" value="Submit" />
      </form>
    );
  }
}
```

Try it on CodePen

Overall, this makes it so that `<input type="text">`, `<textarea>`, and `<select>` all work very similarly - they all accept a `value` attribute that you can use to implement a controlled component.

Note

You can pass an array into the `value` attribute, allowing you to select multiple options in a `select` tag:

```
<select multiple={true} value={['B', 'C']>
```

The file input Tag

In HTML, an `<input type="file">` lets the user choose one or more files from their device storage to be uploaded to a server or manipulated by JavaScript via the [File API](#).

```
<input type="file" />
```

Because its value is read-only, it is an **uncontrolled** component in React. It is discussed together with other uncontrolled components [later in the documentation](#).

Handling Multiple Inputs

When you need to handle multiple controlled `input` elements, you can add a `name` attribute to each element and let the handler function choose what to do based on the value of `event.target.name`.

For example:

```
class Reservation extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      isGoing: true,
      numberOfGuests: 2
    };

    this.handleChange = this.handleChange.bind(this);
  }

  handleChange(event) {
    const target = event.target;
    const value = target.type === 'checkbox' ? target.checked : target.value;
    const name = target.name;
    this.setState({
      [name]: value
    });
  }

  render() {
    return (
      <form>
        <label>
          Is going:
          <input name="isGoing" type="checkbox" checked={this.state.isGoing} onChange={this.handleChange} />
        </label>
        <br />
        <label>
          Number of guests:
          <input
            name="numberOfGuests"
            type="number"
            value={this.state.numberOfGuests}
            onChange={this.handleChange}
          />
        </label>
      </form>
    );
  }
}
```

Note how we used the ES6 computed property name syntax to update the state key corresponding to the given input name:

```
this.setState({
  [name]: value
});
```

It is equivalent to this ES5 code:

```
var partialState = {};
partialState[name] = value; this.setState(partialState);
```

Also, since `setState()` automatically merges a partial state into the current state, we only needed to call it with the changed parts.

Controlled Input Null Value

Specifying the `value` prop on a controlled component prevents the user from changing the input unless you desire so. If you've specified a `value` but the input is still editable, you may have accidentally set `value` to `undefined` or `null`.

The following code demonstrates this. (The input is locked at first but becomes editable after a short delay.)

```
ReactDOM.createRoot(mountNode).render(<input value="hi" />);

setTimeout(function() {
  ReactDOM.createRoot(mountNode).render(<input value={null} />);
}, 1000);
```

Alternatives to Controlled Components

It can sometimes be tedious to use controlled components, because you need to write an event handler for every way your data can change and pipe all of the input state through a React component. This can become particularly annoying when you are converting a preexisting codebase to React, or integrating a React application with a non-React library. In these situations, you might want to check out [uncontrolled components](#), an alternative technique for implementing input forms.

Fully-Fledged Solutions

If you're looking for a complete solution including validation, keeping track of the visited fields, and handling form submission, [Formik](#) is one of the popular choices. However, it is built on the same principles of controlled components and managing state — so don't neglect to learn them.

10. Lifting State Up

Often, several components need to reflect the same changing data. We recommend lifting the shared state up to their closest common ancestor. Let's see how this works in action.

In this section, we will create a temperature calculator that calculates whether the water would boil at a given temperature.

We will start with a component called `BoilingVerdict`. It accepts the `celsius` temperature as a prop, and prints whether it is enough to boil the water:

```
function BoilingVerdict(props) {
  if (props.celsius >= 100) {
    return <p>The water would boil.</p>;
  }
  return <p>The water would not boil.</p>;
}
```

Next, we will create a component called `Calculator`. It renders an `<input>` that lets you enter the temperature, and keeps its value in `this.state.temperature`.

Additionally, it renders the `BoilingVerdict` for the current input value.

```
class Calculator extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = { temperature: '' };
  }

  handleChange(e) {
    this.setState({ temperature: e.target.value });
  }

  render() {
    const temperature = this.state.temperature;
    return (
      <fieldset>
        <legend>Enter temperature in Celsius:</legend> <input value={temperature} onChange={this.handleChange} />
        <BoilingVerdict celsius={parseFloat(temperature)} />
      </fieldset>
    );
  }
}
```

Adding a Second Input

Our new requirement is that, in addition to a Celsius input, we provide a Fahrenheit input, and they are kept in sync.

We can start by extracting a `TemperatureInput` component from `Calculator`. We will add a new `scale` prop to it that can either be `"C"` or `"F"`:

```
const scaleNames = { c: 'Celsius', f: 'Fahrenheit' };
class TemperatureInput extends React.Component {
  constructor(props) {
    super(props);
```

```

    this.handleChange = this.handleChange.bind(this);
    this.state = { temperature: '' };
  }

  handleChange(e) {
    this.setState({ temperature: e.target.value });
  }

  render() {
    const temperature = this.state.temperature;
    const scale = this.props.scale;
    return (
      <fieldset>
        <legend>Enter temperature in {scaleNames[scale]}:</legend>
        <input value={temperature} onChange={this.handleChange} />
      </fieldset>
    );
  }
}

```

We can now change the `Calculator` to render two separate temperature inputs:

```

class Calculator extends React.Component {
  render() {
    return (
      <div>
        <TemperatureInput scale="c" /> <TemperatureInput scale="f" />
      </div>
    );
  }
}

```

We have two inputs now, but when you enter the temperature in one of them, the other doesn't update. This contradicts our requirement: we want to keep them in sync.

We also can't display the `BoilingVerdict` from `Calculator`. The `Calculator` doesn't know the current temperature because it is hidden inside the `TemperatureInput`.

Writing Conversion Functions

First, we will write two functions to convert from Celsius to Fahrenheit and back:

```

function toCelsius(fahrenheit) {
  return (fahrenheit - 32) * 5 / 9;
}

function toFahrenheit(celsius) {
  return (celsius * 9 / 5) + 32;
}

```

These two functions convert numbers. We will write another function that takes a string `temperature` and a converter function as arguments and returns a string. We will use it to calculate the value of one input based on the other input.

It returns an empty string on an invalid `temperature`, and it keeps the output rounded to the third decimal place:

```

function tryConvert(temperature, convert) {
  const input = parseFloat(temperature);
  if (Number.isNaN(input)) {
    return '';
  }
  const output = convert(input);
  const rounded = Math.round(output * 1000) / 1000;
  return rounded.toString();
}

```

For example, `tryConvert('abc', toCelsius)` returns an empty string, and `tryConvert('10.22', toFahrenheit)` returns `'50.396'`.

Lifting State Up

Currently, both `TemperatureInput` components independently keep their values in the local state:

```

class TemperatureInput extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = { temperature: '' };
  }

  handleChange(e) {
    this.setState({ temperature: e.target.value });
  }

  render() {
    const temperature = this.state.temperature; // ...
  }
}

```

However, we want these two inputs to be in sync with each other. When we update the Celsius input, the Fahrenheit input should reflect the converted temperature, and vice versa.

In React, sharing state is accomplished by moving it up to the closest common ancestor of the components that need it. This is called “lifting state up”. We will remove the local state from the `TemperatureInput` and move it into the `Calculator` instead.

If the `Calculator` owns the shared state, it becomes the “source of truth” for the current temperature in both inputs. It can instruct them both to have values that are consistent with each other. Since the props of both `TemperatureInput` components are coming from the same parent `Calculator` component, the two inputs will always be in sync.

Let’s see how this works step by step.

First, we will replace `this.state.temperature` with `this.props.temperature` in the `TemperatureInput` component. For now, let’s pretend `this.props.temperature` already exists, although we will need to pass it from the `Calculator` in the future:

```

render() {
  // Before: const temperature = this.state.temperature;
  const temperature = this.props.temperature; // ...
}

```

We know that props are read-only. When the `temperature` was in the local state, the `TemperatureInput` could just call `this.setState()` to change it. However, now that the `temperature` is coming from the parent as a prop, the `TemperatureInput` has no control over it.

In React, this is usually solved by making a component “controlled”. Just like the DOM `<input>` accepts both a `value` and an `onChange` prop, so can the custom `TemperatureInput` accept both `temperature` and `onTemperatureChange` props from its parent `Calculator`.

Now, when the `TemperatureInput` wants to update its temperature, it calls `this.props.onTemperatureChange`:

```

handleChange(e) {
  // Before: this.setState({temperature: e.target.value});
  this.props.onTemperatureChange(e.target.value); // ...
}

```

Note:

There is no special meaning to either `temperature` or `onTemperatureChange` prop names in custom components. We could have called them anything else, like name them `value` and `onChange` which is a common convention.

The `onTemperatureChange` prop will be provided together with the `temperature` prop by the parent `Calculator` component. It will handle the change by modifying its own local state, thus re-rendering both inputs with the new values. We will look at the new `Calculator` implementation very soon.

Before diving into the changes in the `Calculator`, let’s recap our changes to the `TemperatureInput` component. We have removed the local state from it, and instead of reading `this.state.temperature`, we now read `this.props.temperature`. Instead of calling `this.setState()` when we want to make a change, we now call `this.props.onTemperatureChange()`, which will be provided by the `Calculator`:

```

class TemperatureInput extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
  }

  handleChange(e) {
    this.props.onTemperatureChange(e.target.value);
  }
}

```

```

    }

    render() {
      const temperature = this.props.temperature;
      const scale = this.props.scale;
      return (
        <fieldset>
          <legend>Enter temperature in {scaleNames[scale]}:</legend>
          <input value={temperature} onChange={this.handleChange} />
        </fieldset>
      );
    }
  }
}

```

Now let's turn to the `Calculator` component.

We will store the current input's `temperature` and `scale` in its local state. This is the state we “lifted up” from the inputs, and it will serve as the “source of truth” for both of them. It is the minimal representation of all the data we need to know in order to render both inputs.

For example, if we enter 37 into the Celsius input, the state of the `Calculator` component will be:

```

{
  temperature: '37',
  scale: 'c'
}

```

If we later edit the Fahrenheit field to be 212, the state of the `Calculator` will be:

```

{
  temperature: '212',
  scale: 'f'
}

```

We could have stored the value of both inputs but it turns out to be unnecessary. It is enough to store the value of the most recently changed input, and the scale that it represents. We can then infer the value of the other input based on the current `temperature` and `scale` alone.

The inputs stay in sync because their values are computed from the same state:

```

class Calculator extends React.Component {
  constructor(props) {
    super(props);
    this.handleCelsiusChange = this.handleCelsiusChange.bind(this);
    this.handleFahrenheitChange = this.handleFahrenheitChange.bind(this);
    this.state = { temperature: '', scale: 'c' };
  }

  handleCelsiusChange(temperature) {
    this.setState({ scale: 'c', temperature });
  }

  handleFahrenheitChange(temperature) {
    this.setState({ scale: 'f', temperature });
  }

  render() {
    const scale = this.state.scale;
    const temperature = this.state.temperature;
    const celsius = scale === 'f' ? tryConvert(temperature, toCelsius) : temperature;
    const fahrenheit = scale === 'c' ? tryConvert(temperature, toFahrenheit) : temperature;
    return (
      <div>
        <TemperatureInput
          scale="c"
          temperature={celsius}
          onTemperatureChange={this.handleCelsiusChange}
        />
        <TemperatureInput
          scale="f"
          temperature={fahrenheit}
          onTemperatureChange={this.handleFahrenheitChange} />
        <BoilingVerdict celsius={parseFloat(celsius)} />
      </div>
    );
  }
}

```

```
}  
}
```

Now, no matter which input you edit, `this.state.temperature` and `this.state.scale` in the `Calculator` get updated. One of the inputs gets the value as is, so any user input is preserved, and the other input value is always recalculated based on it.

Let's recap what happens when you edit an input:

- React calls the function specified as `onChange` on the DOM `<input>`. In our case, this is the `handleChange` method in the `TemperatureInput` component.
- The `handleChange` method in the `TemperatureInput` component calls `this.props.onTemperatureChange()` with the new desired value. Its props, including `onTemperatureChange`, were provided by its parent component, the `Calculator`.
- When it previously rendered, the `Calculator` had specified that `onTemperatureChange` of the Celsius `TemperatureInput` is the `Calculator`'s `handleCelsiusChange` method, and `onTemperatureChange` of the Fahrenheit `TemperatureInput` is the `Calculator`'s `handleFahrenheitChange` method. So either of these two `Calculator` methods gets called depending on which input we edited.
- Inside these methods, the `Calculator` component asks React to re-render itself by calling `this.setState()` with the new input value and the current scale of the input we just edited.
- React calls the `Calculator` component's `render` method to learn what the UI should look like. The values of both inputs are recomputed based on the current temperature and the active scale. The temperature conversion is performed here.
- React calls the `render` methods of the individual `TemperatureInput` components with their new props specified by the `Calculator`. It learns what their UI should look like.
- React calls the `render` method of the `BoilingVerdict` component, passing the temperature in Celsius as its props.
- React DOM updates the DOM with the boiling verdict and to match the desired input values. The input we just edited receives its current value, and the other input is updated to the temperature after conversion.

Every update goes through the same steps so the inputs stay in sync.

Lessons Learned

There should be a single “source of truth” for any data that changes in a React application. Usually, the state is first added to the component that needs it for rendering. Then, if other components also need it, you can lift it up to their closest common ancestor. Instead of trying to sync the state between different components, you should rely on the [top-down data flow](#).

Lifting state involves writing more “boilerplate” code than two-way binding approaches, but as a benefit, it takes less work to find and isolate bugs. Since any state “lives” in some component and that component alone can change it, the surface area for bugs is greatly reduced. Additionally, you can implement any custom logic to reject or transform user input.

If something can be derived from either props or state, it probably shouldn't be in the state. For example, instead of storing both `celsiusValue` and `fahrenheitValue`, we store just the last edited `temperature` and its `scale`. The value of the other input can always be calculated from them in the `render()` method. This lets us clear or apply rounding to the other field without losing any precision in the user input.

When you see something wrong in the UI, you can use [React Developer Tools](#) to inspect the props and move up the tree until you find the component responsible for updating the state. This lets you trace the bugs to their source:

11. Composition vs Inheritance

React has a powerful composition model, and we recommend using composition instead of inheritance to reuse code between components.

In this section, we will consider a few problems where developers new to React often reach for inheritance, and show how we can solve them with composition.

Containment

Some components don't know their children ahead of time. This is especially common for components like `Sidebar` or `Dialog` that represent generic “boxes”.

We recommend that such components use the special `children` prop to pass children elements directly into their output:

```
function FancyBorder(props) {
  return (
    <div className='FancyBorder FancyBorder-' + props.color>
      {props.children}
    </div>
  )
}
```

This lets other components pass arbitrary children to them by nesting the JSX:

```
function WelcomeDialog() {
  return (
    <FancyBorder color="blue">
      <h1 className="Dialog-title"> Welcome </h1>
      <p className="Dialog-message"> Thank you for visiting our spacecraft! </p>
    </FancyBorder>
  );
}
```

Anything inside the `<FancyBorder>` JSX tag gets passed into the `FancyBorder` component as a `children` prop. Since `FancyBorder` renders `{props.children}` inside a `<div>`, the passed elements appear in the final output.

While this is less common, sometimes you might need multiple “holes” in a component. In such cases you may come up with your own convention instead of using `children`:

```
function SplitPane(props) {
  return (
    <div className="SplitPane">
      <div className="SplitPane-left"> {props.left} </div> <div className="SplitPane-right"> {props.right} </div>
    </div>
  );
}

function App() {
  return <SplitPane left=<Contacts /> right=<Chat /> />;
}
```

React elements like `<Contacts />` and `<Chat />` are just objects, so you can pass them as props like any other data. This approach may remind you of “slots” in other libraries but there are no limitations on what you can pass as props in React.

Specialization

Sometimes we think about components as being “special cases” of other components. For example, we might say that a `WelcomeDialog` is a special case of `Dialog`.

In React, this is also achieved by composition, where a more “specific” component renders a more “generic” one and configures it with props:

```
function Dialog(props) {
  return (
    <FancyBorder color="blue">
      <h1 className="Dialog-title"> {props.title} </h1> <p className="Dialog-message"> {props.message} </p>
    </FancyBorder>
  );
}

function WelcomeDialog() {
  return <Dialog title="Welcome" message="Thank you for visiting our spacecraft!" />;
}
```

Composition works equally well for components defined as classes:

```
function Dialog(props) {
  return (
    <FancyBorder color="blue">
      <h1 className="Dialog-title"> {props.title}</h1> <p className="Dialog-message"> {props.message}</p>
      {props.children}
    </FancyBorder>
  );
}
```



```

class SignUpDialog extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.handleSignUp = this.handleSignUp.bind(this);
    this.state = { login: '' };
  }

  render() {
    return (
      <Dialog title="Mars Exploration Program" message="How should we refer to you?">
        <input value={this.state.login} onChange={this.handleChange} />
        <button onClick={this.handleSignUp}> Sign Me Up! </button>
      </Dialog>
    );
  }

  handleChange(e) {
    this.setState({ login: e.target.value });
  }

  handleSignUp() {
    alert('Welcome aboard, ${this.state.login}!');
  }
}

```

So What About Inheritance?

At Facebook, we use React in thousands of components, and we haven't found any use cases where we would recommend creating component inheritance hierarchies.

Props and composition give you all the flexibility you need to customize a component's look and behavior in an explicit and safe way. Remember that components may accept arbitrary props, including primitive values, React elements, or functions.

If you want to reuse non-UI functionality between components, we suggest extracting it into a separate JavaScript module. The components may import it and use that function, object, or class, without extending it.

12. Thinking in React

React is, in our opinion, the premier way to build big, fast Web apps with JavaScript. It has scaled very well for us at Facebook and Instagram.

One of the many great parts of React is how it makes you think about apps as you build them. In this document, we'll walk you through the thought process of building a searchable product data table using React.

Start With A Mock

Imagine that we already have a JSON API and a mock from our designer. The mock looks like this:

<input type="text" value="Search..."/>	
<input type="checkbox"/> Only show products in stock	
Name	Price
Sporting Goods	
Football	\$49.99
Baseball	\$9.99
Basketball	\$29.99
Electronics	
iPod Touch	\$99.99
iPhone 5	\$399.99
Nexus 7	\$199.99

Our JSON API returns some data that looks like this:

```

[
  {category: "Sporting Goods", price: "$49.99", stocked: true, name: "Football"},
  {category: "Sporting Goods", price: "$9.99", stocked: true, name: "Baseball"},
  {category: "Sporting Goods", price: "$29.99", stocked: false, name: "Basketball"},
]

```

```

{category: "Electronics", price: "$99.99", stocked: true, name: "iPod Touch"},
{category: "Electronics", price: "$399.99", stocked: false, name: "iPhone 5"},
{category: "Electronics", price: "$199.99", stocked: true, name: "Nexus 7"}
];

```

Step 1: Break The UI Into A Component Hierarchy

The first thing you'll want to do is to draw boxes around every component (and subcomponent) in the mock and give them all names. If you're working with a designer, they may have already done this, so go talk to them! Their Photoshop layer names may end up being the names of your React components!

But how do you know what should be its own component? Use the same techniques for deciding if you should create a new function or object. One such technique is the single responsibility principle, that is, a component should ideally only do one thing. If it ends up growing, it should be decomposed into smaller subcomponents.

Since you're often displaying a JSON data model to a user, you'll find that if your model was built correctly, your UI (and therefore your component structure) will map nicely. That's because UI and data models tend to adhere to the same *information architecture*. Separate your UI into components, where each component matches one piece of your data model.



You'll see here that we have five components in our app. We've italicized the data each component represents. The numbers in the image correspond to the numbers below.

1. `FilterableProductTable` (orange): contains the entirety of the example
2. `SearchBar` (blue): receives all *user input*
3. `ProductTable` (green): displays and filters the *data collection* based on *user input*
4. `ProductCategoryRow` (turquoise): displays a heading for each *category*
5. `ProductRow` (red): displays a row for each *product*

If you look at `ProductTable`, you'll see that the table header (containing the "Name" and "Price" labels) isn't its own component. This is a matter of preference, and there's an argument to be made either way. For this example, we left it as part of `ProductTable` because it is part of rendering the *data collection* which is `ProductTable`'s responsibility. However, if this header grows to be complex (e.g., if we were to add affordances for sorting), it would certainly make sense to make this its own `ProductTableHeader` component.

Now that we've identified the components in our mock, let's arrange them into a hierarchy. Components that appear within another component in the mock should appear as a child in the hierarchy:

- `FilterableProductTable`
 - `SearchBar`
 - `ProductTable`
 - `ProductCategoryRow`

- `ProductRow`

Step 2: Build A Static Version in React

See the Pen [Thinking In React: Step 2](#) on [CodePen](#).

Now that you have your component hierarchy, it's time to implement your app. The easiest way is to build a version that takes your data model and renders the UI but has no interactivity. It's best to decouple these processes because building a static version requires a lot of typing and no thinking, and adding interactivity requires a lot of thinking and not a lot of typing. We'll see why.

To build a static version of your app that renders your data model, you'll want to build components that reuse other components and pass data using *props*. *props* are a way of passing data from parent to child. If you're familiar with the concept of *state*, **don't use state at all** to build this static version. State is reserved only for interactivity, that is, data that changes over time. Since this is a static version of the app, you don't need it.

You can build top-down or bottom-up. That is, you can either start with building the components higher up in the hierarchy (i.e. starting with `FilterableProductTable`) or with the ones lower in it (`ProductRow`). In simpler examples, it's usually easier to go top-down, and on larger projects, it's easier to go bottom-up and write tests as you build.

At the end of this step, you'll have a library of reusable components that render your data model. The components will only have `render()` methods since this is a static version of your app. The component at the top of the hierarchy (`FilterableProductTable`) will take your data model as a prop. If you make a change to your underlying data model and call `root.render()` again, the UI will be updated. You can see how your UI is updated and where to make changes. React's **one-way data flow** (also called *one-way binding*) keeps everything modular and fast.

Refer to the [React docs](#) if you need help executing this step.

A Brief Interlude: Props vs State

There are two types of "model" data in React: props and state. It's important to understand the distinction between the two; skim [the official React docs](#) if you aren't sure what the difference is. See also [FAQ: What is the difference between state and props?](#)

Step 3: Identify The Minimal (but complete) Representation Of UI State

To make your UI interactive, you need to be able to trigger changes to your underlying data model. React achieves this with **state**.

To build your app correctly, you first need to think of the minimal set of mutable state that your app needs. The key here is **DRY: Don't Repeat Yourself**. Figure out the absolute minimal representation of the state your application needs and compute everything else you need on-demand. For example, if you're building a TODO list, keep an array of the TODO items around; don't keep a separate state variable for the count. Instead, when you want to render the TODO count, take the length of the TODO items array.

Think of all the pieces of data in our example application. We have:

- The original list of products
- The search text the user has entered
- The value of the checkbox
- The filtered list of products

Let's go through each one and figure out which one is state. Ask three questions about each piece of data:

1. Is it passed in from a parent via props? If so, it probably isn't state.
2. Does it remain unchanged over time? If so, it probably isn't state.
3. Can you compute it based on any other state or props in your component? If so, it isn't state.

The original list of products is passed in as props, so that's not state. The search text and the checkbox seem to be state since they change over time and can't be computed from anything. And finally, the filtered list of products isn't state because it can be computed by combining the original list of products with the search text and value of the checkbox.

So finally, our state is:

- The search text the user has entered
- The value of the checkbox

Step 4: Identify Where Your State Should Live

See the Pen [Thinking In React: Step 4](#) on [CodePen](#).

OK, so we've identified what the minimal set of app state is. Next, we need to identify which component mutates, or *owns*, this state.

Remember: React is all about one-way data flow down the component hierarchy. It may not be immediately clear which component should own what state. **This is often the most challenging part for newcomers to understand**, so follow these steps to figure it out:

For each piece of state in your application:

- Identify every component that renders something based on that state.
- Find a common owner component (a single component above all the components that need the state in the hierarchy).
- Either the common owner or another component higher up in the hierarchy should own the state.
- If you can't find a component where it makes sense to own the state, create a new component solely for holding the state and add it somewhere in the hierarchy above the common owner component.

Let's run through this strategy for our application:

- `ProductTable` needs to filter the product list based on state and `SearchBar` needs to display the search text and checked state.
- The common owner component is `FilterableProductTable`.
- It conceptually makes sense for the filter text and checked value to live in `FilterableProductTable`.

Cool, so we've decided that our state lives in `FilterableProductTable`. First, add an instance property `this.state = {filterText: '', inStockOnly: false}` to `FilterableProductTable`'s constructor to reflect the initial state of your application. Then, pass `filterText` and `inStockOnly` to `ProductTable` and `SearchBar` as a prop. Finally, use these props to filter the rows in `ProductTable` and set the values of the form fields in `SearchBar`.

You can start seeing how your application will behave: set `filterText` to "ball" and refresh your app. You'll see that the data table is updated correctly.

Step 5: Add Inverse Data Flow

See the Pen [Thinking In React: Step 5](#) on [CodePen](#).

So far, we've built an app that renders correctly as a function of props and state flowing down the hierarchy. Now it's time to support data flowing the other way: the form components deep in the hierarchy need to update the state in `FilterableProductTable`.

React makes this data flow explicit to help you understand how your program works, but it does require a little more typing than traditional two-way data binding.

If you try to type or check the box in the previous version of the example (step 4), you'll see that React ignores your input. This is intentional, as we've set the `value` prop of the `input` to always be equal to the `state` passed in from `FilterableProductTable`.

Let's think about what we want to happen. We want to make sure that whenever the user changes the form, we update the state to reflect the user input. Since components should only update their own state, `FilterableProductTable` will pass callbacks to `SearchBar` that will fire whenever the state should be updated. We can use the `onChange` event on the inputs to be notified of it. The callbacks passed by `FilterableProductTable` will call `setState()`, and the app will be updated.

And That's It

Hopefully, this gives you an idea of how to think about building components and applications with React. While it may be a little more typing than you're used to, remember that code is read far more often than it's written, and it's less difficult to read this

modular, explicit code. As you start to build large libraries of components, you'll appreciate this explicitness and modularity, and with code reuse, your lines of code will start to shrink. :)

II. ADVANCED GUIDES

1. Accessibility

Why Accessibility?

Web accessibility (also referred to as **a11y**) is the design and creation of websites that can be used by everyone. Accessibility support is necessary to allow assistive technology to interpret web pages.

React fully supports building accessible websites, often by using standard HTML techniques.

Standards and Guidelines

WCAG

The [Web Content Accessibility Guidelines](#) provides guidelines for creating accessible web sites.

The following WCAG checklists provide an overview:

- [WCAG checklist from Wuhcag](#)
- [WCAG checklist from WebAIM](#)
- [Checklist from The A11Y Project](#)

WAI-ARIA

The [Web Accessibility Initiative - Accessible Rich Internet Applications](#) document contains techniques for building fully accessible JavaScript widgets.

Note that all `aria-*` HTML attributes are fully supported in JSX. Whereas most DOM properties and attributes in React are camelCased, these attributes should be hyphen-cased (also known as kebab-case, lisp-case, etc) as they are in plain HTML:

```
<input
  type="text"
  aria-label={labelText}
  aria-required="true"
  onChange={onChangeHandler}
  value={inputValue}
  name="name"
/>;
```

Semantic HTML

Semantic HTML is the foundation of accessibility in a web application. Using the various HTML elements to reinforce the meaning of information in our websites will often give us accessibility for free.

- [MDN HTML elements reference](#)

Sometimes we break HTML semantics when we add `<div>` elements to our JSX to make our React code work, especially when working with lists (``, `` and `<dl>`) and the HTML `<table>`. In these cases we should rather use [React Fragments](#) to group together multiple elements.

For example,

```
import React, { Fragment } from 'react';
function ListItem({ item }) {
  return (
    <Fragment>
      <dt>{item.term}</dt> <dd>{item.description}</dd>
    </Fragment>
  );
}
```

```

    });
  }

  function Glossary(props) {
    return (
      <dl>
        {props.items.map((item) => (
          <ListItem item={item} key={item.id} />
        ))}
      </dl>
    );
  }

```

You can map a collection of items to an array of fragments as you would any other type of element as well:

```

function Glossary(props) {
  return (
    <dl>
      {props.items.map((item) => (
        // Fragments should also have a `key` prop when mapping collections
        <Fragment key={item.id}>
          <dt>{item.term}</dt> <dd>{item.description}</dd>
        </Fragment>
      ))}
    </dl>
  );
}

```

When you don't need any props on the Fragment tag you can use the [short syntax](#), if your tooling supports it:

```

function ListItem({ item }) {
  return (
    <>
      <dt>{item.term}</dt>
      <dd>{item.description}</dd>
    </>
  );
}

```

For more info, see [the Fragments documentation](#).

Accessible Forms

Labeling

Every HTML form control, such as `<input>` and `<textarea>`, needs to be labeled accessibly. We need to provide descriptive labels that are also exposed to screen readers.

The following resources show us how to do this:

- [The W3C shows us how to label elements](#)
- [WebAIM shows us how to label elements](#)
- [The Paciello Group explains accessible names](#)

Although these standard HTML practices can be directly used in React, note that the `for` attribute is written as `htmlFor` in JSX:

```

<label htmlFor="namedInput">Name:</label>
<input id="namedInput" type="text" name="name"/>

```

Notifying the user of errors

Error situations need to be understood by all users. The following link shows us how to expose error texts to screen readers as well:

- [The W3C demonstrates user notifications](#)
- [WebAIM looks at form validation](#)

Focus Control

Ensure that your web application can be fully operated with the keyboard only:

- [WebAIM talks about keyboard accessibility](#).

Keyboard focus and focus outline

Keyboard focus refers to the current element in the DOM that is selected to accept input from the keyboard. We see it everywhere as a focus outline similar to that shown in the following image:



Only ever use CSS that removes this outline, for example by setting `outline: 0`, if you are replacing it with another focus outline implementation.

Mechanisms to skip to desired content

Provide a mechanism to allow users to skip past navigation sections in your application as this assists and speeds up keyboard navigation.

Skiplinks or Skip Navigation Links are hidden navigation links that only become visible when keyboard users interact with the page. They are very easy to implement with internal page anchors and some styling:

- [WebAIM - Skip Navigation Links](#)

Also use landmark elements and roles, such as `<main>` and `<aside>`, to demarcate page regions as assistive technology allow the user to quickly navigate to these sections.

Read more about the use of these elements to enhance accessibility here:

- [Accessible Landmarks](#)

Programmatically managing focus

Our React applications continuously modify the HTML DOM during runtime, sometimes leading to keyboard focus being lost or set to an unexpected element. In order to repair this, we need to programmatically nudge the keyboard focus in the right direction. For example, by resetting keyboard focus to a button that opened a modal window after that modal window is closed.

MDN Web Docs takes a look at this and describes how we can build [keyboard-navigable JavaScript widgets](#).

To set focus in React, we can use [Refs to DOM elements](#).

Using this, we first create a ref to an element in the JSX of a component class:

```
class CustomTextInput extends React.Component {
  constructor(props) {
    super(props);
    // Create a ref to store the textInput DOM element
    this.textInput = React.createRef();
  }
  render() {
    // Use the `ref` callback to store a reference to the text input DOM
    // element in an instance field (for example, this.textInput).
    return <input type="text" ref={this.textInput} />;
  }
}
```

Then we can focus it elsewhere in our component when needed:

```
focus() {
  // Explicitly focus the text input using the raw DOM API
  // Note: we're accessing "current" to get the DOM node
  this.textInput.current.focus();
}
```

Sometimes a parent component needs to set focus to an element in a child component. We can do this by [exposing DOM refs to parent components](#) through a special prop on the child component that forwards the parent's ref to the child's DOM node.

When using a [HOC](#) to extend components, it is recommended to [forward the ref](#) to the wrapped component using the `forwardRef` function of React. If a third party HOC does not implement ref forwarding, the above pattern can still be used as a fallback.

A great focus management example is the [react-aria-modal](#). This is a relatively rare example of a fully accessible modal window. Not only does it set initial focus on the cancel button (preventing the keyboard user from accidentally activating the success action) and trap keyboard focus inside the modal, it also resets focus back to the element that initially triggered the modal.

Note:

While this is a very important accessibility feature, it is also a technique that should be used judiciously. Use it to repair the keyboard focus flow when it is disturbed, not to try and anticipate how users want to use applications.

Mouse and pointer events

Ensure that all functionality exposed through a mouse or pointer event can also be accessed using the keyboard alone. Depending only on the pointer device will lead to many cases where keyboard users cannot use your application.

To illustrate this, let's look at a prolific example of broken accessibility caused by click events. This is the outside click pattern, where a user can disable an opened popover by clicking outside the element.

Select an option

Load the option

Remove the option

This is typically implemented by attaching a `click` event to the `window` object that closes the popover:

```
class OuterClickExample extends React.Component {
  constructor(props) {
    super(props);

    this.state = { isOpen: false };
    this.toggleContainer = React.createRef();

    this.onClickHandler = this.onClickHandler.bind(this);
    this.onClickOutsideHandler = this.onClickOutsideHandler.bind(this);
  }

  componentDidMount() {
    window.addEventListener('click', this.onClickOutsideHandler);
  }
  componentWillUnmount() {
    window.removeEventListener('click', this.onClickOutsideHandler);
  }

  onClickHandler() {
    this.setState((currentState) => ({
      isOpen: !currentState.isOpen
    }));
  }

  onClickOutsideHandler(event) {
    if (this.state.isOpen && !this.toggleContainer.current.contains(event.target)) {
      this.setState({ isOpen: false });
    }
  }

  render() {
    return (
```



```

    <div ref={this.toggleContainer}>
      <button onClick={this.onClickHandler}>Select an option</button>
      {this.state.isOpen && (
        <ul>
          <li>Option 1</li> <li>Option 2</li> <li>Option 3</li>
        </ul>
      )}
    </div>
  );
}
}

```

This may work fine for users with pointer devices, such as a mouse, but operating this with the keyboard alone leads to broken functionality when tabbing to the next element as the `window` object never receives a `click` event. This can lead to obscured functionality which blocks users from using your application.

Select an option

Load the option

Remove the option

The same functionality can be achieved by using appropriate event handlers instead, such as `onBlur` and `onFocus`:

```

class BlurExample extends React.Component {
  constructor(props) {
    super(props);

    this.state = { isOpen: false };
    this.timeOutId = null;

    this.onClickHandler = this.onClickHandler.bind(this);
    this.onBlurHandler = this.onBlurHandler.bind(this);
    this.onFocusHandler = this.onFocusHandler.bind(this);
  }

  onClickHandler() {
    this.setState((currentState) => ({
      isOpen: !currentState.isOpen
    }));
  }

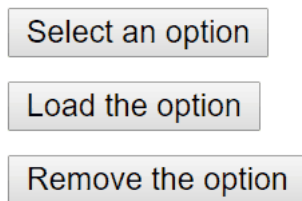
  // We close the popover on the next tick by using setTimeout.
  // This is necessary because we need to first check if
  // another child of the element has received focus as
  // the blur event fires prior to the new focus event.
  onBlurHandler() {
    this.timeOutId = setTimeout(() => {
      this.setState({ isOpen: false });
    });
  }

  // If a child receives focus, do not close the popover.
  onFocusHandler() {
    clearTimeout(this.timeOutId);
  }

  render() {
    // React assists us by bubbling the blur and
    // focus events to the parent.
    return (
      <div onBlur={this.onBlurHandler} onFocus={this.onFocusHandler}>
        <button onClick={this.onClickHandler} aria-haspopup="true" aria-expanded={this.state.isOpen}>
          Select an option
        </button>
        {this.state.isOpen && (
          <ul>
            <li>Option 1</li> <li>Option 2</li> <li>Option 3</li>
          </ul>
        )}
      </div>
    );
  }
}

```

This code exposes the functionality to both pointer device and keyboard users. Also note the added `aria-*` props to support screen-reader users. For simplicity's sake the keyboard events to enable `arrow key` interaction of the popover options have not been implemented.



This is one example of many cases where depending on only pointer and mouse events will break functionality for keyboard users. Always testing with the keyboard will immediately highlight the problem areas which can then be fixed by using keyboard aware event handlers.

More Complex Widgets

A more complex user experience should not mean a less accessible one. Whereas accessibility is most easily achieved by coding as close to HTML as possible, even the most complex widget can be coded accessibly.

Here we require knowledge of [ARIA Roles](#) as well as [ARIA States and Properties](#). These are toolboxes filled with HTML attributes that are fully supported in JSX and enable us to construct fully accessible, highly functional React components.

Each type of widget has a specific design pattern and is expected to function in a certain way by users and user agents alike:

- [ARIA Authoring Practices Guide \(APG\) - Design Patterns and Examples](#)
- [Heydon Pickering - ARIA Examples](#)
- [Inclusive Components](#)

Other Points for Consideration

Setting the language

Indicate the human language of page texts as screen reader software uses this to select the correct voice settings:

- [WebAIM - Document Language](#)

Setting the document title

Set the document `<title>` to correctly describe the current page content as this ensures that the user remains aware of the current page context:

- [WCAG - Understanding the Document Title Requirement](#)

We can set this in React using the [React Document Title Component](#).

Color contrast

Ensure that all readable text on your website has sufficient color contrast to remain maximally readable by users with low vision:

- [WCAG - Understanding the Color Contrast Requirement](#)
- [Everything About Color Contrast And Why You Should Rethink It](#)
- [A11yProject - What is Color Contrast](#)

It can be tedious to manually calculate the proper color combinations for all cases in your website so instead, you can [calculate an entire accessible color palette with Colorable](#).

Both the aXe and WAVE tools mentioned below also include color contrast tests and will report on contrast errors.

If you want to extend your contrast testing abilities you can use these tools:

- [WebAIM - Color Contrast Checker](#)
- [The Paciello Group - Color Contrast Analyzer](#)

Development and Testing Tools

There are a number of tools we can use to assist in the creation of accessible web applications.

The keyboard

By far the easiest and also one of the most important checks is to test if your entire website can be reached and used with the keyboard alone. Do this by:

1. Disconnecting your mouse.
2. Using `Tab` and `Shift+Tab` to browse.
3. Using `Enter` to activate elements.
4. Where required, using your keyboard arrow keys to interact with some elements, such as menus and dropdowns.

Development assistance

We can check some accessibility features directly in our JSX code. Often intellisense checks are already provided in JSX aware IDE's for the ARIA roles, states and properties. We also have access to the following tool:

eslint-plugin-jsx-a11y

The [eslint-plugin-jsx-a11y](#) plugin for ESLint provides AST linting feedback regarding accessibility issues in your JSX. Many IDE's allow you to integrate these findings directly into code analysis and source code windows.

[Create React App](#) has this plugin with a subset of rules activated. If you want to enable even more accessibility rules, you can create an `.eslintrc` file in the root of your project with this content:

```
{
  "extends": ["react-app", "plugin:jsx-a11y/recommended"],
  "plugins": ["jsx-a11y"]
}
```

Testing accessibility in the browser

A number of tools exist that can run accessibility audits on web pages in your browser. Please use them in combination with other accessibility checks mentioned here as they can only test the technical accessibility of your HTML.

aXe, aXe-core and react-axe

Deque Systems offers [aXe-core](#) for automated and end-to-end accessibility tests of your applications. This module includes integrations for Selenium.

[The Accessibility Engine](#) or aXe, is an accessibility inspector browser extension built on [aXe-core](#).

You can also use the [@axe-core/react](#) module to report these accessibility findings directly to the console while developing and debugging.

WebAIM WAVE

The [Web Accessibility Evaluation Tool](#) is another accessibility browser extension.

Accessibility inspectors and the Accessibility Tree

[The Accessibility Tree](#) is a subset of the DOM tree that contains accessible objects for every DOM element that should be exposed to assistive technology, such as screen readers.

In some browsers we can easily view the accessibility information for each element in the accessibility tree:

- [Using the Accessibility Inspector in Firefox](#)

- [Using the Accessibility Inspector in Chrome](#)
- [Using the Accessibility Inspector in OS X Safari](#)

Screen readers

Testing with a screen reader should form part of your accessibility tests.

Please note that browser / screen reader combinations matter. It is recommended that you test your application in the browser best suited to your screen reader of choice.

Commonly Used Screen Readers

NVDA in Firefox

[NonVisual Desktop Access](#) or NVDA is an open source Windows screen reader that is widely used.

Refer to the following guides on how to best use NVDA:

- [WebAIM - Using NVDA to Evaluate Web Accessibility](#)
- [Deque - NVDA Keyboard Shortcuts](#)

VoiceOver in Safari

VoiceOver is an integrated screen reader on Apple devices.

Refer to the following guides on how to activate and use VoiceOver:

- [WebAIM - Using VoiceOver to Evaluate Web Accessibility](#)
- [Deque - VoiceOver for OS X Keyboard Shortcuts](#)
- [Deque - VoiceOver for iOS Shortcuts](#)

JAWS in Internet Explorer

[Job Access With Speech](#) or JAWS, is a prolifically used screen reader on Windows.

Refer to the following guides on how to best use JAWS:

- [WebAIM - Using JAWS to Evaluate Web Accessibility](#)
- [Deque - JAWS Keyboard Shortcuts](#)

Other Screen Readers

ChromeVox in Google Chrome

[ChromeVox](#) is an integrated screen reader on Chromebooks and is available [as an extension](#) for Google Chrome.

Refer to the following guides on how best to use ChromeVox:

- [Google Chromebook Help - Use the Built-in Screen Reader](#)
- [ChromeVox Classic Keyboard Shortcuts Reference](#)

2. Code-Splitting

Bundling

Most React apps will have their files “bundled” using tools like [Webpack](#), [Rollup](#) or [Browserify](#). Bundling is the process of following imported files and merging them into a single file: a “bundle”. This bundle can then be included on a webpage to load an entire app at once.

Example

App:

```
// app.js
import { add } from './math.js';

console.log(add(16, 26)); // 42

// math.js
export function add(a, b) {
  return a + b;
}
```

Bundle:

```
function add(a, b) {
  return a + b;
}

console.log(add(16, 26)); // 42
```

Note:

Your bundles will end up looking a lot different than this.

If you're using [Create React App](#), [Next.js](#), [Gatsby](#), or a similar tool, you will have a Webpack setup out of the box to bundle your app.

If you aren't, you'll need to set up bundling yourself. For example, see the [Installation](#) and [Getting Started](#) guides on the Webpack docs.

Code Splitting

Bundling is great, but as your app grows, your bundle will grow too. Especially if you are including large third-party libraries. You need to keep an eye on the code you are including in your bundle so that you don't accidentally make it so large that your app takes a long time to load.

To avoid winding up with a large bundle, it's good to get ahead of the problem and start "splitting" your bundle. Code-Splitting is a feature supported by bundlers like [Webpack](#), [Rollup](#) and Browserify (via [factor-bundle](#)) which can create multiple bundles that can be dynamically loaded at runtime.

Code-splitting your app can help you "lazy-load" just the things that are currently needed by the user, which can dramatically improve the performance of your app. While you haven't reduced the overall amount of code in your app, you've avoided loading code that the user may never need, and reduced the amount of code needed during the initial load.

import()

The best way to introduce code-splitting into your app is through the dynamic `import()` syntax.

Before:

```
import { add } from './math';

console.log(add(16, 26));
```

After:

```
import("./math").then(math => {
  console.log(math.add(16, 26));
});
```

When Webpack comes across this syntax, it automatically starts code-splitting your app. If you're using Create React App, this is already configured for you and you can [start using it](#) immediately. It's also supported out of the box in [Next.js](#).

If you're setting up Webpack yourself, you'll probably want to read Webpack's [guide on code splitting](#). Your Webpack config should look vaguely [like this](#).

When using [Babel](#), you'll need to make sure that Babel can parse the dynamic import syntax but is not transforming it. For that you will need [@babel/plugin-syntax-dynamic-import](#).

React.lazy

The `React.lazy` function lets you render a dynamic import as a regular component.

Before:

```
import OtherComponent from './OtherComponent';
```

After:

```
const OtherComponent = React.lazy(() => import('./OtherComponent'));
```

This will automatically load the bundle containing the `OtherComponent` when this component is first rendered.

`React.lazy` takes a function that must call a dynamic `import()`. This must return a `Promise` which resolves to a module with a `default` export containing a React component.

The lazy component should then be rendered inside a `Suspense` component, which allows us to show some fallback content (such as a loading indicator) while we're waiting for the lazy component to load.

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

const OtherComponent = React.lazy(() => import('./OtherComponent'));

function MyComponent() {
  return (
    <div>
      <Suspense fallback={<div>Loading...</div>}>
        <OtherComponent />
      </Suspense>
    </div>;
  )
}
```

The `fallback` prop accepts any React elements that you want to render while waiting for the component to load. You can place the `Suspense` component anywhere above the lazy component. You can even wrap multiple lazy components with a single `Suspense` component.

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

const OtherComponent = React.lazy(() => import('./OtherComponent'));
const AnotherComponent = React.lazy(() => import('./AnotherComponent'));

function MyComponent() {
  return (
    <div>
      <Suspense fallback={<div>Loading...</div>}>
        <section>
          <OtherComponent />
          <AnotherComponent />
        </section>
      </Suspense>
    </div>;
  )
}
```

Avoiding fallbacks

Any component may suspend as a result of rendering, even components that were already shown to the user. In order for screen content to always be consistent, if an already shown component suspends, React has to hide its tree up to the closest `<Suspense>` boundary. However, from the user's perspective, this can be disorienting.

Consider this tab switcher:

```
import React, { Suspense } from 'react';
import Tabs from './Tabs';
import Glimmer from './Glimmer';

const Comments = React.lazy(() => import('./Comments'));
const Photos = React.lazy(() => import('./Photos'));

function MyComponent() {
  const [tab, setTab] = React.useState('photos');

  function handleTabSelect(tab) {
    setTab(tab);
  }

  return (
    <div>
      <Tabs onTabSelect={handleTabSelect} />
      <Suspense fallback={<Glimmer />}>{tab === 'photos' ? <Photos /> : <Comments />}</Suspense>
    </div>
  );
}
```

In this example, if `tab` gets changed from `'photos'` to `'comments'`, but `Comments` suspends, the user will see a glimmer. This makes sense because the user no longer wants to see `Photos`, the `Comments` component is not ready to render anything, and React needs to keep the user experience consistent, so it has no choice but to show the `Glimmer` above.

However, sometimes this user experience is not desirable. In particular, it is sometimes better to show the “old” UI while the new UI is being prepared. You can use the new `startTransition` API to make React do this:

```
function handleTabSelect(tab) {
  startTransition(() => {
    setTab(tab);
  });
}
```

Here, you tell React that setting `tab` to `'comments'` is not an urgent update, but is a transition that may take some time. React will then keep the old UI in place and interactive, and will switch to showing `<Comments />` when it is ready. See Transitions for more info.

Error boundaries

If the other module fails to load (for example, due to network failure), it will trigger an error. You can handle these errors to show a nice user experience and manage recovery with Error Boundaries. Once you’ve created your Error Boundary, you can use it anywhere above your lazy components to display an error state when there’s a network error.

```
import React, { Suspense } from 'react';
import MyErrorBoundary from './MyErrorBoundary';

const OtherComponent = React.lazy(() => import('./OtherComponent'));
const AnotherComponent = React.lazy(() => import('./AnotherComponent'));

const MyComponent = () => (
  <div>
    <MyErrorBoundary>
      <Suspense fallback={<div>Loading...</div>}>
        <section>
          <OtherComponent />
          <AnotherComponent />
        </section>
      </Suspense>
    </MyErrorBoundary>
  </div>
);
```

Route-based code splitting

Deciding where in your app to introduce code splitting can be a bit tricky. You want to make sure you choose places that will split bundles evenly, but won’t disrupt the user experience.

A good place to start is with routes. Most people on the web are used to page transitions taking some amount of time to load. You also tend to be re-rendering the entire page at once so your users are unlikely to be interacting with other elements on the page at the same time.

Here's an example of how to setup route-based code splitting into your app using libraries like [React Router](#) with `React.lazy`.

```
import React, { Suspense, lazy } from 'react';
import { BrowserRouter as Router, Routes, Route } from 'react-router-dom';

const Home = lazy(() => import('./routes/Home'));
const About = lazy(() => import('./routes/About'));

const App = () => (
  <Router>
    <Suspense fallback={<div>Loading...</div>}>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
      </Routes>
    </Suspense>
  </Router>
);
```

Named Exports

`React.lazy` currently only supports default exports. If the module you want to import uses named exports, you can create an intermediate module that reexports it as the default. This ensures that tree shaking keeps working and that you don't pull in unused components.

```
// ManyComponents.js
export const MyComponent = /* ... */;
export const MyUnusedComponent = /* ... */;

// MyComponent.js
export { MyComponent as default } from "../ManyComponents.js";

// MyApp.js
import React, { lazy } from 'react';
const MyComponent = lazy(() => import("./MyComponent.js"));
```

3. Context

Context provides a way to pass data through the component tree without having to pass props down manually at every level.

In a typical React application, data is passed top-down (parent to child) via props, but such usage can be cumbersome for certain types of props (e.g. locale preference, UI theme) that are required by many components within an application. Context provides a way to share values like these between components without having to explicitly pass a prop through every level of the tree.

- [When to Use Context](#)
- [Before You Use Context](#)
- [API](#)
 - [React.createContext](#)
 - [Context.Provider](#)
 - [Class.contextType](#)
 - [Context.Consumer](#)
 - [Context.displayName](#)
- [Examples](#)
 - [Dynamic Context](#)
 - [Updating Context from a Nested Component](#)
 - [Consuming Multiple Contexts](#)

- [Caveats](#)
- [Legacy API](#)

When to Use Context

Context is designed to share data that can be considered “global” for a tree of React components, such as the current authenticated user, theme, or preferred language. For example, in the code below we manually thread through a “theme” prop in order to style the Button component:

```
class App extends React.Component {
  render() {
    return <Toolbar theme="dark" />;
  }
}

function Toolbar(props) {
  // The Toolbar component must take an extra "theme" prop
  // and pass it to the ThemedButton. This can become painful
  // if every single button in the app needs to know the theme
  // because it would have to be passed through all components.
  return (
    <div>
      <ThemedButton theme={props.theme} />
    </div>
  );
}

class ThemedButton extends React.Component {
  render() {
    return <Button theme={this.props.theme} />;
  }
}
```

Using context, we can avoid passing props through intermediate elements:

```
// Context lets us pass a value deep into the component tree
// without explicitly threading it through every component.
// Create a context for the current theme (with "light" as the default).
const ThemeContext = React.createContext('light');
class App extends React.Component {
  render() {
    // Use a Provider to pass the current theme to the tree below.
    // Any component can read it, no matter how deep it is.
    // In this example, we're passing "dark" as the current value.
    return (
      <ThemeContext.Provider value="dark">
        <Toolbar />
      </ThemeContext.Provider>
    );
  }
}

// A component in the middle doesn't have to
// pass the theme down explicitly anymore.
function Toolbar() {
  return (
    <div>
      <ThemedButton />
    </div>
  );
}

class ThemedButton extends React.Component {
  // Assign a contextType to read the current theme context.
  // React will find the closest theme Provider above and use its value.
  // In this example, the current theme is "dark".
  static contextType = ThemeContext;
  render() {
    return <Button theme={this.context} />;
  }
}
```

Before You Use Context

Context is primarily used when some data needs to be accessible by *many* components at different nesting levels. Apply it sparingly because it makes component reuse more difficult.

If you only want to avoid passing some props through many levels, **component composition** is often a simpler solution than context.

For example, consider a `Page` component that passes a `user` and `avatarSize` prop several levels down so that deeply nested `Link` and `Avatar` components can read it:

```
<Page user={user} avatarSize={avatarSize} /> // ... which renders ...
<PageLayout user={user} avatarSize={avatarSize} /> // ... which renders ...
<NavigationBar user={user} avatarSize={avatarSize} /> // ... which renders ...
<Link href={user.permalink}>
  <Avatar user={user} size={avatarSize} />
</Link>
```

It might feel redundant to pass down the `user` and `avatarSize` props through many levels if in the end only the `Avatar` component really needs it. It's also annoying that whenever the `Avatar` component needs more props from the top, you have to add them at all the intermediate levels too.

One way to solve this issue **without context** is to pass down the `Avatar` component itself so that the intermediate components don't need to know about the `user` or `avatarSize` props:

```
function Page(props) {
  const user = props.user;
  const userLink = (
    <Link href={user.permalink}>
      <Avatar user={user} size={props.avatarSize} />
    </Link>
  );
  return <PageLayout userLink={userLink} />;
}

// Now, we have:
<Page user={user} avatarSize={avatarSize} /> // ... which renders ...
<PageLayout userLink={...} /> // ... which renders ...
<NavigationBar userLink={...} /> // ... which renders ...
{props.userLink}
```

With this change, only the top-most `Page` component needs to know about the `Link` and `Avatar` components' use of `user` and `avatarSize`.

This *inversion of control* can make your code cleaner in many cases by reducing the amount of props you need to pass through your application and giving more control to the root components. Such inversion, however, isn't the right choice in every case; moving more complexity higher in the tree makes those higher-level components more complicated and forces the lower-level components to be more flexible than you may want.

You're not limited to a single child for a component. You may pass multiple children, or even have multiple separate "slots" for children, [as documented here](#):

```
function Page(props) {
  const user = props.user;
  const content = <Feed user={user} />;
  const topBar = (
    <NavigationBar>
      <Link href={user.permalink}>
        <Avatar user={user} size={props.avatarSize} />
      </Link>
    </NavigationBar>
  );
  return <PageLayout topBar={topBar} content={content} />;
}
```

This pattern is sufficient for many cases when you need to decouple a child from its immediate parents. You can take it even further with [render props](#) if the child needs to communicate with the parent before rendering.

However, sometimes the same data needs to be accessible by many components in the tree, and at different nesting levels. Context lets you "broadcast" such data, and changes to it, to all components below. Common examples where using context might be simpler than the alternatives include managing the current locale, theme, or a data cache.

API

React.createContext

```
const MyContext = React.createContext(defaultValue);
```

Creates a Context object. When React renders a component that subscribes to this Context object it will read the current context value from the closest matching `Provider` above it in the tree.

The `defaultValue` argument is **only** used when a component does not have a matching `Provider` above it in the tree. This default value can be helpful for testing components in isolation without wrapping them. Note: passing `undefined` as a `Provider` value does not cause consuming components to use `defaultValue`.

Context.Provider

```
<MyContext.Provider value={/* some value */}>
```

Every Context object comes with a `Provider` React component that allows consuming components to subscribe to context changes.

The `Provider` component accepts a `value` prop to be passed to consuming components that are descendants of this `Provider`. One `Provider` can be connected to many consumers. `Providers` can be nested to override values deeper within the tree.

All consumers that are descendants of a `Provider` will re-render whenever the `Provider`'s `value` prop changes. The propagation from `Provider` to its descendant consumers (including `contextType` and `useContext`) is not subject to the `shouldComponentUpdate` method, so the consumer is updated even when an ancestor component skips an update.

Changes are determined by comparing the new and old values using the same algorithm as `Object.is`.

Note

The way changes are determined can cause some issues when passing objects as `value`: see [Caveats](#).

Class.contextType

```
class MyClass extends React.Component {
  componentDidMount() {
    let value = this.context;
    /* perform a side-effect at mount using the value of MyContext */
  }
  componentDidUpdate() {
    let value = this.context;
    /* ... */
  }
  componentWillUnmount() {
    let value = this.context;
    /* ... */
  }
  render() {
    let value = this.context;
    /* render something based on the value of MyContext */
  }
}
MyClass.contextType = MyContext;
```

The `contextType` property on a class can be assigned a Context object created by `React.createContext()`. Using this property lets you consume the nearest current value of that Context type using `this.context`. You can reference this in any of the lifecycle methods including the render function.

Note:

You can only subscribe to a single context using this API. If you need to read more than one see [Consuming Multiple Contexts](#).

If you are using the experimental [public class fields syntax](#), you can use a **static** class field to initialize your `contextType`.

```
class MyClass extends React.Component {
  static contextType = MyContext;
  render() {
    let value = this.context;
    /* render something based on the value */
  }
}
```

Context.Consumer

```
<MyContext.Consumer>
  {value => /* render something based on the context value */}
</MyContext.Consumer>
```

A React component that subscribes to context changes. Using this component lets you subscribe to a context within a [function component](#).

Requires a [function as a child](#). The function receives the current context value and returns a React node. The `value` argument passed to the function will be equal to the `value` prop of the closest Provider for this context above in the tree. If there is no Provider for this context above, the `value` argument will be equal to the `defaultValue` that was passed to `createContext()`.

Note

For more information about the ‘function as a child’ pattern, see [render props](#).

Context.displayName

Context object accepts a `displayName` string property. React DevTools uses this string to determine what to display for the context.

For example, the following component will appear as MyDisplayName in the DevTools:

```
const MyContext = React.createContext(/* some value */);
MyContext.displayName = 'MyDisplayName';
<MyContext.Provider> // "MyDisplayName.Provider" in DevTools
<MyContext.Consumer> // "MyDisplayName.Consumer" in DevTools
```

Examples

Dynamic Context

A more complex example with dynamic values for the theme:

theme-context.js

```
export const themes = {
  light: {
    foreground: '#000000',
    background: '#ffffff'
  },
  dark: {
    foreground: '#ffffff',
    background: '#222222'
  }
};

export const ThemeContext = React.createContext(themes.dark); // default value
```

themed-button.js

```
import { ThemeContext } from './theme-context';

class ThemedButton extends React.Component {
  render() {
    let props = this.props;
    let theme = this.context;
    return <button {...props} style={{ backgroundColor: theme.background }} />;
  }
}
ThemedButton.contextType = ThemeContext;
export default ThemedButton;
```

app.js

```
import { ThemeContext, themes } from './theme-context';
import ThemedButton from './themed-button';

// An intermediate component that uses the ThemedButton
function Toolbar(props) {
  return <ThemedButton onClick={props.changeTheme}> Change Theme</ThemedButton>;
}

class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      theme: themes.light
    };

    this.toggleTheme = () => {
      this.setState((state) => ({
        theme: state.theme === themes.dark ? themes.light : themes.dark
      }));
    };
  }

  render() {
    // The ThemedButton button inside the ThemeProvider
    // uses the theme from state while the one outside uses
    // the default dark theme
    return (
      <Page>
        <ThemeContext.Provider value={this.state.theme}>
          <Toolbar changeTheme={this.toggleTheme} />
        </ThemeContext.Provider>
        <Section>
          <ThemedButton />
        </Section>
      </Page>
    );
  }
}

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

Updating Context from a Nested Component

It is often necessary to update the context from a component that is nested somewhere deeply in the component tree. In this case you can pass a function down through the context to allow consumers to update the context:

theme-context.js

```
// Make sure the shape of the default value passed to
// createContext matches the shape that the consumers expect!
export const ThemeContext = React.createContext({
  theme: themes.dark,
  toggleTheme: () => {}
});
```

theme-toggler-button.js

```
import { ThemeContext } from './theme-context';

function ThemeTogglerButton() {
```

```
// The Theme Toggler Button receives not only the theme
// but also a toggleTheme function from the context
return (
  <ThemeContext.Consumer>
    {{ theme, toggleTheme }} => (
      <button onClick={toggleTheme} style={{ backgroundColor: theme.background }}>
        Toggle Theme
      </button>
    )
  </ThemeContext.Consumer>
);
}

export default ThemeTogglerButton;
```

app.js

```
import { ThemeContext, themes } from './theme-context';
import ThemeTogglerButton from './theme-toggler-button';

class App extends React.Component {
  constructor(props) {
    super(props);

    this.toggleTheme = () => {
      this.setState((state) => ({
        theme: state.theme === themes.dark ? themes.light : themes.dark
      }));
    };

    // State also contains the updater function so it will
    // be passed down into the context provider
    this.state = {
      theme: themes.light,
      toggleTheme: this.toggleTheme
    };
  }

  render() {
    // The entire state is passed to the provider
    return (
      <ThemeContext.Provider value={this.state}>
        <Content />
      </ThemeContext.Provider>
    );
  }
}

function Content() {
  return (
    <div>
      <ThemeTogglerButton />
    </div>
  );
}

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

Consuming Multiple Contexts

To keep context re-rendering fast, React needs to make each context consumer a separate node in the tree.

```
// Theme context, default to light theme
const ThemeContext = React.createContext('light');

// Signed-in user context
const UserContext = React.createContext({
  name: 'Guest'
});

class App extends React.Component {
  render() {
    const { signedInUser, theme } = this.props;

    // App component that provides initial context values
    return (
      <ThemeContext.Provider value={theme}>
        <UserContext.Provider value={signedInUser}>
```

```

    <Layout />
  </UserContext.Provider>
</ThemeContext.Provider>
);
}
}

function Layout() {
  return (
    <div>
      <Sidebar /> <Content />
    </div>
  );
}

// A component may consume multiple contexts
function Content() {
  return (
    <ThemeContext.Consumer>
      {(theme) => <UserContext.Consumer> {(user) => <ProfilePage user={user} theme={theme} />} </UserContext.Consumer>}
    </ThemeContext.Consumer>
  );
}

```

If two or more context values are often used together, you might want to consider creating your own render prop component that provides both.

Caveats

Because context uses reference identity to determine when to re-render, there are some gotchas that could trigger unintentional renders in consumers when a provider's parent re-renders. For example, the code below will re-render all consumers every time the Provider re-renders because a new object is always created for `value`:

```

class App extends React.Component {
  render() {
    return (
      <MyContext.Provider value={{ something: 'something' }}>
        <Toolbar />
      </MyContext.Provider>
    );
  }
}

```

To get around this, lift the value into the parent's state:

```

class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      value: { something: 'something' }
    };
  }

  render() {
    return (
      <MyContext.Provider value={this.state.value}>
        <Toolbar />
      </MyContext.Provider>
    );
  }
}

```

Legacy API

Note

React previously shipped with an experimental context API. The old API will be supported in all 16.x releases, but applications using it should migrate to the new version. The legacy API will be removed in a future major React version. Read the [legacy context docs here](#).

4. Error Boundaries

In the past, JavaScript errors inside components used to corrupt React's internal state and cause it to emit cryptic errors on next renders. These errors were always caused by an earlier error in the application code, but React did not provide a way to handle them gracefully in components, and could not recover from them.

Introducing Error Boundaries

A JavaScript error in a part of the UI shouldn't break the whole app. To solve this problem for React users, React 16 introduces a new concept of an "error boundary".

Error boundaries are React components that **catch JavaScript errors anywhere in their child component tree, log those errors, and display a fallback UI** instead of the component tree that crashed. Error boundaries catch errors during rendering, in lifecycle methods, and in constructors of the whole tree below them.

Note

Error boundaries do **not** catch errors for:

- Event handlers ([learn more](#))
- Asynchronous code (e.g. `setTimeout` or `requestAnimationFrame` callbacks)
- Server side rendering
- Errors thrown in the error boundary itself (rather than its children)

A class component becomes an error boundary if it defines either (or both) of the lifecycle methods `static getDerivedStateFromError()` or `componentDidCatch()`. Use `static getDerivedStateFromError()` to render a fallback UI after an error has been thrown. Use `componentDidCatch()` to log error information.

```
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;
  }
}
```

Then you can use it as a regular component:

```
<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.

Note that **error boundaries only catch errors in the components below them in the tree**. An error boundary can't catch an error within itself. If an error boundary fails trying to render the error message, the error will propagate to the closest error boundary above it. This, too, is similar to how the `catch {}` block works in JavaScript.

Live Demo

Check out [this example of declaring and using an error boundary](#).

Where to Place Error Boundaries

The granularity of error boundaries is up to you. You may wrap top-level route components to display a “Something went wrong” message to the user, just like how server-side frameworks often handle crashes. You may also wrap individual widgets in an error boundary to protect them from crashing the rest of the application.

New Behavior for Uncaught Errors

This change has an important implication. **As of React 16, errors that were not caught by any error boundary will result in unmounting of the whole React component tree.**

We debated this decision, but in our experience it is worse to leave corrupted UI in place than to completely remove it. For example, in a product like Messenger leaving the broken UI visible could lead to somebody sending a message to the wrong person. Similarly, it is worse for a payments app to display a wrong amount than to render nothing.

This change means that as you migrate to React 16, you will likely uncover existing crashes in your application that have been unnoticed before. Adding error boundaries lets you provide better user experience when something goes wrong.

For example, Facebook Messenger wraps content of the sidebar, the info panel, the conversation log, and the message input into separate error boundaries. If some component in one of these UI areas crashes, the rest of them remain interactive.

We also encourage you to use JS error reporting services (or build your own) so that you can learn about unhandled exceptions as they happen in production, and fix them.

Component Stack Traces

React 16 prints all errors that occurred during rendering to the console in development, even if the application accidentally swallows them. In addition to the error message and the JavaScript stack, it also provides component stack traces. Now you can see where exactly in the component tree the failure has happened:

```
► React caught an error thrown by BuggyCounter. You should fix this error in your code. react-dom.development.js:7708
React will try to recreate this component tree from scratch using the error boundary you provided, ErrorBoundary.

Error: I crashed!

The error is located at:
  in BuggyCounter (created by App)
  in ErrorBoundary (created by App)
  in div (created by App)
  in App
```

You can also see the filenames and line numbers in the component stack trace. This works by default in [Create React App](#) projects:

```
► React caught an error thrown by BuggyCounter. You should fix this error in your code. react-dom.development.js:7708
React will try to recreate this component tree from scratch using the error boundary you provided, ErrorBoundary.

Error: I crashed!

The error is located at:
  in BuggyCounter (at App.js:26)
  in ErrorBoundary (at App.js:21)
  in div (at App.js:8)
  in App (at index.js:5)
```

If you don't use Create React App, you can add [this plugin](#) manually to your Babel configuration. Note that it's intended only for development and **must be disabled in production**.

Note

Component names displayed in the stack traces depend on the `Function.name` property. If you support older browsers and devices which may not yet provide this natively (e.g. IE 11), consider including a `Function.name` polyfill in your bundled application, such as `function.name-polyfill`. Alternatively, you may explicitly set the `displayName` property on all your components.

How About try/catch?

`try` / `catch` is great but it only works for imperative code:

```
try {
  showButton();
} catch (error) {
  // ...
}
```

However, React components are declarative and specify *what* should be rendered:

```
<Button />
```

Error boundaries preserve the declarative nature of React, and behave as you would expect. For example, even if an error occurs in a `componentDidUpdate` method caused by a `setState` somewhere deep in the tree, it will still correctly propagate to the closest error boundary.

How About Event Handlers?

Error boundaries **do not** catch errors inside event handlers.

React doesn't need error boundaries to recover from errors in event handlers. Unlike the render method and lifecycle methods, the event handlers don't happen during rendering. So if they throw, React still knows what to display on the screen.

If you need to catch an error inside an event handler, use the regular JavaScript `try` / `catch` statement:

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = { error: null };
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    try {
      // Do something that could throw
    } catch (error) {
      this.setState({ error });
    }
  }

  render() {
    if (this.state.error) {
      return <h1>Caught an error.</h1>;
    }
    return <button onClick={this.handleClick}>Click Me</button>;
  }
}
```

Note that the above example is demonstrating regular JavaScript behavior and doesn't use error boundaries.

Naming Changes from React 15

React 15 included a very limited support for error boundaries under a different method name: `unstable_handleError`. This method no longer works, and you will need to change it to `componentDidCatch` in your code starting from the first 16 beta release.

For this change, we've provided a [codemod](#) to automatically migrate your code.

5. Forwarding Refs

Ref forwarding is a technique for automatically passing a [ref](#) through a component to one of its children. This is typically not necessary for most components in the application. However, it can be useful for some kinds of components, especially in reusable component libraries. The most common scenarios are described below.

Forwarding refs to DOM components

Consider a `FancyButton` component that renders the native `button` DOM element:

```
function FancyButton(props) {
  return (
    <button className="FancyButton">
      {props.children}
    </button>
  );
}
```

React components hide their implementation details, including their rendered output. Other components using `FancyButton` **usually will not need to obtain a ref** to the inner `button` DOM element. This is good because it prevents components from relying on each other's DOM structure too much.

Although such encapsulation is desirable for application-level components like `FeedStory` or `Comment`, it can be inconvenient for highly reusable “leaf” components like `FancyButton` or `MyTextInput`. These components tend to be used throughout the application in a similar manner as a regular DOM `button` and `input`, and accessing their DOM nodes may be unavoidable for managing focus, selection, or animations.

Ref forwarding is an opt-in feature that lets some components take a `ref` they receive, and pass it further down (in other words, “forward” it) to a child.

In the example below, `FancyButton` uses `React.forwardRef` to obtain the `ref` passed to it, and then forward it to the DOM `button` that it renders:

```
const FancyButton = React.forwardRef((props, ref) => (
  <button ref={ref} className="FancyButton">
    {props.children}
  </button>
));

// You can now get a ref directly to the DOM button:
const ref = React.createRef();
<FancyButton ref={ref}>Click me!</FancyButton>;
```

This way, components using `FancyButton` can get a ref to the underlying `button` DOM node and access it if necessary—just like if they used a DOM `button` directly.

Here is a step-by-step explanation of what happens in the above example:

1. We create a **React ref** by calling `React.createRef` and assign it to a `ref` variable.
2. We pass our `ref` down to `<FancyButton ref={ref}>` by specifying it as a JSX attribute.
3. React passes the `ref` to the `(props, ref) => ...` function inside `forwardRef` as a second argument.
4. We forward this `ref` argument down to `<button ref={ref}>` by specifying it as a JSX attribute.
5. When the ref is attached, `ref.current` will point to the `<button>` DOM node.

Note

The second `ref` argument only exists when you define a component with `React.forwardRef` call. Regular function or class components don't receive the `ref` argument, and `ref` is not available in `props` either.

Ref forwarding is not limited to DOM components. You can forward refs to class component instances, too.

Note for component library maintainers

When you start using `forwardRef` in a component library, you should treat it as a breaking change and release a new major version of your library. This is because your library likely has an observably different behavior (such as what refs get assigned to, and what types are exported), and this can break apps and other libraries that depend on the old behavior.

Conditionally applying `React.forwardRef` when it exists is also not recommended for the same reasons: it changes how your library behaves and can break your users' apps when they upgrade React itself.

Forwarding refs in higher-order components

This technique can also be particularly useful with [higher-order components](#) (also known as HOCs). Let's start with an example HOC that logs component props to the console:

```
function logProps(WrappedComponent) {
  class LogProps extends React.Component {
    componentDidUpdate(prevProps) {
      console.log('old props:', prevProps);
      console.log('new props:', this.props);
    }

    render() {
      return <WrappedComponent {...this.props} />;
    }
  }

  return LogProps;
}
```

The "logProps" HOC passes all `props` through to the component it wraps, so the rendered output will be the same. For example, we can use this HOC to log all props that get passed to our "fancy button" component:

```
class FancyButton extends React.Component {
  focus() {
    // ...
  }

  // ...
}

// Rather than exporting FancyButton, we export LogProps.
// It will render a FancyButton though.
export default logProps(FancyButton);
```

There is one caveat to the above example: refs will not get passed through. That's because `ref` is not a prop. Like `key`, it's handled differently by React. If you add a ref to a HOC, the ref will refer to the outermost container component, not the wrapped component.

This means that refs intended for our `FancyButton` component will actually be attached to the `LogProps` component:

```
import FancyButton from './FancyButton';

const ref = React.createRef(); // The FancyButton component we imported is the LogProps HOC.
// Even though the rendered output will be the same,
// Our ref will point to LogProps instead of the inner FancyButton component!
// This means we can't call e.g. ref.current.focus()
<FancyButton label="Click Me" handleClick={handleClick} ref={ref} />;
```

Fortunately, we can explicitly forward refs to the inner `FancyButton` component using the `React.forwardRef` API. `React.forwardRef` accepts a render function that receives `props` and `ref` parameters and returns a React node. For example:

```
function logProps(Component) {
  class LogProps extends React.Component {
    componentDidUpdate(prevProps) {
      console.log('old props:', prevProps);
      console.log('new props:', this.props);
    }

    render() {
      const { forwardedRef, ...rest } = this.props; // Assign the custom prop "forwardedRef" as a ref
      return <Component ref={forwardedRef} {...rest} />;
    }
  }

  // Note the second param "ref" provided by React.forwardRef.
  // We can pass it along to LogProps as a regular prop, e.g. "forwardedRef"
  // And it can then be attached to the Component.
  return React.forwardRef((props, ref) => {
    return <LogProps {...props} forwardedRef={ref} />;
  });
}
```

Displaying a custom name in DevTools

`React.forwardRef` accepts a render function. React DevTools uses this function to determine what to display for the ref forwarding component.

For example, the following component will appear as *"ForwardRef"* in the DevTools:

```
const WrappedComponent = React.forwardRef((props, ref) => {  
  return <LogProps {...props} forwardedRef={ref} />;  
});
```

If you name the render function, DevTools will also include its name (e.g. *"ForwardRef(myFunction)"*):

```
const WrappedComponent = React.forwardRef(function myFunction(props, ref) {  
  return <LogProps {...props} forwardedRef={ref} />;  
});
```

You can even set the function's `displayName` property to include the component you're wrapping:

```
function logProps(Component) {  
  class LogProps extends React.Component {  
    // ...  
  }  
  
  function forwardRef(props, ref) {  
    return <LogProps {...props} forwardedRef={ref} />;  
  }  
  
  // Give this component a more helpful display name in DevTools.  
  // e.g. "ForwardRef(logProps(MyComponent))"  
  const name = Component.displayName || Component.name;  
  forwardRef.displayName = `logProps(${name})`;  
  return React.forwardRef(forwardRef);  
}
```

6. Fragments

A common pattern in React is for a component to return multiple elements. Fragments let you group a list of children without adding extra nodes to the DOM.

```
render() {  
  return (  
    <React.Fragment>  
      <ChildA />  
      <ChildB />  
      <ChildC />  
    </React.Fragment>  
  );  
}
```

There is also a new [short syntax](#) for declaring them.

Motivation

A common pattern is for a component to return a list of children. Take this example React snippet:

```
class Table extends React.Component {  
  render() {  
    return (  
      <table>  
        <tr>  
          <Columns />  
        </tr>  
      </table>  
    );  
  }  
}
```

```
}  
}
```

`<Columns />` would need to return multiple `<td>` elements in order for the rendered HTML to be valid. If a parent div was used inside the `render()` of `<Columns />`, then the resulting HTML will be invalid.

```
class Columns extends React.Component {  
  render() {  
    return (  
      <div>  
        <td>Hello</td>  
        <td>World</td>  
      </div>  
    );  
  }  
}
```

results in a `<Table />` output of:

```
<table>  
  <tr>  
    <div>  
      <td>Hello</td>  
      <td>World</td>  
    </div>  
  </tr>  
</table>;
```

Fragments solve this problem.

Usage

```
class Columns extends React.Component {  
  render() {  
    return (  
      <React.Fragment>  
        <td>Hello</td> <td>World</td>  
      </React.Fragment>  
    );  
  }  
}
```

which results in a correct `<Table />` output of:

```
<table>  
  <tr>  
    <td>Hello</td>  
    <td>World</td>  
  </tr>  
</table>;
```

Short Syntax

There is a new, shorter syntax you can use for declaring fragments. It looks like empty tags:

```
class Columns extends React.Component {  
  render() {  
    return (  
      <>  
        <td>Hello</td> <td>World</td>  
      </>  
    );  
  }  
}
```

You can use `<</>` the same way you'd use any other element except that it doesn't support keys or attributes.

Keyed Fragments

Fragments declared with the explicit `<React.Fragment>` syntax may have keys. A use case for this is mapping a collection to an array of fragments — for example, to create a description list:

```
function Glossary(props) {
  return (
    <dl>
      {props.items.map((item) => (
        // Without the `key`, React will fire a key warning
        <React.Fragment key={item.id}>
          <dt>{item.term}</dt>
          <dd>{item.description}</dd>
        </React.Fragment>
      ))}
    </dl>
  );
}
```

`key` is the only attribute that can be passed to `Fragment`. In the future, we may add support for additional attributes, such as event handlers.

7. Higher-Order Components

A higher-order component (HOC) is an advanced technique in React for reusing component logic. HOCs are not part of the React API, per se. They are a pattern that emerges from React's compositional nature.

Concretely, a **higher-order component is a function that takes a component and returns a new component**.

```
const EnhancedComponent = higherOrderComponent(WrappedComponent);
```

Whereas a component transforms props into UI, a higher-order component transforms a component into another component.

HOCs are common in third-party React libraries, such as Redux's `connect` and Relay's `createFragmentContainer`.

In this document, we'll discuss why higher-order components are useful, and how to write your own.

Use HOCs For Cross-Cutting Concerns

Note

We previously recommended mixins as a way to handle cross-cutting concerns. We've since realized that mixins create more trouble than they are worth. [Read more](#) about why we've moved away from mixins and how you can transition your existing components.

Components are the primary unit of code reuse in React. However, you'll find that some patterns aren't a straightforward fit for traditional components.

For example, say you have a `CommentList` component that subscribes to an external data source to render a list of comments:

```
class CommentList extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = {
      // "DataSource" is some global data source
      comments: DataSource.getComments()
    };
  }

  componentDidMount() {
    // Subscribe to changes
    DataSource.addChangeListener(this.handleChange);
  }

  componentWillUnmount() {
```

```

    // Clean up listener
    DataSource.removeChangeListener(this.handleChange);
  }

  handleChange() {
    // Update component state whenever the data source changes
    this.setState({
      comments: DataSource.getComments()
    });
  }

  render() {
    return (
      <div>
        {this.state.comments.map((comment) => (
          <Comment comment={comment} key={comment.id} />
        ))}
      </div>
    );
  }
}

```

Later, you write a component for subscribing to a single blog post, which follows a similar pattern:

```

class BlogPost extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = {
      blogPost: DataSource.getBlogPost(props.id)
    };
  }

  componentDidMount() {
    DataSource.addChangeListener(this.handleChange);
  }

  componentWillUnmount() {
    DataSource.removeChangeListener(this.handleChange);
  }

  handleChange() {
    this.setState({
      blogPost: DataSource.getBlogPost(this.props.id)
    });
  }

  render() {
    return <TextBlock text={this.state.blogPost} />;
  }
}

```

`CommentList` and `BlogPost` aren't identical — they call different methods on `DataSource`, and they render different output. But much of their implementation is the same:

- On mount, add a change listener to `DataSource`.
- Inside the listener, call `setState` whenever the data source changes.
- On unmount, remove the change listener.

You can imagine that in a large app, this same pattern of subscribing to `DataSource` and calling `setState` will occur over and over again. We want an abstraction that allows us to define this logic in a single place and share it across many components. This is where higher-order components excel.

We can write a function that creates components, like `CommentList` and `BlogPost`, that subscribe to `DataSource`. The function will accept as one of its arguments a child component that receives the subscribed data as a prop. Let's call the function `withSubscription`:

```

const CommentListWithSubscription = withSubscription(CommentList, (DataSource) => {
  return DataSource.getComments()
});

const BlogPostWithSubscription = withSubscription(BlogPost, (DataSource, props) => {
  return DataSource.getBlogPost(props.id)
});

```


The first parameter is the wrapped component. The second parameter retrieves the data we're interested in, given a `DataSource` and the current props.

When `CommentListWithSubscription` and `BlogPostWithSubscription` are rendered, `CommentList` and `BlogPost` will be passed a `data` prop with the most current data retrieved from `DataSource`:

```
// This function takes a component...
function withSubscription(WrappedComponent, selectData) {
  // ...and returns another component...
  return class extends React.Component {
    constructor(props) {
      super(props);
      this.handleChange = this.handleChange.bind(this);
      this.state = {
        data: selectData(DataSource, props)
      };
    }

    componentDidMount() {
      // ... that takes care of the subscription...
      DataSource.addChangeListener(this.handleChange);
    }

    componentWillUnmount() {
      DataSource.removeChangeListener(this.handleChange);
    }

    handleChange() {
      this.setState({
        data: selectData(DataSource, this.props)
      });
    }

    render() {
      // ... and renders the wrapped component with the fresh data!
      // Notice that we pass through any additional props
      return <WrappedComponent data={this.state.data} {...this.props} />;
    }
  };
}
```

Note that a HOC doesn't modify the input component, nor does it use inheritance to copy its behavior. Rather, a HOC *composes* the original component by *wrapping* it in a container component. A HOC is a pure function with zero side-effects.

And that's it! The wrapped component receives all the props of the container, along with a new prop, `data`, which it uses to render its output. The HOC isn't concerned with how or why the data is used, and the wrapped component isn't concerned with where the data came from.

Because `withSubscription` is a normal function, you can add as many or as few arguments as you like. For example, you may want to make the name of the `data` prop configurable, to further isolate the HOC from the wrapped component. Or you could accept an argument that configures `shouldComponentUpdate`, or one that configures the data source. These are all possible because the HOC has full control over how the component is defined.

Like components, the contract between `withSubscription` and the wrapped component is entirely props-based. This makes it easy to swap one HOC for a different one, as long as they provide the same props to the wrapped component. This may be useful if you change data-fetching libraries, for example.

Don't Mutate the Original Component. Use Composition.

Resist the temptation to modify a component's prototype (or otherwise mutate it) inside a HOC.

```
function logProps(InputComponent) {
  InputComponent.prototype.componentDidUpdate = function (prevProps) {
    console.log('Current props: ', this.props);
    console.log('Previous props: ', prevProps);
  };
  // The fact that we're returning the original input is a hint that it has
  // been mutated.
  return InputComponent;
}

// EnhancedComponent will log whenever props are received
const EnhancedComponent = logProps(InputComponent);
```

There are a few problems with this. One is that the input component cannot be reused separately from the enhanced component. More crucially, if you apply another HOC to `EnhancedComponent` that also mutates `componentDidUpdate`, the first HOC's functionality will be overridden! This HOC also won't work with function components, which do not have lifecycle methods.

Mutating HOCs are a leaky abstraction—the consumer must know how they are implemented in order to avoid conflicts with other HOCs.

Instead of mutation, HOCs should use composition, by wrapping the input component in a container component:

```
function logProps(WrappedComponent) {
  return class extends React.Component {
    componentDidUpdate(prevProps) {
      console.log('Current props: ', this.props);
      console.log('Previous props: ', prevProps);
    }
    render() {
      // Wraps the input component in a container, without mutating it. Good!
      return <WrappedComponent {...this.props} />;
    }
  };
}
```

This HOC has the same functionality as the mutating version while avoiding the potential for clashes. It works equally well with class and function components. And because it's a pure function, it's composable with other HOCs, or even with itself.

You may have noticed similarities between HOCs and a pattern called **container components**. Container components are part of a strategy of separating responsibility between high-level and low-level concerns. Containers manage things like subscriptions and state, and pass props to components that handle things like rendering UI. HOCs use containers as part of their implementation. You can think of HOCs as parameterized container component definitions.

Convention: Pass Unrelated Props Through to the Wrapped Component

HOCs add features to a component. They shouldn't drastically alter its contract. It's expected that the component returned from a HOC has a similar interface to the wrapped component.

HOCs should pass through props that are unrelated to its specific concern. Most HOCs contain a render method that looks something like this:

```
render() {
  // Filter out extra props that are specific to this HOC and shouldn't be
  // passed through
  const { extraProp, ...passThroughProps } = this.props;

  // Inject props into the wrapped component. These are usually state values or
  // instance methods.
  const injectedProp = someStateOrInstanceMethod;

  // Pass props to wrapped component
  return (
    <WrappedComponent injectedProp={injectedProp} {...passThroughProps} />
  );
}
```

This convention helps ensure that HOCs are as flexible and reusable as possible.

Convention: Maximizing Composability

Not all HOCs look the same. Sometimes they accept only a single argument, the wrapped component:

```
const NavbarWithRouter = withRouter(Navbar);
```

Usually, HOCs accept additional arguments. In this example from Relay, a config object is used to specify a component's data dependencies:

```
const CommentWithRelay = Relay.createContainer(Comment, config);
```

The most common signature for HOCs looks like this:

```
// React Redux's `connect`  
const ConnectedComment = connect(commentSelector, commentActions)(CommentList);
```

What?! If you break it apart, it's easier to see what's going on.

```
// connect is a function that returns another function  
const enhance = connect(commentListSelector, commentListActions);  
// The returned function is a HOC, which returns a component that is connected  
// to the Redux store  
const ConnectedComment = enhance(CommentList);
```

In other words, `connect` is a higher-order function that returns a higher-order component!

This form may seem confusing or unnecessary, but it has a useful property. Single-argument HOCs like the one returned by the `connect` function have the signature `Component => Component`. Functions whose output type is the same as its input type are really easy to compose together.

```
// Instead of doing this...  
const EnhancedComponent = withRouter(connect(commentSelector)(WrappedComponent));  
  
// ... you can use a function composition utility  
// compose(f, g, h) is the same as (...args) => f(g(h(...args)))  
const enhance = compose(  
  // These are both single-argument HOCs  
  withRouter,  
  connect(commentSelector)  
);  
const EnhancedComponent = enhance(WrappedComponent);
```

(This same property also allows `connect` and other enhancer-style HOCs to be used as decorators, an experimental JavaScript proposal.)

The `compose` utility function is provided by many third-party libraries including [lodash](#) (as `lodash.flowRight`), [Redux](#), and [Ramda](#).

Convention: Wrap the Display Name for Easy Debugging

The container components created by HOCs show up in the [React Developer Tools](#) like any other component. To ease debugging, choose a display name that communicates that it's the result of a HOC.

The most common technique is to wrap the display name of the wrapped component. So if your higher-order component is named `WithSubscription`, and the wrapped component's display name is `CommentList`, use the display name `WithSubscription(CommentList)`:

```
function withSubscription(WrappedComponent) {  
  class WithSubscription extends React.Component {  
    /* ... */  
  }  
  WithSubscription.displayName = `WithSubscription(${getDisplayName(WrappedComponent)})`;  
  return WithSubscription;  
}  
  
function getDisplayName(WrappedComponent) {  
  return WrappedComponent.displayName || WrappedComponent.name || 'Component';  
}
```

Caveats

Higher-order components come with a few caveats that aren't immediately obvious if you're new to React.

Don't Use HOCs Inside the render Method

React's diffing algorithm (called [Reconciliation](#)) uses component identity to determine whether it should update the existing subtree or throw it away and mount a new one. If the component returned from `render` is identical (`===`) to the component from the previous render, React recursively updates the subtree by diffing it with the new one. If they're not equal, the previous subtree is unmounted completely.

Normally, you shouldn't need to think about this. But it matters for HOCs because it means you can't apply a HOC to a component within the render method of a component:

```
render() {
  // A new version of EnhancedComponent is created on every render
  // EnhancedComponent1 !== EnhancedComponent2
  const EnhancedComponent = enhance(MyComponent);
  // That causes the entire subtree to unmount/remount each time!
  return <EnhancedComponent />;
}
```

The problem here isn't just about performance — remounting a component causes the state of that component and all of its children to be lost.

Instead, apply HOCs outside the component definition so that the resulting component is created only once. Then, its identity will be consistent across renders. This is usually what you want, anyway.

In those rare cases where you need to apply a HOC dynamically, you can also do it inside a component's lifecycle methods or its constructor.

Static Methods Must Be Copied Over

Sometimes it's useful to define a static method on a React component. For example, Relay containers expose a static method `getFragment` to facilitate the composition of GraphQL fragments.

When you apply a HOC to a component, though, the original component is wrapped with a container component. That means the new component does not have any of the static methods of the original component.

```
// Define a static method
WrappedComponent.staticMethod = function () {
  /*...*/
};
// Now apply a HOC
const EnhancedComponent = enhance(WrappedComponent);

// The enhanced component has no static method
typeof EnhancedComponent.staticMethod === 'undefined'; // true
```

To solve this, you could copy the methods onto the container before returning it:

```
function enhance(WrappedComponent) {
  class Enhance extends React.Component {
    /*...*/
  }
  // Must know exactly which method(s) to copy :(
  Enhance.staticMethod = WrappedComponent.staticMethod;
  return Enhance;
}
```

However, this requires you to know exactly which methods need to be copied. You can use [hoist-non-react-statics](#) to automatically copy all non-React static methods:

```
import hoistNonReactStatic from 'hoist-non-react-statics';
function enhance(WrappedComponent) {
  class Enhance extends React.Component {
    /*...*/
  }
  hoistNonReactStatic(Enhance, WrappedComponent);
  return Enhance;
}
```

Another possible solution is to export the static method separately from the component itself.

```
// Instead of...
MyComponent.someFunction = someFunction;
export default MyComponent;

// ...export the method separately...
export { someFunction };

// ...and in the consuming module, import both
import MyComponent, { someFunction } from './MyComponent.js';
```

Refs Aren't Passed Through

While the convention for higher-order components is to pass through all props to the wrapped component, this does not work for refs. That's because `ref` is not really a prop — like `key`, it's handled specially by React. If you add a ref to an element whose component is the result of a HOC, the ref refers to an instance of the outermost container component, not the wrapped component.

The solution for this problem is to use the `React.forwardRef` API (introduced with React 16.3). [Learn more about it in the forwarding refs section.](#)

8. Integrating with Other Libraries

React can be used in any web application. It can be embedded in other applications and, with a little care, other applications can be embedded in React. This guide will examine some of the more common use cases, focusing on integration with [jQuery](#) and [Backbone](#), but the same ideas can be applied to integrating components with any existing code.

Integrating with DOM Manipulation Plugins

React is unaware of changes made to the DOM outside of React. It determines updates based on its own internal representation, and if the same DOM nodes are manipulated by another library, React gets confused and has no way to recover.

This does not mean it is impossible or even necessarily difficult to combine React with other ways of affecting the DOM, you just have to be mindful of what each is doing.

The easiest way to avoid conflicts is to prevent the React component from updating. You can do this by rendering elements that React has no reason to update, like an empty `<div />`.

How to Approach the Problem

To demonstrate this, let's sketch out a wrapper for a generic jQuery plugin.

We will attach a `ref` to the root DOM element. Inside `componentDidMount`, we will get a reference to it so we can pass it to the jQuery plugin.

To prevent React from touching the DOM after mounting, we will return an empty `<div />` from the `render()` method. The `<div />` element has no properties or children, so React has no reason to update it, leaving the jQuery plugin free to manage that part of the DOM:

```
class SomePlugin extends React.Component {
  componentDidMount() {
    this.$el = $(this.el);    this.$el.somePlugin(); }

  componentWillUnmount() {
    this.$el.somePlugin('destroy'); }

  render() {
    return <div ref={el => this.el = el} />;
  }
}
```

Note that we defined both `componentDidMount` and `componentWillUnmount` [lifecycle methods](#). Many jQuery plugins attach event listeners to the DOM so it's important to detach them in `componentWillUnmount`. If the plugin does not provide a method for cleanup, you will probably have to provide your own, remembering to remove any event listeners the plugin registered to prevent memory leaks.

Integrating with jQuery Chosen Plugin

For a more concrete example of these concepts, let's write a minimal wrapper for the plugin [Chosen](#), which augments `<select>` inputs.

Note:

Just because it's possible, doesn't mean that it's the best approach for React apps. We encourage you to use React components when you can. React components are easier to reuse in React applications, and often provide more control over their behavior and appearance.

First, let's look at what Chosen does to the DOM.

If you call it on a `<select>` DOM node, it reads the attributes off of the original DOM node, hides it with an inline style, and then appends a separate DOM node with its own visual representation right after the `<select>`. Then it fires jQuery events to notify us about the changes.

Let's say that this is the API we're striving for with our `<Chosen>` wrapper React component:

```
function Example() {
  return (
    <Chosen onChange={value => console.log(value)}>
      <option>vanilla</option>
      <option>chocolate</option>
      <option>strawberry</option>
    </Chosen>
  );
}
```

We will implement it as an [uncontrolled component](#) for simplicity.

First, we will create an empty component with a `render()` method where we return `<select>` wrapped in a `<div>`:

```
class Chosen extends React.Component {
  render() {
    return (
      <div>
        <select className="Chosen-select" ref={el => (this.el = el)}>
          {this.props.children}
        </select>
      </div>
    );
  }
}
```

Notice how we wrapped `<select>` in an extra `<div>`. This is necessary because Chosen will append another DOM element right after the `<select>` node we passed to it. However, as far as React is concerned, `<div>` always only has a single child. This is how we ensure that React updates won't conflict with the extra DOM node appended by Chosen. It is important that if you modify the DOM outside of React flow, you must ensure React doesn't have a reason to touch those DOM nodes.

Next, we will implement the lifecycle methods. We need to initialize Chosen with the ref to the `<select>` node in `componentDidMount`, and tear it down in `componentWillUnmount`:

```
componentDidMount() {
  this.$el = $(this.el); this.$el.chosen();
}

componentWillUnmount() {
  this.$el.chosen('destroy');
}
```

Note that React assigns no special meaning to the `this.el` field. It only works because we have previously assigned this field from a `ref` in the `render()` method:

```
<select className="Chosen-select" ref={el => this.el = el}>
```

This is enough to get our component to render, but we also want to be notified about the value changes. To do this, we will subscribe to the jQuery `change` event on the `<select>` managed by Chosen.

We won't pass `this.props.onChange` directly to Chosen because component's props might change over time, and that includes event handlers. Instead, we will declare a `handleChange()` method that calls `this.props.onChange`, and subscribe it to the jQuery `change` event:

```
componentDidMount() {
  this.$el = $(this.el);
  this.$el.chosen();

  this.handleChange = this.handleChange.bind(this);
  this.$el.on('change', this.handleChange);
}

componentWillUnmount() {
  this.$el.off('change', this.handleChange);
  this.$el.chosen('destroy');
}

handleChange(e) {
  this.props.onChange(e.target.value);
}
```

Finally, there is one more thing left to do. In React, props can change over time. For example, the `<Chosen>` component can get different children if parent component's state changes. This means that at integration points it is important that we manually update the DOM in response to prop updates, since we no longer let React manage the DOM for us.

Chosen's documentation suggests that we can use jQuery `trigger()` API to notify it about changes to the original DOM element. We will let React take care of updating `this.props.children` inside `<select>`, but we will also add a `componentDidUpdate()` lifecycle method that notifies Chosen about changes in the children list:

```
componentDidUpdate(prevProps) {
  if (prevProps.children !== this.props.children) {
    this.$el.trigger("chosen:updated");
  }
}
```

This way, Chosen will know to update its DOM element when the `<select>` children managed by React change.

The complete implementation of the `Chosen` component looks like this:

```
class Chosen extends React.Component {
  componentDidMount() {
    this.$el = $(this.el);
    this.$el.chosen();

    this.handleChange = this.handleChange.bind(this);
    this.$el.on('change', this.handleChange);
  }

  componentDidUpdate(prevProps) {
    if (prevProps.children !== this.props.children) {
      this.$el.trigger('chosen:updated');
    }
  }

  componentWillUnmount() {
    this.$el.off('change', this.handleChange);
    this.$el.chosen('destroy');
  }

  handleChange(e) {
    this.props.onChange(e.target.value);
  }

  render() {
    return (
      <div>
        <select className="Chosen-select" ref={(el) => (this.el = el)}>
          {this.props.children}
        </select>
      </div>
    );
  }
}
```

Integrating with Other View Libraries

React can be embedded into other applications thanks to the flexibility of `createRoot()`.

Although React is commonly used at startup to load a single root React component into the DOM, `createRoot()` can also be called multiple times for independent parts of the UI which can be as small as a button, or as large as an app.

In fact, this is exactly how React is used at Facebook. This lets us write applications in React piece by piece, and combine them with our existing server-generated templates and other client-side code.

Replacing String-Based Rendering with React

A common pattern in older web applications is to describe chunks of the DOM as a string and insert it into the DOM like so: `$el.html(htmlString)`. These points in a codebase are perfect for introducing React. Just rewrite the string based rendering as a React component.

So the following jQuery implementation...

```
$('#container').html('<button id="btn">Say Hello</button>');
$('#btn').click(function() {
  alert('Hello!');
});
```

...could be rewritten using a React component:

```
function Button() {
  return <button id="btn">Say Hello</button>;
}

$('#btn').click(function() {
  alert('Hello!');
});
```

From here you could start moving more logic into the component and begin adopting more common React practices. For example, in components it is best not to rely on IDs because the same component can be rendered multiple times. Instead, we will use the [React event system](#) and register the click handler directly on the React `<button>` element:

```
function Button(props) {
  return <button onClick={props.onClick}>Say Hello</button>;
}

function HelloButton() {
  function handleClick() { alert('Hello!'); }
  return <Button onClick={handleClick} />;
}
```

Try it on CodePen

You can have as many such isolated components as you like, and use `ReactDOM.createRoot()` to render them to different DOM containers. Gradually, as you convert more of your app to React, you will be able to combine them into larger components, and move some of the `ReactDOM.createRoot()` calls up the hierarchy.

Embedding React in a Backbone View

[Backbone](#) views typically use HTML strings, or string-producing template functions, to create the content for their DOM elements. This process, too, can be replaced with rendering a React component.

Below, we will create a Backbone view called `ParagraphView`. It will override Backbone's `render()` function to render a React `<Paragraph>` component into the DOM element provided by Backbone (`this.el`). Here, too, we are using `ReactDOM.createRoot()`:

```
function Paragraph(props) {
  return <p>{props.text}</p>;
}

const ParagraphView = Backbone.View.extend({
  initialize(options) {
    this.reactRoot = ReactDOM.createRoot(this.el);
  },
  render() {
    const text = this.model.get('text');
```



```

    this.reactRoot.render(<Paragraph text={text} />);
    return this;
  },
  remove() {
    this.reactRoot.unmount();
    Backbone.View.prototype.remove.call(this);
  }
});

```

It is important that we also call `root.unmount()` in the `remove` method so that React unregisters event handlers and other resources associated with the component tree when it is detached.

When a component is removed *from within* a React tree, the cleanup is performed automatically, but because we are removing the entire tree by hand, we must call this method.

Integrating with Model Layers

While it is generally recommended to use unidirectional data flow such as [React state](#), [Flux](#), or [Redux](#), React components can use a model layer from other frameworks and libraries.

Using Backbone Models in React Components

The simplest way to consume [Backbone](#) models and collections from a React component is to listen to the various change events and manually force an update.

Components responsible for rendering models would listen to `'change'` events, while components responsible for rendering collections would listen for `'add'` and `'remove'` events. In both cases, call `this.forceUpdate()` to rerender the component with the new data.

In the example below, the `List` component renders a Backbone collection, using the `Item` component to render individual items.

```

class Item extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
  }

  handleChange() {
    this.forceUpdate();
  }

  componentDidMount() {
    this.props.model.on('change', this.handleChange);
  }

  componentWillUnmount() {
    this.props.model.off('change', this.handleChange);
  }

  render() {
    return <li>{this.props.model.get('text')}</li>;
  }
}

class List extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
  }

  handleChange() {
    this.forceUpdate();
  }

  componentDidMount() {
    this.props.collection.on('add', 'remove', this.handleChange);
  }

  componentWillUnmount() {
    this.props.collection.off('add', 'remove', this.handleChange);
  }

  render() {
    return (
      <ul>
        {this.props.collection.map((model) => (
          <Item key={model.cid} model={model} />
        ))}
      </ul>
    );
  }
}

```

```

    })}
  </ul>
);
}
}

```

Extracting Data from Backbone Models

The approach above requires your React components to be aware of the Backbone models and collections. If you later plan to migrate to another data management solution, you might want to concentrate the knowledge about Backbone in as few parts of the code as possible.

One solution to this is to extract the model's attributes as plain data whenever it changes, and keep this logic in a single place. The following is a [higher-order component](#) that extracts all attributes of a Backbone model into state, passing the data to the wrapped component.

This way, only the higher-order component needs to know about Backbone model internals, and most components in the app can stay agnostic of Backbone.

In the example below, we will make a copy of the model's attributes to form the initial state. We subscribe to the `change` event (and unsubscribe on unmounting), and when it happens, we update the state with the model's current attributes. Finally, we make sure that if the `model` prop itself changes, we don't forget to unsubscribe from the old model, and subscribe to the new one.

Note that this example is not meant to be exhaustive with regards to working with Backbone, but it should give you an idea for how to approach this in a generic way:

```

function connectToBackboneModel(WrappedComponent) {
  return class BackboneComponent extends React.Component {
    constructor(props) {
      super(props);
      this.state = Object.assign({}, props.model.attributes);
      this.handleChange = this.handleChange.bind(this);
    }

    componentDidMount() {
      this.props.model.on('change', this.handleChange);
    }

    componentWillReceiveProps(nextProps) {
      this.setState(Object.assign({}, nextProps.model.attributes));
      if (nextProps.model !== this.props.model) {
        this.props.model.off('change', this.handleChange);
        nextProps.model.on('change', this.handleChange);
      }
    }

    componentWillUnmount() {
      this.props.model.off('change', this.handleChange);
    }

    handleChange(model) {
      this.setState(model.changedAttributes());
    }

    render() {
      const propsExceptModel = Object.assign({}, this.props);
      delete propsExceptModel.model;
      return <WrappedComponent {...propsExceptModel} {...this.state} />;
    }
  };
}

```

To demonstrate how to use it, we will connect a `NameInput` React component to a Backbone model, and update its `firstName` attribute every time the input changes:

```

function NameInput(props) {
  return (
    <p>
      <input value={props.firstName} onChange={props.handleChange} /> <br /> My name is {props.firstName}.
    </p>
  );
}

```

```
const BackboneNameInput = connectToBackboneModel(NameInput);
function Example(props) {
  function handleChange(e) {
    props.model.set('firstName', e.target.value);
  }

  return <BackboneNameInput model={props.model} handleChange={handleChange} />;
}

const model = new Backbone.Model({ firstName: 'Frodo' });
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Example model={model} />);
```

9. JSX In Depth

Fundamentally, JSX just provides syntactic sugar for the `React.createElement(component, props, ...children)` function. The JSX code:

```
<MyButton color="blue" shadowSize={2}>
  Click Me
</MyButton>
```

compiles into:

```
React.createElement(
  MyButton,
  {color: 'blue', shadowSize: 2},
  'Click Me'
)
```

You can also use the self-closing form of the tag if there are no children. So:

```
<div className="sidebar" />
```

compiles into:

```
React.createElement(
  'div',
  {className: 'sidebar'}
)
```

If you want to test out how some specific JSX is converted into JavaScript, you can try out [the online Babel compiler](#).

Specifying The React Element Type

The first part of a JSX tag determines the type of the React element.

Capitalized types indicate that the JSX tag is referring to a React component. These tags get compiled into a direct reference to the named variable, so if you use the JSX `<Foo />` expression, `Foo` must be in scope.

React Must Be in Scope

Since JSX compiles into calls to `React.createElement`, the `React` library must also always be in scope from your JSX code.

For example, both of the imports are necessary in this code, even though `React` and `CustomButton` are not directly referenced from JavaScript:

```
import React from 'react';
import CustomButton from './CustomButton';
function WarningButton() {
  // return React.createElement(CustomButton, {color: 'red'}, null);
  return <CustomButton color="red" />;
}
```

If you don't use a JavaScript bundler and loaded React from a `<script>` tag, it is already in scope as the `React` global.

Using Dot Notation for JSX Type

You can also refer to a React component using dot-notation from within JSX. This is convenient if you have a single module that exports many React components. For example, if `MyComponents.DatePicker` is a component, you can use it directly from JSX with:

```
import React from 'react';

const MyComponents = {
  DatePicker: function DatePicker(props) {
    return <div>Imagine a {props.color} datepicker here.</div>;
  }
};

function BlueDatePicker() {
  return <MyComponents.DatePicker color="blue" />;
}
```

User-Defined Components Must Be Capitalized

When an element type starts with a lowercase letter, it refers to a built-in component like `<div>` or `` and results in a string `'div'` or `'span'` passed to `React.createElement`. Types that start with a capital letter like `<Foo />` compile to `React.createElement(Foo)` and correspond to a component defined or imported in your JavaScript file.

We recommend naming components with a capital letter. If you do have a component that starts with a lowercase letter, assign it to a capitalized variable before using it in JSX.

For example, this code will not run as expected:

```
import React from 'react';

// Wrong! This is a component and should have been capitalized:
function hello(props) {
  // Correct! This use of <div> is legitimate because div is a valid HTML tag:
  return <div>Hello {props.toWhat}</div>;
}

function HelloWorld() {
  // Wrong! React thinks <hello /> is an HTML tag because it's not capitalized:
  return <hello toWhat="World" />;
}
```

To fix this, we will rename `hello` to `Hello` and use `<Hello />` when referring to it:

```
import React from 'react';

// Correct! This is a component and should be capitalized:
function Hello(props) {
  // Correct! This use of <div> is legitimate because div is a valid HTML tag:
  return <div>Hello {props.toWhat}</div>;
}

function HelloWorld() {
  // Correct! React knows <Hello /> is a component because it's capitalized.
  return <Hello toWhat="World" />;
}
```

Choosing the Type at Runtime

You cannot use a general expression as the React element type. If you do want to use a general expression to indicate the type of the element, just assign it to a capitalized variable first. This often comes up when you want to render a different component based on a prop:

```
import React from 'react';
import { PhotoStory, VideoStory } from './stories';
```

```
const components = {
  photo: PhotoStory,
  video: VideoStory
};

function Story(props) {
  // Wrong! JSX type can't be an expression.
  return <components[props.storyType] story={props.story} />;
}
```

To fix this, we will assign the type to a capitalized variable first:

```
import React from 'react';
import { PhotoStory, VideoStory } from './stories';

const components = {
  photo: PhotoStory,
  video: VideoStory
};

function Story(props) {
  // Correct! JSX type can be a capitalized variable.
  const SpecificStory = components[props.storyType];
  return <SpecificStory story={props.story} />;
}
```

Props in JSX

There are several different ways to specify props in JSX.

JavaScript Expressions as Props

You can pass any JavaScript expression as a prop, by surrounding it with `{ }`. For example, in this JSX:

```
<MyComponent foo={1 + 2 + 3 + 4} />
```

For `MyComponent`, the value of `props.foo` will be `10` because the expression `1 + 2 + 3 + 4` gets evaluated.

`if` statements and `for` loops are not expressions in JavaScript, so they can't be used in JSX directly. Instead, you can put these in the surrounding code. For example:

```
function NumberDescriber(props) {
  let description;
  if (props.number % 2 == 0) {
    description = <strong>even</strong>;
  } else {
    description = <i>odd</i>;
  }
  return (
    <div>
      {props.number} is an {description} number
    </div>
  );
}
```

You can learn more about [conditional rendering](#) and [loops](#) in the corresponding sections.

String Literals

You can pass a string literal as a prop. These two JSX expressions are equivalent:

```
<MyComponent message="hello world" />
<MyComponent message={'hello world'} />
```

When you pass a string literal, its value is HTML-unesaped. So these two JSX expressions are equivalent:

```
<MyComponent message="&lt;3" />
<MyComponent message={'<3'} />
```

This behavior is usually not relevant. It's only mentioned here for completeness.

Props Default to “True”

If you pass no value for a prop, it defaults to `true`. These two JSX expressions are equivalent:

```
<MyTextBox autocomplete />
<MyTextBox autocomplete={true} />
```

In general, we don't recommend *not* passing a value for a prop, because it can be confused with the [ES6 object shorthand](#) `{foo}` which is short for `{foo: foo}` rather than `{foo: true}`. This behavior is just there so that it matches the behavior of HTML.

Spread Attributes

If you already have `props` as an object, and you want to pass it in JSX, you can use `...` as a “spread” syntax to pass the whole props object. These two components are equivalent:

```
function App1() {
  return <Greeting firstName="Ben" lastName="Hector" />;
}

function App2() {
  const props = { firstName: 'Ben', lastName: 'Hector' };
  return <Greeting {...props} />;
}
```

You can also pick specific props that your component will consume while passing all other props using the spread syntax.

```
const Button = (props) => {
  const { kind, ...other } = props;
  const className = kind === 'primary' ? 'PrimaryButton' : 'SecondaryButton';
  return <button className={className} {...other} />;
};

const App = () => {
  return (
    <div>
      <Button kind="primary" onClick={() => console.log('clicked!')}>
        Hello World!
      </Button>
    </div>
  );
};
```

In the example above, the `kind` prop is safely consumed and *is not* passed on to the `<button>` element in the DOM. All other props are passed via the `...other` object making this component really flexible. You can see that it passes an `onClick` and `children` props.

Spread attributes can be useful but they also make it easy to pass unnecessary props to components that don't care about them or to pass invalid HTML attributes to the DOM. We recommend using this syntax sparingly.

Children in JSX

In JSX expressions that contain both an opening tag and a closing tag, the content between those tags is passed as a special prop: `props.children`. There are several different ways to pass children:

String Literals

You can put a string between the opening and closing tags and `props.children` will just be that string. This is useful for many of the built-in HTML elements. For example:

```
<MyComponent>Hello world!</MyComponent>
```

This is valid JSX, and `props.children` in `MyComponent` will simply be the string `"Hello world!"`. HTML is unescaped, so you can generally write JSX just like you would write HTML in this way:

```
<div>This is valid HTML &amp; JSX at the same time.</div>
```

JSX removes whitespace at the beginning and ending of a line. It also removes blank lines. New lines adjacent to tags are removed; new lines that occur in the middle of string literals are condensed into a single space. So these all render to the same thing:

```
<div>Hello World</div>
<div>Hello World</div>
<div>HelloWorld</div>
<div>Hello World</div>
```

JSX Children

You can provide more JSX elements as the children. This is useful for displaying nested components:

```
<MyContainer>
  <MyFirstComponent />
  <MySecondComponent />
</MyContainer>
```

You can mix together different types of children, so you can use string literals together with JSX children. This is another way in which JSX is like HTML, so that this is both valid JSX and valid HTML:

```
<div>
  Here is a list:
  <ul>
    <li>Item 1</li>
    <li>Item 2</li>
  </ul>
</div>
```

A React component can also return an array of elements:

```
render() {
  // No need to wrap list items in an extra element!
  return [
    // Don't forget the keys :)
    <li key="A">First item</li>,
    <li key="B">Second item</li>,
    <li key="C">Third item</li>,
  ];
}
```

JavaScript Expressions as Children

You can pass any JavaScript expression as children, by enclosing it within `{ }`. For example, these expressions are equivalent:

```
<MyComponent>foo</MyComponent>
<MyComponent>{'foo'}</MyComponent>
```

This is often useful for rendering a list of JSX expressions of arbitrary length. For example, this renders an HTML list:

```
function Item(props) {
  return <li>{props.message}</li>;
}

function TodoList() {
  const todos = ['finish doc', 'submit pr', 'nag dan to review'];
  return (
    <ul>
      {todos.map((message) => (
        <Item key={message} message={message} />
      ))}
    </ul>
  );
}
```

JavaScript expressions can be mixed with other types of children. This is often useful in lieu of string templates:

```
function Hello(props) {
  return <div>Hello {props.addressee}</div>;
}
```

Functions as Children

Normally, JavaScript expressions inserted in JSX will evaluate to a string, a React element, or a list of those things. However, `props.children` works just like any other prop in that it can pass any sort of data, not just the sorts that React knows how to render. For example, if you have a custom component, you could have it take a callback as `props.children`:

```
// Calls the children callback numTimes to produce a repeated component
function Repeat(props) {
  let items = [];
  for (let i = 0; i < props.numTimes; i++) {
    items.push(props.children(i));
  }
  return <div>{items}</div>;
}

function ListOfTenThings() {
  return <Repeat numTimes={10}> {(index) => <div key={index}>This is item {index} in the list</div> } </Repeat>;
}
```

Children passed to a custom component can be anything, as long as that component transforms them into something React can understand before rendering. This usage is not common, but it works if you want to stretch what JSX is capable of.

Booleans, Null, and Undefined Are Ignored

`false`, `null`, `undefined`, and `true` are valid children. They simply don't render. These JSX expressions will all render to the same thing:

```
<div />
<div></div>
<div>{false}</div>
<div>{null}</div>
<div>{undefined}</div>
<div>{true}</div>
```

This can be useful to conditionally render React elements. This JSX renders the `<Header />` component only if `showHeader` is `true`:

```
<div>
  {showHeader && <Header />}
  <Content />
</div>
```

One caveat is that some “falsy” values, such as the `0` number, are still rendered by React. For example, this code will not behave as you might expect because `0` will be printed when `props.messages` is an empty array:


```
<div>
  {
    props.messages.length &&
    <MessageList messages={props.messages} />
  }
</div>
```

To fix this, make sure that the expression before `&&` is always boolean:

```
<div>
  {props.messages.length > 0 && <MessageList messages={props.messages} />}
</div>
```

Conversely, if you want a value like `false`, `true`, `null`, or `undefined` to appear in the output, you have to convert it to a string first:

```
<div>My JavaScript variable is {String(myVariable)}.</div>
```

10. Optimizing Performance

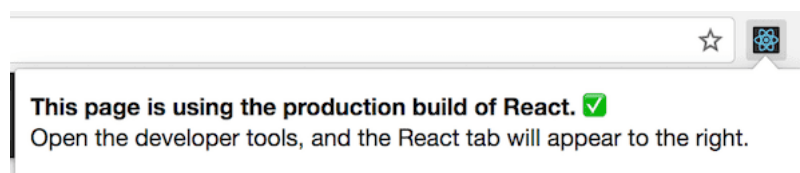
Internally, React uses several clever techniques to minimize the number of costly DOM operations required to update the UI. For many applications, using React will lead to a fast user interface without doing much work to specifically optimize for performance. Nevertheless, there are several ways you can speed up your React application.

Use the Production Build

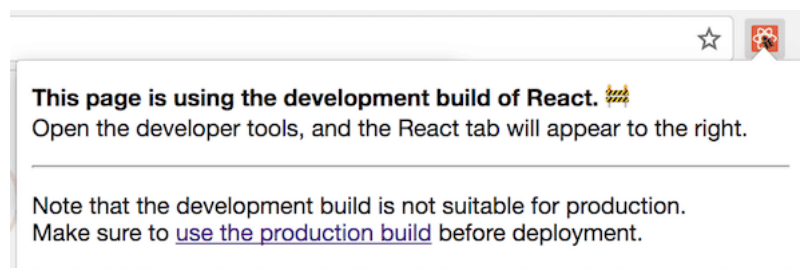
If you're benchmarking or experiencing performance problems in your React apps, make sure you're testing with the minified production build.

By default, React includes many helpful warnings. These warnings are very useful in development. However, they make React larger and slower so you should make sure to use the production version when you deploy the app.

If you aren't sure whether your build process is set up correctly, you can check it by installing [React Developer Tools for Chrome](#). If you visit a site with React in production mode, the icon will have a dark background:



If you visit a site with React in development mode, the icon will have a red background:



It is expected that you use the development mode when working on your app, and the production mode when deploying your app to the users.

You can find instructions for building your app for production below.

Create React App

If your project is built with [Create React App](#), run:

```
npm run build
```

This will create a production build of your app in the `build/` folder of your project.

Remember that this is only necessary before deploying to production. For normal development, use `npm start`.

Single-File Builds

We offer production-ready versions of React and React DOM as single files:

```
<script src="https://unpkg.com/react@18/umd/react.production.min.js"></script>
<script src="https://unpkg.com/react-dom@18/umd/react-dom.production.min.js"></script>
```

Remember that only React files ending with `.production.min.js` are suitable for production.

Brunch

For the most efficient Brunch production build, install the [terser-brunch](#) plugin:

```
# If you use npm
npm install --save-dev terser-brunch

# If you use Yarn
yarn add --dev terser-brunch
```

Then, to create a production build, add the `-p` flag to the `build` command:

```
brunch build -p
```

Remember that you only need to do this for production builds. You shouldn't pass the `-p` flag or apply this plugin in development, because it will hide useful React warnings and make the builds much slower.

Browserify

For the most efficient Browserify production build, install a few plugins:

```
# If you use npm
npm install --save-dev envify terser uglifyify

# If you use Yarn
yarn add --dev envify terser uglifyify
```

To create a production build, make sure that you add these transforms (**the order matters**):

- The [envify](#) transform ensures the right build environment is set. Make it global (`g`).
- The [uglifyify](#) transform removes development imports. Make it global too (`g`).
- Finally, the resulting bundle is piped to [terser](#) for mangling ([read why](#)).

For example:

```
browserify ./index.js \
  -g [ envify --NODE_ENV production ] \
  -g uglifyify \
  | terser --compress --mangle > ./bundle.js
```

Remember that you only need to do this for production builds. You shouldn't apply these plugins in development because they will hide useful React warnings, and make the builds much slower.

Rollup

For the most efficient Rollup production build, install a few plugins:

```
# If you use npm
npm install --save-dev rollup-plugin-commonjs rollup-plugin-replace rollup-plugin-terser

# If you use Yarn
yarn add --dev rollup-plugin-commonjs rollup-plugin-replace rollup-plugin-terser
```

To create a production build, make sure that you add these plugins **(the order matters)**:

- The `replace` plugin ensures the right build environment is set.
- The `commonjs` plugin provides support for CommonJS in Rollup.
- The `terser` plugin compresses and mangles the final bundle.

```
plugins: [
  // ...
  require('rollup-plugin-replace')({
    'process.env.NODE_ENV': JSON.stringify('production')
  }),
  require('rollup-plugin-commonjs')(),
  require('rollup-plugin-terser')(),
  // ...
]
```

For a complete setup example [see this gist](#).

Remember that you only need to do this for production builds. You shouldn't apply the `terser` plugin or the `replace` plugin with `'production'` value in development because they will hide useful React warnings, and make the builds much slower.

webpack

Note:

If you're using Create React App, please follow [the instructions above](#). This section is only relevant if you configure webpack directly.

Webpack v4+ will minify your code by default in production mode.

```
const TerserPlugin = require('terser-webpack-plugin');

module.exports = {
  mode: 'production',
  optimization: {
    minimizer: [new TerserPlugin({ /* additional options here */ })],
  },
};
```

You can learn more about this in [webpack documentation](#).

Remember that you only need to do this for production builds. You shouldn't apply `TerserPlugin` in development because it will hide useful React warnings, and make the builds much slower.

Profiling Components with the DevTools Profiler

`react-dom` 16.5+ and `react-native` 0.57+ provide enhanced profiling capabilities in DEV mode with the React DevTools Profiler. An overview of the Profiler can be found in the blog post [“Introducing the React Profiler”](#). A video walkthrough of the profiler is also [available on YouTube](#).

If you haven't yet installed the React DevTools, you can find them here:

- [Chrome Browser Extension](#)
- [Firefox Browser Extension](#)
- [Standalone Node Package](#)

Note

A production profiling bundle of `react-dom` is also available as `react-dom/profiling`. Read more about how to use this bundle at fb.me/react-profiling

Note

Before React 17, we use the standard [User Timing API](#) to profile components with the chrome performance tab. For a more detailed walkthrough, check out [this article by Ben Schwarz](#).

Virtualize Long Lists

If your application renders long lists of data (hundreds or thousands of rows), we recommend using a technique known as “windowing”. This technique only renders a small subset of your rows at any given time, and can dramatically reduce the time it takes to re-render the components as well as the number of DOM nodes created.

[react-window](#) and [react-virtualized](#) are popular windowing libraries. They provide several reusable components for displaying lists, grids, and tabular data. You can also create your own windowing component, like [Twitter did](#), if you want something more tailored to your application's specific use case.

Avoid Reconciliation

React builds and maintains an internal representation of the rendered UI. It includes the React elements you return from your components. This representation lets React avoid creating DOM nodes and accessing existing ones beyond necessity, as that can be slower than operations on JavaScript objects. Sometimes it is referred to as a “virtual DOM”, but it works the same way on React Native.

When a component's props or state change, React decides whether an actual DOM update is necessary by comparing the newly returned element with the previously rendered one. When they are not equal, React will update the DOM.

Even though React only updates the changed DOM nodes, re-rendering still takes some time. In many cases it's not a problem, but if the slowdown is noticeable, you can speed all of this up by overriding the lifecycle function `shouldComponentUpdate`, which is triggered before the re-rendering process starts. The default implementation of this function returns `true`, leaving React to perform the update:

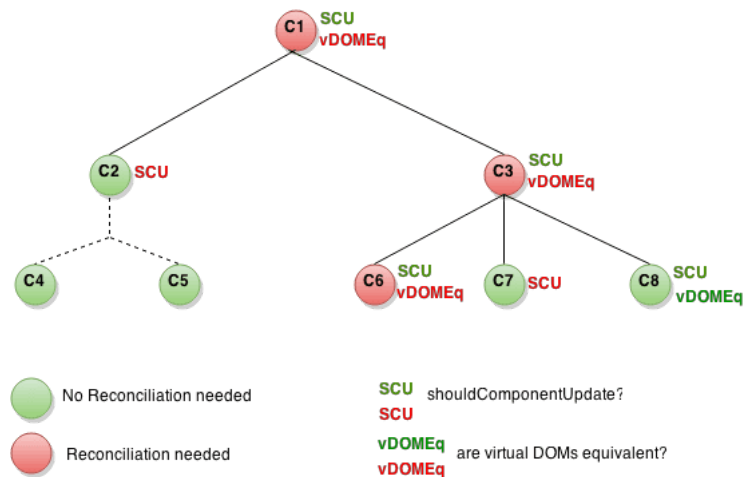
```
shouldComponentUpdate(nextProps, nextState) {  
  return true;  
}
```

If you know that in some situations your component doesn't need to update, you can return `false` from `shouldComponentUpdate` instead, to skip the whole rendering process, including calling `render()` on this component and below.

In most cases, instead of writing `shouldComponentUpdate()` by hand, you can inherit from `React.PureComponent`. It is equivalent to implementing `shouldComponentUpdate()` with a shallow comparison of current and previous props and state.

shouldComponentUpdate In Action

Here's a subtree of components. For each one, `SCU` indicates what `shouldComponentUpdate` returned, and `VDOMEq` indicates whether the rendered React elements were equivalent. Finally, the circle's color indicates whether the component had to be reconciled or not.



Since `shouldComponentUpdate` returned `false` for the subtree rooted at C2, React did not attempt to render C2, and thus didn't even have to invoke `shouldComponentUpdate` on C4 and C5.

For C1 and C3, `shouldComponentUpdate` returned `true`, so React had to go down to the leaves and check them. For C6 `shouldComponentUpdate` returned `true`, and since the rendered elements weren't equivalent React had to update the DOM.

The last interesting case is C8. React had to render this component, but since the React elements it returned were equal to the previously rendered ones, it didn't have to update the DOM.

Note that React only had to do DOM mutations for C6, which was inevitable. For C8, it bailed out by comparing the rendered React elements, and for C2's subtree and C7, it didn't even have to compare the elements as we bailed out on `shouldComponentUpdate`, and `render` was not called.

Examples

If the only way your component ever changes is when the `props.color` or the `state.count` variable changes, you could have `shouldComponentUpdate` check that:

```

class CounterButton extends React.Component {
  constructor(props) {
    super(props);
    this.state = { count: 1 };
  }

  shouldComponentUpdate(nextProps, nextState) {
    if (this.props.color !== nextProps.color) {
      return true;
    }
    if (this.state.count !== nextState.count) {
      return true;
    }
    return false;
  }

  render() {
    return (
      <button
        color={this.props.color}
        onClick={() => {
          this.setState((state) => ({ count: state.count + 1 }));
        }}
      >
        Count: {this.state.count}
      </button>
    );
  }
}

```

In this code, `shouldComponentUpdate` is just checking if there is any change in `props.color` or `state.count`. If those values don't change, the component doesn't update. If your component got more complex, you could use a similar pattern of doing a "shallow comparison" between all the fields of `props` and `state` to determine if the component should update. This pattern is

common enough that React provides a helper to use this logic - just inherit from `React.PureComponent`. So this code is a simpler way to achieve the same thing:

```
class CounterButton extends React.PureComponent {
  constructor(props) {
    super(props);
    this.state = { count: 1 };
  }

  render() {
    return (
      <button
        color={this.props.color}
        onClick={() => {
          this.setState((state) => ({ count: state.count + 1 }));
        }}
      >
        Count: {this.state.count}
      </button>
    );
  }
}
```

Most of the time, you can use `React.PureComponent` instead of writing your own `shouldComponentUpdate`. It only does a shallow comparison, so you can't use it if the props or state may have been mutated in a way that a shallow comparison would miss.

This can be a problem with more complex data structures. For example, let's say you want a `ListOfWords` component to render a comma-separated list of words, with a parent `WordAdder` component that lets you click a button to add a word to the list. This code does *not* work correctly:

```
class ListOfWords extends React.PureComponent {
  render() {
    return <div>{this.props.words.join(', ')}</div>;
  }
}

class WordAdder extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      words: ['marklar']
    };
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    // This section is bad style and causes a bug
    const words = this.state.words;
    words.push('marklar');
    this.setState({ words: words });
  }

  render() {
    return (
      <div>
        <button onClick={this.handleClick} />
        <ListOfWords words={this.state.words} />
      </div>
    );
  }
}
```

The problem is that `PureComponent` will do a simple comparison between the old and new values of `this.props.words`. Since this code mutates the `words` array in the `handleClick` method of `WordAdder`, the old and new values of `this.props.words` will compare as equal, even though the actual words in the array have changed. The `ListOfWords` will thus not update even though it has new words that should be rendered.

The Power Of Not Mutating Data

The simplest way to avoid this problem is to avoid mutating values that you are using as props or state. For example, the `handleClick` method above could be rewritten using `concat` as:

```
handleClick() {
  this.setState(state => ({
    words: state.words.concat(['marklar'])
  }));
}
```

ES6 supports a [spread syntax](#) for arrays which can make this easier. If you're using Create React App, this syntax is available by default.

```
handleClick() {
  this.setState(state => ({
    words: [...state.words, 'marklar'],
  }));
};
```

You can also rewrite code that mutates objects to avoid mutation, in a similar way. For example, let's say we have an object named `colormap` and we want to write a function that changes `colormap.right` to be `'blue'`. We could write:

```
function updateColorMap(colormap) {
  colormap.right = 'blue';
}
```

To write this without mutating the original object, we can use [Object.assign](#) method:

```
function updateColorMap(colormap) {
  return Object.assign({}, colormap, {right: 'blue'});
}
```

`updateColorMap` now returns a new object, rather than mutating the old one. `Object.assign` is in ES6 and requires a polyfill.

[Object spread syntax](#) makes it easier to update objects without mutation as well:

```
function updateColorMap(colormap) {
  return {...colormap, right: 'blue'};
}
```

This feature was added to JavaScript in ES2018.

If you're using Create React App, both `Object.assign` and the object spread syntax are available by default.

When you deal with deeply nested objects, updating them in an immutable way can feel convoluted. If you run into this problem, check out [Immer](#) or [immutability-helper](#). These libraries let you write highly readable code without losing the benefits of immutability.

11. Portals

Portals provide a first-class way to render children into a DOM node that exists outside the DOM hierarchy of the parent component.

```
ReactDOM.createPortal(child, container)
```

The first argument (`child`) is any [renderable React child](#), such as an element, string, or fragment. The second argument (`container`) is a DOM element.

Usage

Normally, when you return an element from a component's render method, it's mounted into the DOM as a child of the nearest parent node:

```
render() {
  // React mounts a new div and renders the children into it
  return (
    <div>
      {this.props.children}
    </div>
  );
}
```

```

    </div>
  );
}

```

However, sometimes it's useful to insert a child into a different location in the DOM:

```

render() {
  // React does *not* create a new div. It renders the children into `domNode`.
  // `domNode` is any valid DOM node, regardless of its location in the DOM.
  return ReactDOM.createPortal(
    this.props.children,
    domNode
  );
}

```

A typical use case for portals is when a parent component has an `overflow: hidden` or `z-index` style, but you need the child to visually “break out” of its container. For example, dialogs, hovercards, and tooltips.

Note:

When working with portals, remember that managing keyboard focus becomes very important.

For modal dialogs, ensure that everyone can interact with them by following the WAI-ARIA Modal Authoring Practices.

Event Bubbling Through Portals

Even though a portal can be anywhere in the DOM tree, it behaves like a normal React child in every other way. Features like context work exactly the same regardless of whether the child is a portal, as the portal still exists in the *React tree* regardless of position in the *DOM tree*.

This includes event bubbling. An event fired from inside a portal will propagate to ancestors in the containing *React tree*, even if those elements are not ancestors in the *DOM tree*. Assuming the following HTML structure:

```

<html>
  <body>
    <div id="app-root"></div>
    <div id="modal-root"></div>
  </body>
</html>;

```

A `Parent` component in `#app-root` would be able to catch an uncaught, bubbling event from the sibling node `#modal-root`.

```

// These two containers are siblings in the DOM
const appRoot = document.getElementById('app-root');
const modalRoot = document.getElementById('modal-root');

class Modal extends React.Component {
  constructor(props) {
    super(props);
    this.el = document.createElement('div');
  }

  componentDidMount() {
    // The portal element is inserted in the DOM tree after
    // the Modal's children are mounted, meaning that children
    // will be mounted on a detached DOM node. If a child
    // component requires to be attached to the DOM tree
    // immediately when mounted, for example to measure a
    // DOM node, or uses 'autoFocus' in a descendant, add
    // state to Modal and only render the children when Modal
    // is inserted in the DOM tree.
    modalRoot.appendChild(this.el);
  }

  componentWillUnmount() {
    modalRoot.removeChild(this.el);
  }

  render() {
    return ReactDOM.createPortal(this.props.children, this.el);
  }
}

```



```

    }
  }

  class Parent extends React.Component {
    constructor(props) {
      super(props);
      this.state = { clicks: 0 };
      this.handleClick = this.handleClick.bind(this);
    }

    handleClick() {
      // This will fire when the button in Child is clicked,
      // updating Parent's state, even though button
      // is not direct descendant in the DOM.
      this.setState((state) => ({ clicks: state.clicks + 1 }));
    }

    render() {
      return (
        <div onClick={this.handleClick}>
          <p>Number of clicks: {this.state.clicks}</p>
          <p>
            Open up the browser DevTools to observe that the button is not a child of the div with the onClick handler.
          </p>
          <Modal>
            <Child />
          </Modal>
        </div>
      );
    }
  }

  function Child() {
    // The click event on this button will bubble up to parent,
    // because there is no 'onClick' attribute defined
    return (
      <div className="modal">
        <button>Click</button>
      </div>
    );
  }

  const root = ReactDOM.createRoot(appRoot);
  root.render(<Parent />);

```

Catching an event bubbling up from a portal in a parent component allows the development of more flexible abstractions that are not inherently reliant on portals. For example, if you render a `<Modal />` component, the parent can capture its events regardless of whether it's implemented using portals.

12. Profiler API

The **Profiler** measures how often a React application renders and what the “cost” of rendering is. Its purpose is to help identify parts of an application that are slow and may benefit from optimizations such as memoization.

Note:

Profiling adds some additional overhead, so it is **disabled in the production build**.

To opt into production profiling, React provides a special production build with profiling enabled.

Read more about how to use this build at fb.me/react-profiling

Usage

A **Profiler** can be added anywhere in a React tree to measure the cost of rendering that part of the tree. It requires two props: an `id` (string) and an `onRender` callback (function) which React calls any time a component within the tree “commits” an update.

For example, to profile a **Navigation** component and its descendants:

```

render(
  <App>
    <Profiler id="Navigation" onRender={callback}>
      <Navigation {...props} />
    </Profiler>
    <Main {...props} />
  </App>
);

```

Multiple `Profiler` components can be used to measure different parts of an application:

```
render(  
  <App>  
    <Profiler id="Navigation" onRender={callback}>  
      <Navigation {...props} />  
    </Profiler>  
    <Profiler id="Main" onRender={callback}>  
      <Main {...props} />  
    </Profiler>  
  </App>  
);
```

`Profiler` components can also be nested to measure different components within the same subtree:

```
render(  
  <App>  
    <Profiler id="Panel" onRender={callback}>  
      <Panel {...props}>  
        <Profiler id="Content" onRender={callback}>  
          <Content {...props} />  
        </Profiler>  
        <Profiler id="PreviewPane" onRender={callback}>  
          <PreviewPane {...props} />  
        </Profiler>  
      </Panel>  
    </Profiler>  
  </App>  
);
```

Note

Although `Profiler` is a light-weight component, it should be used only when necessary; each use adds some CPU and memory overhead to an application.

`onRender` Callback

The `Profiler` requires an `onRender` function as a prop. React calls this function any time a component within the profiled tree "commits" an update. It receives parameters describing what was rendered and how long it took.

```
function onRenderCallback(  
  id, // the "id" prop of the Profiler tree that has just committed  
  phase, // either "mount" (if the tree just mounted) or "update" (if it re-rendered)  
  actualDuration, // time spent rendering the committed update  
  baseDuration, // estimated time to render the entire subtree without memoization  
  startTime, // when React began rendering this update  
  commitTime, // when React committed this update  
  interactions // the Set of interactions belonging to this update  
) {  
  // Aggregate or log render timings...  
}
```

Let's take a closer look at each of the props:

- `id: string` - The `id` prop of the `Profiler` tree that has just committed. This can be used to identify which part of the tree was committed if you are using multiple profilers.
- `phase: "mount" | "update"` - Identifies whether the tree has just been mounted for the first time or re-rendered due to a change in props, state, or hooks.
- `actualDuration: number` - Time spent rendering the `Profiler` and its descendants for the current update. This indicates how well the subtree makes use of memoization (e.g. [React.memo](#), [useMemo](#), [shouldComponentUpdate](#)). Ideally this value should decrease significantly after the initial mount as many of the descendants will only need to re-render if their specific props change.
- `baseDuration: number` - Duration of the most recent `render` time for each individual component within the `Profiler` tree. This value estimates a worst-case cost of rendering (e.g. the initial mount or a tree with no memoization).

- `startTime: number` - Timestamp when React began rendering the current update.
- `commitTime: number` - Timestamp when React committed the current update. This value is shared between all profilers in a commit, enabling them to be grouped if desirable.
- `interactions: Set` - Set of “interactions” that were being traced when the update was scheduled (e.g. when `render` or `setState` were called).

Note

Interactions can be used to identify the cause of an update, although the API for tracing them is still experimental.

Learn more about it at fb.me/react-interaction-tracing

13. React Without ES6

Normally you would define a React component as a plain JavaScript class:

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

If you don't use ES6 yet, you may use the `create-react-class` module instead:

```
var createReactClass = require('create-react-class');
var Greeting = createReactClass({
  render: function () {
    return <h1>Hello, {this.props.name}</h1>;
  }
});
```

The API of ES6 classes is similar to `createReactClass()` with a few exceptions.

Declaring Default Props

With functions and ES6 classes `defaultProps` is defined as a property on the component itself:

```
class Greeting extends React.Component {
  // ...
}

Greeting.defaultProps = {
  name: 'Mary'
};
```

With `createReactClass()`, you need to define `getDefaultProps()` as a function on the passed object:

```
var Greeting = createReactClass({
  getDefaultProps: function() {
    return {
      name: 'Mary'
    };
  },
  // ...
});
```

Setting the Initial State

In ES6 classes, you can define the initial state by assigning `this.state` in the constructor:

```
class Counter extends React.Component {
  constructor(props) {
    super(props);
    this.state = { count: props.initialCount };
  }
  // ...
}
```

With `createReactClass()`, you have to provide a separate `getInitialState` method that returns the initial state:

```
var Counter = createReactClass({
  getInitialState: function () {
    return { count: this.props.initialCount };
  }
  // ...
});
```

Autobinding

In React components declared as ES6 classes, methods follow the same semantics as regular ES6 classes. This means that they don't automatically bind `this` to the instance. You'll have to explicitly use `.bind(this)` in the constructor:

```
class SayHello extends React.Component {
  constructor(props) {
    super(props);
    this.state = { message: 'Hello!' };
    // This line is important!
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    alert(this.state.message);
  }

  render() {
    // Because `this.handleClick` is bound, we can use it as an event handler.
    return <button onClick={this.handleClick}>Say hello</button>;
  }
}
```

With `createReactClass()`, this is not necessary because it binds all methods:

```
var SayHello = createReactClass({
  getInitialState: function () {
    return { message: 'Hello!' };
  },

  handleClick: function () {
    alert(this.state.message);
  },

  render: function () {
    return <button onClick={this.handleClick}>Say hello</button>;
  }
});
```

This means writing ES6 classes comes with a little more boilerplate code for event handlers, but the upside is slightly better performance in large applications.

If the boilerplate code is too unattractive to you, you may use [ES2022 Class Properties](#) syntax:

```
class SayHello extends React.Component {
  constructor(props) {
    super(props);
    this.state = { message: 'Hello!' };
  }

  // Using an arrow here binds the method:
  handleClick = () => {
    alert(this.state.message);
  };
}
```

```
render() {
  return <button onClick={this.handleClick}>Say hello</button>;
}
}
```

You also have a few other options:

- Bind methods in the constructor.
- Use arrow functions, e.g. `onClick={() => this.handleClick(e)}`.
- Keep using `createReactClass`.

Mixins

Note:

ES6 launched without any mixin support. Therefore, there is no support for mixins when you use React with ES6 classes.

We also found numerous issues in codebases using mixins, and don't recommend using them in the new code.

This section exists only for the reference.

Sometimes very different components may share some common functionality. These are sometimes called cross-cutting concerns. `createReactClass` lets you use a legacy `mixins` system for that.

One common use case is a component wanting to update itself on a time interval. It's easy to use `setInterval()`, but it's important to cancel your interval when you don't need it anymore to save memory. React provides lifecycle methods that let you know when a component is about to be created or destroyed. Let's create a simple mixin that uses these methods to provide an easy `setInterval()` function that will automatically get cleaned up when your component is destroyed.

```
var SetIntervalMixin = {
  componentWillMount: function () {
    this.intervals = [];
  },
  setInterval: function () {
    this.intervals.push(setInterval.apply(null, arguments));
  },
  componentWillUnmount: function () {
    this.intervals.forEach(clearInterval);
  }
};

var createReactClass = require('create-react-class');

var TickTock = createReactClass({
  mixins: [SetIntervalMixin], // Use the mixin
  getInitialState: function () {
    return { seconds: 0 };
  },
  componentDidMount: function () {
    this.setInterval(this.tick, 1000); // Call a method on the mixin
  },
  tick: function () {
    this.setState({ seconds: this.state.seconds + 1 });
  },
  render: function () {
    return <p>React has been running for {this.state.seconds} seconds.</p>;
  }
});

const root = ReactDOM.createRoot(document.getElementById('example'));
root.render(<TickTock />);
```

If a component is using multiple mixins and several mixins define the same lifecycle method (i.e. several mixins want to do some cleanup when the component is destroyed), all of the lifecycle methods are guaranteed to be called. Methods defined on mixins run in the order mixins were listed, followed by a method call on the component.

14. React Without JSX

JSX is not a requirement for using React. Using React without JSX is especially convenient when you don't want to set up compilation in your build environment.

Each JSX element is just syntactic sugar for calling `React.createElement(component, props, ...children)`. So, anything you can do with JSX can also be done with just plain JavaScript.

For example, this code written with JSX:

```
class Hello extends React.Component {
  render() {
    return <div>Hello {this.props.toWhat}</div>;
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Hello toWhat="World" />);
```

can be compiled to this code that does not use JSX:

```
class Hello extends React.Component {
  render() {
    return React.createElement('div', null, `Hello ${this.props.toWhat}`);
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(React.createElement(Hello, { toWhat: 'World' }, null));
```

If you're curious to see more examples of how JSX is converted to JavaScript, you can try out [the online Babel compiler](#).

The component can either be provided as a string, as a subclass of `React.Component`, or a plain function.

If you get tired of typing `React.createElement` so much, one common pattern is to assign a shorthand:

```
const e = React.createElement;

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(e('div', null, 'Hello World'));
```

If you use this shorthand form for `React.createElement`, it can be almost as convenient to use React without JSX.

Alternatively, you can refer to community projects such as [react-hyperscript](#) and [hyperscript-helpers](#) which offer a terser syntax.

15. Reconciliation

React provides a declarative API so that you don't have to worry about exactly what changes on every update. This makes writing applications a lot easier, but it might not be obvious how this is implemented within React. This article explains the choices we made in React's "diffing" algorithm so that component updates are predictable while being fast enough for high-performance apps.

Motivation

When you use React, at a single point in time you can think of the `render()` function as creating a tree of React elements. On the next state or props update, that `render()` function will return a different tree of React elements. React then needs to figure out how to efficiently update the UI to match the most recent tree.

There are some generic solutions to this algorithmic problem of generating the minimum number of operations to transform one tree into another. However, the [state of the art algorithms](#) have a complexity in the order of $O(n^3)$ where n is the number of elements in the tree.

If we used this in React, displaying 1000 elements would require in the order of one billion comparisons. This is far too expensive. Instead, React implements a heuristic $O(n)$ algorithm based on two assumptions:

1. Two elements of different types will produce different trees.
2. The developer can hint at which child elements may be stable across different renders with a `key` prop.

In practice, these assumptions are valid for almost all practical use cases.

The Diffing Algorithm

When diffing two trees, React first compares the two root elements. The behavior is different depending on the types of the root elements.

Elements Of Different Types

Whenever the root elements have different types, React will tear down the old tree and build the new tree from scratch. Going from `<a>` to ``, or from `<Article>` to `<Comment>`, or from `<Button>` to `<div>` - any of those will lead to a full rebuild.

When tearing down a tree, old DOM nodes are destroyed. Component instances receive `componentWillUnmount()`. When building up a new tree, new DOM nodes are inserted into the DOM. Component instances receive `UNSAFE_componentWillMount()` and then `componentDidMount()`. Any state associated with the old tree is lost.

Any components below the root will also get unmounted and have their state destroyed. For example, when diffing:

```
<div><Counter /></div>
<span><Counter /></span>
```

This will destroy the old `Counter` and remount a new one.

Note:

This method is considered legacy and you should avoid it in new code:

- `UNSAFE_componentWillMount()`

DOM Elements Of The Same Type

When comparing two React DOM elements of the same type, React looks at the attributes of both, keeps the same underlying DOM node, and only updates the changed attributes. For example:

```
<div className="before" title="stuff" />
<div className="after" title="stuff" />
```

By comparing these two elements, React knows to only modify the `className` on the underlying DOM node.

When updating `style`, React also knows to update only the properties that changed. For example:

```
<div style={{color: 'red', fontWeight: 'bold'}} />
<div style={{color: 'green', fontWeight: 'bold'}} />
```

When converting between these two elements, React knows to only modify the `color` style, not the `fontWeight`.

After handling the DOM node, React then recurses on the children.

Component Elements Of The Same Type

When a component updates, the instance stays the same, so that state is maintained across renders. React updates the props of the underlying component instance to match the new element, and

calls `UNSAFE_componentWillReceiveProps()`, `UNSAFE_componentWillUpdate()` and `componentDidUpdate()` on the underlying instance.

Next, the `render()` method is called and the diff algorithm recurses on the previous result and the new result.

Note:

These methods are considered legacy and you should avoid them in new code:

- `UNSAFE_componentWillUpdate()`

- `UNSAFE_componentWillReceiveProps()`

Recurring On Children

By default, when recursing on the children of a DOM node, React just iterates over both lists of children at the same time and generates a mutation whenever there's a difference.

For example, when adding an element at the end of the children, converting between these two trees works well:

```
<ul>
  <li>first</li>
  <li>second</li>
</ul>
<ul>
  <li>first</li>
  <li>second</li>
  <li>third</li>
</ul>
```

React will match the two `first` trees, match the two `second` trees, and then insert the `third` tree.

If you implement it naively, inserting an element at the beginning has worse performance. For example, converting between these two trees works poorly:

```
<ul>
  <li>Duke</li>
  <li>Villanova</li>
</ul>
<ul>
  <li>Connecticut</li>
  <li>Duke</li>
  <li>Villanova</li>
</ul>
```

React will mutate every child instead of realizing it can keep the `Duke` and `Villanova` subtrees intact. This inefficiency can be a problem.

Keys

In order to solve this issue, React supports a `key` attribute. When children have keys, React uses the key to match children in the original tree with children in the subsequent tree. For example, adding a `key` to our inefficient example above can make the tree conversion efficient:

```
<ul>
  <li key="2015">Duke</li>
  <li key="2016">Villanova</li>
</ul>
<ul>
  <li key="2014">Connecticut</li>
  <li key="2015">Duke</li>
  <li key="2016">Villanova</li>
</ul>
```

Now React knows that the element with key `'2014'` is the new one, and the elements with the keys `'2015'` and `'2016'` have just moved.

In practice, finding a key is usually not hard. The element you are going to display may already have a unique ID, so the key can just come from your data:

```
<li key={item.id}>{item.name}</li>
```

When that's not the case, you can add a new ID property to your model or hash some parts of the content to generate a key. The key only has to be unique among its siblings, not globally unique.

As a last resort, you can pass an item's index in the array as a key. This can work well if the items are never reordered, but reorders will be slow.

Reorders can also cause issues with component state when indexes are used as keys. Component instances are updated and reused based on their key. If the key is an index, moving an item changes it. As a result, component state for things like uncontrolled inputs can get mixed up and updated in unexpected ways.

Here is [an example of the issues that can be caused by using indexes as keys](#) on CodePen, and here is [an updated version of the same example showing how not using indexes as keys will fix these reordering, sorting, and prepending issues](#).

Tradeoffs

It is important to remember that the reconciliation algorithm is an implementation detail. React could rerender the whole app on every action; the end result would be the same. Just to be clear, rerender in this context means calling `render` for all components, it doesn't mean React will unmount and remount them. It will only apply the differences following the rules stated in the previous sections.

We are regularly refining the heuristics in order to make common use cases faster. In the current implementation, you can express the fact that a subtree has been moved amongst its siblings, but you cannot tell that it has moved somewhere else. The algorithm will rerender that full subtree.

Because React relies on heuristics, if the assumptions behind them are not met, performance will suffer.

1. The algorithm will not try to match subtrees of different component types. If you see yourself alternating between two component types with very similar output, you may want to make it the same type. In practice, we haven't found this to be an issue.
2. Keys should be stable, predictable, and unique. Unstable keys (like those produced by `Math.random()`) will cause many component instances and DOM nodes to be unnecessarily recreated, which can cause performance degradation and lost state in child components.

16. Refs and the DOM

Refs provide a way to access DOM nodes or React elements created in the render method.

In the typical React dataflow, [props](#) are the only way that parent components interact with their children. To modify a child, you re-render it with new props. However, there are a few cases where you need to imperatively modify a child outside of the typical dataflow. The child to be modified could be an instance of a React component, or it could be a DOM element. For both of these cases, React provides an escape hatch.

When to Use Refs

There are a few good use cases for refs:

- Managing focus, text selection, or media playback.
- Triggering imperative animations.
- Integrating with third-party DOM libraries.

Avoid using refs for anything that can be done declaratively.

For example, instead of exposing `open()` and `close()` methods on a `Dialog` component, pass an `isOpen` prop to it.

Don't Overuse Refs

Your first inclination may be to use refs to "make things happen" in your app. If this is the case, take a moment and think more critically about where state should be owned in the component hierarchy. Often, it becomes clear that the proper place to "own" that state is at a higher level in the hierarchy. See the [Lifting State Up](#) guide for examples of this.

Note

The examples below have been updated to use the `React.createRef()` API introduced in React 16.3. If you are using an earlier release of React, we recommend using [callback refs](#) instead.

Creating Refs

Refs are created using `React.createRef()` and attached to React elements via the `ref` attribute. Refs are commonly assigned to an instance property when a component is constructed so they can be referenced throughout the component.

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.myRef = React.createRef();
  }
  render() {
    return <div ref={this.myRef} />;
  }
}
```

Accessing Refs

When a ref is passed to an element in `render`, a reference to the node becomes accessible at the `current` attribute of the ref.

```
const node = this.myRef.current;
```

The value of the ref differs depending on the type of the node:

- When the `ref` attribute is used on an HTML element, the `ref` created in the constructor with `React.createRef()` receives the underlying DOM element as its `current` property.
- When the `ref` attribute is used on a custom class component, the `ref` object receives the mounted instance of the component as its `current`.
- **You may not use the `ref` attribute on function components** because they don't have instances.

The examples below demonstrate the differences.

Adding a Ref to a DOM Element

This code uses a `ref` to store a reference to a DOM node:

```
class CustomTextInput extends React.Component {
  constructor(props) {
    super(props);
    // create a ref to store the textInput DOM element
    this.textInput = React.createRef();
    this.focusTextInput = this.focusTextInput.bind(this);
  }

  focusTextInput() {
    // Explicitly focus the text input using the raw DOM API
    // Note: we're accessing "current" to get the DOM node
    this.textInput.current.focus();
  }

  render() {
    // tell React that we want to associate the <input> ref
    // with the `textInput` that we created in the constructor
    return (
      <div>
        {' '}
        <input type="text" ref={this.textInput} />{' '}
        <input type="button" value="Focus the text input" onClick={this.focusTextInput} />{' '}
      </div>
    );
  }
}
```

React will assign the `current` property with the DOM element when the component mounts, and assign it back to `null` when it unmounts. `ref` updates happen before `componentDidMount` or `componentDidUpdate` lifecycle methods.

Adding a Ref to a Class Component

If we wanted to wrap the `CustomTextInput` above to simulate it being clicked immediately after mounting, we could use a ref to get access to the custom input and call its `focusTextInput` method manually:

```
class AutoFocusTextInput extends React.Component {
  constructor(props) {
    super(props);
    this.textInput = React.createRef();
  }
```

```

    }

    componentDidMount() {
      this.textInput.current.focusTextInput();
    }

    render() {
      return <CustomTextInput ref={this.textInput} />;
    }
  }
}

```

Note that this only works if `CustomTextInput` is declared as a class:

```

class CustomTextInput extends React.Component {
  // ...
}

```

Refs and Function Components

By default, **you may not use the `ref` attribute on function components** because they don't have instances:

```

function MyFunctionComponent() {
  return <input />;
}

class Parent extends React.Component {
  constructor(props) {
    super(props);
    this.textInput = React.createRef();
  }
  render() {
    // This will *not* work!
    return <MyFunctionComponent ref={this.textInput} />;
  }
}

```

If you want to allow people to take a `ref` to your function component, you can use `forwardRef` (possibly in conjunction with `useImperativeHandle`), or you can convert the component to a class.

You can, however, **use the `ref` attribute inside a function component** as long as you refer to a DOM element or a class component:

```

function CustomTextInput(props) {
  // textInput must be declared here so the ref can refer to it
  const textInput = useRef(null);
  function handleClick() {
    textInput.current.focus();
  }

  return (
    <div>
      <input type="text" ref={textInput} /> <input type="button" value="Focus the text input" onClick={handleClick} />
    </div>
  );
}

```

Exposing DOM Refs to Parent Components

In rare cases, you might want to have access to a child's DOM node from a parent component. This is generally not recommended because it breaks component encapsulation, but it can occasionally be useful for triggering focus or measuring the size or position of a child DOM node.

While you could [add a ref to the child component](#), this is not an ideal solution, as you would only get a component instance rather than a DOM node. Additionally, this wouldn't work with function components.

If you use React 16.3 or higher, we recommend to use [ref forwarding](#) for these cases. **Ref forwarding lets components opt into exposing any child component's ref as their own.** You can find a detailed example of how to expose a child's DOM node to a parent component [in the ref forwarding documentation](#).

If you use React 16.2 or lower, or if you need more flexibility than provided by ref forwarding, you can use [this alternative approach](#) and explicitly pass a ref as a differently named prop.

When possible, we advise against exposing DOM nodes, but it can be a useful escape hatch. Note that this approach requires you to add some code to the child component. If you have absolutely no control over the child component implementation, your last option is to use `findDOMNode()`, but it is discouraged and deprecated in `StrictMode`.

Callback Refs

React also supports another way to set refs called “callback refs”, which gives more fine-grain control over when refs are set and unset.

Instead of passing a `ref` attribute created by `createRef()`, you pass a function. The function receives the React component instance or HTML DOM element as its argument, which can be stored and accessed elsewhere.

The example below implements a common pattern: using the `ref` callback to store a reference to a DOM node in an instance property.

```
class CustomTextInput extends React.Component {
  constructor(props) {
    super(props);

    this.textInput = null;
    this.setTextInputRef = (element) => {
      this.textInput = element;
    };
    this.focusTextInput = () => {
      // Focus the text input using the raw DOM API
      if (this.textInput) this.textInput.focus();
    };
  }

  componentDidMount() {
    // autofocus the input on mount
    this.focusTextInput();
  }

  render() {
    // Use the `ref` callback to store a reference to the text input DOM
    // element in an instance field (for example, this.textInput).
    return (
      <div>
        <input type="text" ref={this.setTextInputRef} />
        <input type="button" value="Focus the text input" onClick={this.focusTextInput} />
      </div>
    );
  }
}
```

React will call the `ref` callback with the DOM element when the component mounts, and call it with `null` when it unmounts. Refs are guaranteed to be up-to-date before `componentDidMount` or `componentDidUpdate` fires.

You can pass callback refs between components like you can with object refs that were created with `React.createRef()`.

```
function CustomTextInput(props) {
  return (
    <div>
      <input ref={props.inputRef} />
    </div>
  );
}

class Parent extends React.Component {
  render() {
    return <CustomTextInput inputRef={(el) => (this.inputElement = el)} />;
  }
}
```

In the example above, `Parent` passes its ref callback as an `inputRef` prop to the `CustomTextInput`, and the `CustomTextInput` passes the same function as a special `ref` attribute to the `<input>`. As a result, `this.inputElement` in `Parent` will be set to the DOM node corresponding to the `<input>` element in the `CustomTextInput`.

Legacy API: String Refs

If you worked with React before, you might be familiar with an older API where the `ref` attribute is a string, like `"textInput"`, and the DOM node is accessed as `this.refs.textInput`. We advise against it because string refs have [some issues](#), are considered legacy, and **are likely to be removed in one of the future releases**.

Note

If you're currently using `this.refs.textInput` to access refs, we recommend using either the [callback pattern](#) or the [createRef API](#) instead.

Caveats with callback refs

If the `ref` callback is defined as an inline function, it will get called twice during updates, first with `null` and then again with the DOM element. This is because a new instance of the function is created with each render, so React needs to clear the old ref and set up the new one. You can avoid this by defining the `ref` callback as a bound method on the class, but note that it shouldn't matter in most cases.

17. Render Props

The term **"render prop"** refers to a technique for sharing code between React components using a prop whose value is a function.

A component with a render prop takes a function that returns a React element and calls it instead of implementing its own render logic.

```
<DataProvider render={(data) => <h1>Hello {data.target}</h1>} />;
```

Libraries that use render props include [React Router](#), [Downshift](#) and [Formik](#).

In this document, we'll discuss why render props are useful, and how to write your own.

Use Render Props for Cross-Cutting Concerns

Components are the primary unit of code reuse in React, but it's not always obvious how to share the state or behavior that one component encapsulates to other components that need that same state.

For example, the following component tracks the mouse position in a web app:

```
class MouseTracker extends React.Component {
  constructor(props) {
    super(props);
    this.handleClick = this.handleClick.bind(this);
    this.state = { x: 0, y: 0 };
  }

  handleClick(event) {
    this.setState({
      x: event.clientX,
      y: event.clientY
    });
  }

  render() {
    return (
      <div style={{ height: '100vh' }} onClick={this.handleClick}>
        <h1>Move the mouse around!</h1>
        <p>
          The current mouse position is ({this.state.x}, {this.state.y})
        </p>
      </div>
    );
  }
}
```

As the cursor moves around the screen, the component displays its (x, y) coordinates in a `<p>`.

Now the question is: How can we reuse this behavior in another component? In other words, if another component needs to know about the cursor position, can we encapsulate that behavior so that we can easily share it with that component?

Since components are the basic unit of code reuse in React, let's try refactoring the code a bit to use a `<Mouse>` component that encapsulates the behavior we need to reuse elsewhere.

```
// The <Mouse> component encapsulates the behavior we need...
class Mouse extends React.Component {
  constructor(props) {
    super(props);
    this.handleMouseMove = this.handleMouseMove.bind(this);
    this.state = { x: 0, y: 0 };
  }

  handleMouseMove(event) {
    this.setState({
      x: event.clientX,
      y: event.clientY
    });
  }

  render() {
    return (
      <div style={{ height: '100vh' }} onMouseMove={this.handleMouseMove}>
        /* ...but how do we render something other than a <p>? */
        <p>
          The current mouse position is ({this.state.x}, {this.state.y})
        </p>
      </div>
    );
  }
}

class MouseTracker extends React.Component {
  render() {
    return (
      <>
        <h1>Move the mouse around!</h1>
        <Mouse />
      </>
    );
  }
}
```

Now the `<Mouse>` component encapsulates all behavior associated with listening for `mousemove` events and storing the (x, y) position of the cursor, but it's not yet truly reusable.

For example, let's say we have a `<Cat>` component that renders the image of a cat chasing the mouse around the screen. We might use a `<Cat mouse={{ x, y }}>` prop to tell the component the coordinates of the mouse so it knows where to position the image on the screen.

As a first pass, you might try rendering the `<Cat>` inside `<Mouse>`'s `render` method, like this:

```
class Cat extends React.Component {
  render() {
    const mouse = this.props.mouse;
    return ;
  }
}

class MouseWithCat extends React.Component {
  constructor(props) {
    super(props);
    this.handleMouseMove = this.handleMouseMove.bind(this);
    this.state = { x: 0, y: 0 };
  }

  handleMouseMove(event) {
    this.setState({
      x: event.clientX,
      y: event.clientY
    });
  }

  render() {
    return (
      <div style={{ height: '100vh' }} onMouseMove={this.handleMouseMove}>
        /*
         We could just swap out the <p> for a <Cat> here ... but then
         we would need to create a separate <MouseWithSomethingElse>
         component every time we need to use it, so <MouseWithCat>
         isn't really reusable yet.
        */
      <Cat mouse={this.state} />
      </div>
    );
  }
}
```

```

        <Cat mouse={this.state} />
      </div>
    );
  }
}

class MouseTracker extends React.Component {
  render() {
    return (
      <div>
        <h1>Move the mouse around!</h1>
        <MouseWithCat />
      </div>
    );
  }
}

```

This approach will work for our specific use case, but we haven't achieved the objective of truly encapsulating the behavior in a reusable way. Now, every time we want the mouse position for a different use case, we have to create a new component (i.e. essentially another `<MouseWithCat>`) that renders something specifically for that use case.

Here's where the `render` prop comes in: Instead of hard-coding a `<Cat>` inside a `<Mouse>` component, and effectively changing its rendered output, we can provide `<Mouse>` with a function prop that it uses to dynamically determine what to render—a `render` prop.

```

class Cat extends React.Component {
  render() {
    const mouse = this.props.mouse;
    return (
      
    );
  }
}

class Mouse extends React.Component {
  constructor(props) {
    super(props);
    this.handleClick = this.handleClick.bind(this);
    this.state = { x: 0, y: 0 };
  }

  handleClick(event) {
    this.setState({
      x: event.clientX,
      y: event.clientY
    });
  }

  render() {
    return (
      <div style={{ height: '100vh' }} onClick={this.handleClick}>
        /*
         * Instead of providing a static representation of what <Mouse> renders,
         * use the `render` prop to dynamically determine what to render.
         */
        {this.props.render(this.state)}
      </div>
    );
  }
}

class MouseTracker extends React.Component {
  render() {
    return (
      <div>
        <h1>Move the mouse around!</h1>
        <Mouse render={(mouse) => <Cat mouse={mouse} />} />
      </div>
    );
  }
}

```

Now, instead of effectively cloning the `<Mouse>` component and hard-coding something else in its `render` method to solve for a specific use case, we provide a `render` prop that `<Mouse>` can use to dynamically determine what it renders.

More concretely, a **render prop** is a function prop that a component uses to know what to render.

This technique makes the behavior that we need to share extremely portable. To get that behavior, render a `<Mouse>` with a `render` prop that tells it what to render with the current (x, y) of the cursor.

One interesting thing to note about render props is that you can implement most [higher-order components](#) (HOC) using a regular component with a render prop. For example, if you would prefer to have a `withMouse` HOC instead of a `<Mouse>` component, you could easily create one using a regular `<Mouse>` with a render prop:

```
// If you really want a HOC for some reason, you can easily
// create one using a regular component with a render prop!
function withMouse(Component) {
  return class extends React.Component {
    render() {
      return <Mouse render={mouse => <Component {...this.props} mouse={mouse} /> />;
    }
  };
}
```

So using a render prop makes it possible to use either pattern.

Using Props Other Than `render`

It's important to remember that just because the pattern is called “render props” you don't *have to use a prop named `render` to use this pattern*. In fact, any prop that is a function that a component uses to know what to render is technically a “render prop”.

Although the examples above use `render`, we could just as easily use the `children` prop!

```
<Mouse children={mouse => (<p>The mouse position is {mouse.x}, {mouse.y}</p>)} />;
```

And remember, the `children` prop doesn't actually need to be named in the list of “attributes” in your JSX element. Instead, you can put it directly *inside* the element!

```
<Mouse>{mouse => (<p>The mouse position is {mouse.x}, {mouse.y}</p>)}</Mouse>
```

You'll see this technique used in the [react-motion](#) API.

Since this technique is a little unusual, you'll probably want to explicitly state that `children` should be a function in your `propTypes` when designing an API like this.

```
Mouse.propTypes = {
  children: PropTypes.func.isRequired
};
```

Caveats

Be careful when using Render Props with `React.PureComponent`

Using a render prop can negate the advantage that comes from using `React.PureComponent` if you create the function inside a `render` method. This is because the shallow prop comparison will always return `false` for new props, and each `render` in this case will generate a new value for the render prop.

For example, continuing with our `<Mouse>` component from above, if `Mouse` were to extend `React.PureComponent` instead of `React.Component`, our example would look like this:

```
class Mouse extends React.PureComponent {
  // Same implementation as above...
}

class MouseTracker extends React.Component {
  render() {
    return (
      <div>
```



```

    <h1>Move the mouse around!</h1>

    /*
     * This is bad! The value of the `render` prop will
     * be different on each render.
     */
    <Mouse render={mouse} => <Cat mouse={mouse} /> />
  </div>
);
}
}

```

In this example, each time `<MouseTracker>` renders, it generates a new function as the value of the `<Mouse render>` prop, thus negating the effect of `<Mouse>` extending `React.PureComponent` in the first place!

To get around this problem, you can sometimes define the prop as an instance method, like so:

```

class MouseTracker extends React.Component {
  // Defined as an instance method, `this.renderTheCat` always
  // refers to *same* function when we use it in render
  renderTheCat(mouse) {
    return <Cat mouse={mouse} />;
  }

  render() {
    return (
      <div>
        <h1>Move the mouse around!</h1>
        <Mouse render={this.renderTheCat} />
      </div>
    );
  }
}

```

In cases where you cannot define the prop statically (e.g. because you need to close over the component's props and/or state) `<Mouse>` should extend `React.Component` instead.

18. Static Type Checking

Static type checkers like [Flow](#) and [TypeScript](#) identify certain types of problems before you even run your code. They can also improve developer workflow by adding features like auto-completion. For this reason, we recommend using Flow or TypeScript instead of `PropTypes` for larger code bases.

Flow

[Flow](#) is a static type checker for your JavaScript code. It is developed at Facebook and is often used with React. It lets you annotate the variables, functions, and React components with a special type syntax, and catch mistakes early. You can read an [introduction to Flow](#) to learn its basics.

To use Flow, you need to:

- Add Flow to your project as a dependency.
- Ensure that Flow syntax is stripped from the compiled code.
- Add type annotations and run Flow to check them.

We will explain these steps below in detail.

Adding Flow to a Project

First, navigate to your project directory in the terminal. You will need to run the following command:

If you use [Yarn](#), run:

```
yarn add --dev flow-bin
```

If you use [npm](#), run:

```
npm install --save-dev flow-bin
```

This command installs the latest version of Flow into your project.

Now, add `flow` to the `"scripts"` section of your `package.json` to be able to use this from the terminal:

```
{
  // ...
  "scripts": {
    "flow": "flow",    // ...
  },
  // ...
}
```

Finally, run one of the following commands:

If you use [Yarn](#), run:

```
yarn run flow init
```

If you use [npm](#), run:

```
npm run flow init
```

This command will create a Flow configuration file that you will need to commit.

Stripping Flow Syntax from the Compiled Code

Flow extends the JavaScript language with a special syntax for type annotations. However, browsers aren't aware of this syntax, so we need to make sure it doesn't end up in the compiled JavaScript bundle that is sent to the browser.

The exact way to do this depends on the tools you use to compile JavaScript.

Create React App

If your project was set up using [Create React App](#), congratulations! The Flow annotations are already being stripped by default so you don't need to do anything else in this step.

Babel

Note:

These instructions are *not* for Create React App users. Even though Create React App uses Babel under the hood, it is already configured to understand Flow. Only follow this step if you *don't* use Create React App.

If you manually configured Babel for your project, you will need to install a special preset for Flow.

If you use Yarn, run:

```
yarn add --dev @babel/preset-flow
```

If you use npm, run:

```
npm install --save-dev @babel/preset-flow
```

Then add the `flow` preset to your [Babel configuration](#). For example, if you configure Babel through `.babelrc` file, it could look like this:

```
{
  "presets": [
    "@babel/preset-flow",    "react"
  ]
}
```

This will let you use the Flow syntax in your code.

Note:

Flow does not require the `react` preset, but they are often used together. Flow itself understands JSX syntax out of the box.

Other Build Setups

If you don't use either Create React App or Babel, you can use [flow-remove-types](#) to strip the type annotations.

Running Flow

If you followed the instructions above, you should be able to run Flow for the first time.

```
yarn flow
```

If you use npm, run:

```
npm run flow
```

You should see a message like:

```
No errors!  
🌟 Done in 0.17s.
```

Adding Flow Type Annotations

By default, Flow only checks the files that include this annotation:

```
// @flow
```

Typically it is placed at the top of a file. Try adding it to some files in your project and run `yarn flow` or `npm run flow` to see if Flow already found any issues.

There is also [an option](#) to force Flow to check *all* files regardless of the annotation. This can be too noisy for existing projects, but is reasonable for a new project if you want to fully type it with Flow.

Now you're all set! We recommend to check out the following resources to learn more about Flow:

- [Flow Documentation: Type Annotations](#)
- [Flow Documentation: Editors](#)
- [Flow Documentation: React](#)
- [Linting in Flow](#)

TypeScript

[TypeScript](#) is a programming language developed by Microsoft. It is a typed superset of JavaScript, and includes its own compiler. Being a typed language, TypeScript can catch errors and bugs at build time, long before your app goes live. You can learn more about using TypeScript with React [here](#).

To use TypeScript, you need to:

- Add TypeScript as a dependency to your project
- Configure the TypeScript compiler options
- Use the right file extensions
- Add definitions for libraries you use

Let's go over these in detail.

Using TypeScript with Create React App

Create React App supports TypeScript out of the box.

To create a **new project** with TypeScript support, run:

```
npx create-react-app my-app --template typescript
```

You can also add it to an **existing Create React App project**, [as documented here](#).

Note:

If you use Create React App, you can **skip the rest of this page**. It describes the manual setup which doesn't apply to Create React App users.

Adding TypeScript to a Project

It all begins with running one command in your terminal.

If you use [Yarn](#), run:

```
yarn add --dev typescript
```

If you use [npm](#), run:

```
npm install --save-dev typescript
```

Congrats! You've installed the latest version of TypeScript into your project. Installing TypeScript gives us access to the `tsc` command. Before configuration, let's add `tsc` to the "scripts" section in our `package.json`:

```
{
  // ...
  "scripts": {
    "build": "tsc",    // ...
  },
  // ...
}
```

Configuring the TypeScript Compiler

The compiler is of no help to us until we tell it what to do. In TypeScript, these rules are defined in a special file called `tsconfig.json`. To generate this file:

If you use [Yarn](#), run:

```
yarn run tsc --init
```

If you use [npm](#), run:

```
npx tsc --init
```

Looking at the now generated `tsconfig.json`, you can see that there are many options you can use to configure the compiler. For a detailed description of all the options, check [here](#).

Of the many options, we'll look at `rootDir` and `outDir`. In its true fashion, the compiler will take in typescript files and generate javascript files. However we don't want to get confused with our source files and the generated output.

We'll address this in two steps:

- Firstly, let's arrange our project structure like this. We'll place all our source code in the `src` directory.

```
├─ package.json
├─ src
├─ └─ index.ts
└─ tsconfig.json
```

- Next, we'll tell the compiler where our source code is and where the output should go.

```
// tsconfig.json
```

```
{
  "compilerOptions": {
    // ...
    "rootDir": "src",    "outDir": "build"    // ...
  },
}
```

Great! Now when we run our build script the compiler will output the generated javascript to the `build` folder. The [TypeScript React Starter](#) provides a `tsconfig.json` with a good set of rules to get you started.

Generally, you don't want to keep the generated javascript in your source control, so be sure to add the build folder to your `.gitignore`.

File extensions

In React, you most likely write your components in a `.js` file. In TypeScript we have 2 file extensions:

`.ts` is the default file extension while `.tsx` is a special extension used for files which contain `JSX`.

Running TypeScript

If you followed the instructions above, you should be able to run TypeScript for the first time.

```
yarn build
```

If you use npm, run:

```
npm run build
```

If you see no output, it means that it completed successfully.

Type Definitions

To be able to show errors and hints from other packages, the compiler relies on declaration files. A declaration file provides all the type information about a library. This enables us to use javascript libraries like those on npm in our project.

There are two main ways to get declarations for a library:

Bundled - The library bundles its own declaration file. This is great for us, since all we need to do is install the library, and we can use it right away. To check if a library has bundled types, look for an `index.d.ts` file in the project. Some libraries will have it specified in their `package.json` under the `typings` or `types` field.

DefinitelyTyped - DefinitelyTyped is a huge repository of declarations for libraries that don't bundle a declaration file. The declarations are crowd-sourced and managed by Microsoft and open source contributors. React for example doesn't bundle its own declaration file. Instead we can get it from DefinitelyTyped. To do so enter this command in your terminal.

```
# yarn
yarn add --dev @types/react

# npm
npm i --save-dev @types/react
```

Local Declarations Sometimes the package that you want to use doesn't bundle declarations nor is it available on DefinitelyTyped. In that case, we can have a local declaration file. To do this, create a `declarations.d.ts` file in the root of your source directory. A simple declaration could look like this:

```
declare module 'querystring' {
  export function stringify(val: object): string
  export function parse(val: string): object
}
```

You are now ready to code! We recommend to check out the following resources to learn more about TypeScript:

- [TypeScript Documentation: Everyday Types](#)
- [TypeScript Documentation: Migrating from JavaScript](#)
- [TypeScript Documentation: React and Webpack](#)

ReScript

[ReScript](#) is a typed language that compiles to JavaScript. Some of its core features are guaranteed 100% type coverage, first-class JSX support and [dedicated React bindings](#) to allow integration in existing JS / TS React codebases.

You can find more infos on integrating ReScript in your existing JS / React codebase [here](#).

Kotlin

[Kotlin](#) is a statically typed language developed by JetBrains. Its target platforms include the JVM, Android, LLVM, and [JavaScript](#).

JetBrains develops and maintains several tools specifically for the React community: [React bindings](#) as well as [Create React Kotlin App](#). The latter helps you start building React apps with Kotlin with no build configuration.

Other Languages

Note there are other statically typed languages that compile to JavaScript and are thus React compatible. For example, [F#](#)/[Fable](#) with [elmish-react](#). Check out their respective sites for more information, and feel free to add more statically typed languages that work with React to this page!

19. Strict Mode

`StrictMode` is a tool for highlighting potential problems in an application. Like `Fragment`, `StrictMode` does not render any visible UI. It activates additional checks and warnings for its descendants.

Note:

Strict mode checks are run in development mode only; *they do not impact the production build.*

You can enable strict mode for any part of your application. For example:

```
import React from 'react';

function ExampleApplication() {
  return (
    <div>
      <Header />
      <React.StrictMode>
        <div>
          <ComponentOne /> <ComponentTwo />
        </div>
      </React.StrictMode>
      <Footer />
    </div>
  );
}
```

In the above example, strict mode checks will *not* be run against the `Header` and `Footer` components. However, `ComponentOne` and `ComponentTwo`, as well as all of their descendants, will have the checks.

`StrictMode` currently helps with:

- [Identifying components with unsafe lifecycles](#)
- [Warning about legacy string ref API usage](#)
- [Warning about deprecated findDOMNode usage](#)
- [Detecting unexpected side effects](#)
- [Detecting legacy context API](#)
- [Ensuring reusable state](#)

Additional functionality will be added with future releases of React.

Identifying unsafe lifecycles

As explained in [this blog post](#), certain legacy lifecycle methods are unsafe for use in async React applications. However, if your application uses third party libraries, it can be difficult to ensure that these lifecycles aren't being used. Fortunately, strict mode can help with this!

When strict mode is enabled, React compiles a list of all class components using the unsafe lifecycles, and logs a warning message with information about these components, like so:

```
►Warning: Unsafe lifecycle methods were found within a strict-mode tree:
  in div (created by ExampleApplication)
  in ExampleApplication

componentWillMount: Please update the following components to use componentDidMount instead: ThirdPartyComponent
Learn more about this warning here:
https://fb.me/react-strict-mode-warnings
```

Addressing the issues identified by strict mode *now* will make it easier for you to take advantage of concurrent rendering in future releases of React.

Warning about legacy string ref API usage

Previously, React provided two ways for managing refs: the legacy string ref API and the callback API. Although the string ref API was the more convenient of the two, it had [several downsides](#) and so our official recommendation was to [use the callback](#)

form instead.

React 16.3 added a third option that offers the convenience of a string ref without any of the downsides:

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);

    this.inputRef = React.createRef();
  }

  render() {
    return <input type="text" ref={this.inputRef} />;
  }

  componentDidMount() {
    this.inputRef.current.focus();
  }
}
```

Since object refs were largely added as a replacement for string refs, strict mode now warns about usage of string refs.

Note:

Callback refs will continue to be supported in addition to the new `createRef` API.

You don't need to replace callback refs in your components. They are slightly more flexible, so they will remain as an advanced feature.

[Learn more about the new `createRef` API here.](#)

Warning about deprecated findDOMNode usage

React used to support `findDOMNode` to search the tree for a DOM node given a class instance. Normally you don't need this because you can [attach a ref directly to a DOM node](#).

`findDOMNode` can also be used on class components but this was breaking abstraction levels by allowing a parent to demand that certain children were rendered. It creates a refactoring hazard where you can't change the implementation details of a component because a parent might be reaching into its DOM node. `findDOMNode` only returns the first child, but with the use of Fragments, it is possible for a component to render multiple DOM nodes. `findDOMNode` is a one time read API. It only gave you an answer when you asked for it. If a child component renders a different node, there is no way to handle this change. Therefore `findDOMNode` only worked if components always return a single DOM node that never changes.

You can instead make this explicit by passing a ref to your custom component and pass that along to the DOM using [ref forwarding](#).

You can also add a wrapper DOM node in your component and attach a ref directly to it.

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.wrapper = React.createRef();
  }
  render() {
    return <div ref={this.wrapper}>{this.props.children}</div>;
  }
}
```

Note:

In CSS, the `display: contents` attribute can be used if you don't want the node to be part of the layout.

Detecting unexpected side effects

Conceptually, React does work in two phases:

- The **render** phase determines what changes need to be made to e.g. the DOM. During this phase, React calls `render` and then compares the result to the previous render.
- The **commit** phase is when React applies any changes. (In the case of React DOM, this is when React inserts, updates, and removes DOM nodes.) React also calls lifecycles like `componentDidMount` and `componentDidUpdate` during this phase.

The commit phase is usually very fast, but rendering can be slow. For this reason, the upcoming concurrent mode (which is not enabled by default yet) breaks the rendering work into pieces, pausing and resuming the work to avoid blocking the browser. This means that React may invoke render phase lifecycles more than once before committing, or it may invoke them without committing at all (because of an error or a higher priority interruption).

Render phase lifecycles include the following class component methods:

- `constructor`
- `componentWillMount` (or `UNSAFE_componentWillMount`)
- `componentWillReceiveProps` (or `UNSAFE_componentWillReceiveProps`)
- `componentWillUpdate` (or `UNSAFE_componentWillUpdate`)
- `getDerivedStateFromProps`
- `shouldComponentUpdate`
- `render`
- `setState` updater functions (the first argument)

Because the above methods might be called more than once, it's important that they do not contain side-effects. Ignoring this rule can lead to a variety of problems, including memory leaks and invalid application state. Unfortunately, it can be difficult to detect these problems as they can often be non-deterministic.

Strict mode can't automatically detect side effects for you, but it can help you spot them by making them a little more deterministic. This is done by intentionally double-invoking the following functions:

- Class component `constructor`, `render`, and `shouldComponentUpdate` methods
- Class component static `getDerivedStateFromProps` method
- Function component bodies
- State updater functions (the first argument to `setState`)
- Functions passed to `useState`, `useMemo`, or `useReducer`

Note:

This only applies to development mode. *Lifecycles will not be double-invoked in production mode.*

For example, consider the following code:

```
class TopLevelRoute extends React.Component {
  constructor(props) {
    super(props);

    SharedApplicationState.recordEvent('ExampleComponent');
  }
}
```

At first glance, this code might not seem problematic. But if `SharedApplicationState.recordEvent` is not idempotent, then instantiating this component multiple times could lead to invalid application state. This sort of subtle bug might not manifest during development, or it might do so inconsistently and so be overlooked.

By intentionally double-invoking methods like the component constructor, strict mode makes patterns like this easier to spot.

Note:

In React 17, React automatically modifies the console methods like `console.log()` to silence the logs in the second call to lifecycle functions. However, it may cause undesired behavior in certain cases where a workaround can be used.

Starting from React 18, React does not suppress any logs. However, if you have React DevTools installed, the logs from the second call will appear slightly dimmed. React DevTools also offers a setting (off by default) to suppress them completely.

Detecting legacy context API

The legacy context API is error-prone, and will be removed in a future major version. It still works for all 16.x releases but will show this warning message in strict mode:

```
⛔ Warning: Legacy context API has been detected within a strict-mode tree:
  in div (at App.js:32)
  in App (at index.js:7)

Please update the following components: LegacyContextConsumer, LegacyContextProvider

Learn more about this warning here:
https://fb.me/react-strict-mode-warnings
```

Read the [new context API documentation](#) to help migrate to the new version.

Ensuring reusable state

In the future, we'd like to add a feature that allows React to add and remove sections of the UI while preserving state. For example, when a user tabs away from a screen and back, React should be able to immediately show the previous screen. To do this, React will support remounting trees using the same component state used before unmounting.

This feature will give React better performance out-of-the-box, but requires components to be resilient to effects being mounted and destroyed multiple times. Most effects will work without any changes, but some effects do not properly clean up subscriptions in the destroy callback, or implicitly assume they are only mounted or destroyed once.

To help surface these issues, React 18 introduces a new development-only check to Strict Mode. This new check will automatically unmount and remount every component, whenever a component mounts for the first time, restoring the previous state on the second mount.

To demonstrate the development behavior you'll see in Strict Mode with this feature, consider what happens when React mounts a new component. Without this change, when a component mounts, React creates the effects:

- React mounts the component.
 - Layout effects are created.
 - Effects are created.

With Strict Mode starting in React 18, whenever a component mounts in development, React will simulate immediately unmounting and remounting the component:

- React mounts the component.
 - Layout effects are created.
 - Effects are created.
- React simulates effects being destroyed on a mounted component.
 - Layout effects are destroyed.
 - Effects are destroyed.
- React simulates effects being re-created on a mounted component.
 - Layout effects are created
 - Effect setup code runs

On the second mount, React will restore the state from the first mount. This feature simulates user behavior such as a user tabbing away from a screen and back, ensuring that code will properly handle state restoration.

When the component unmounts, effects are destroyed as normal:

- React unmounts the component.

- `Layout effects are destroyed.`
- `Effects are destroyed.`

Unmounting and remounting includes:

- `componentDidMount`
- `componentWillUnmount`
- `useEffect`
- `useLayoutEffect`
- `useInsertionEffect`

Note:

This only applies to development mode, *production behavior is unchanged*.

20. Typechecking With PropTypes

Note:

`React.PropTypes` has moved into a different package since React v15.5. Please use [the `prop-types` library](#) instead.

We provide [a codemod script](#) to automate the conversion.

As your app grows, you can catch a lot of bugs with typechecking. For some applications, you can use JavaScript extensions like [Flow](#) or [TypeScript](#) to typecheck your whole application. But even if you don't use those, React has some built-in typechecking abilities. To run typechecking on the props for a component, you can assign the special `propTypes` property:

```
import PropTypes from 'prop-types';

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

Greeting.propTypes = {
  name: PropTypes.string
};
```

In this example, we are using a class component, but the same functionality could also be applied to function components, or components created by [React.memo](#) or [React.forwardRef](#).

`PropTypes` exports a range of validators that can be used to make sure the data you receive is valid. In this example, we're using `PropTypes.string`. When an invalid value is provided for a prop, a warning will be shown in the JavaScript console. For performance reasons, `propTypes` is only checked in development mode.

PropTypes

Here is an example documenting the different validators provided:

```
import PropTypes from 'prop-types';

MyComponent.propTypes = {
  // You can declare that a prop is a specific JS type. By default, these
  // are all optional.
  optionalArray: PropTypes.array,
  optionalBool: PropTypes.bool,
  optionalFunc: PropTypes.func,
  optionalNumber: PropTypes.number,
  optionalObject: PropTypes.object,
  optionalString: PropTypes.string,
  optionalSymbol: PropTypes.symbol,
```

```

// Anything that can be rendered: numbers, strings, elements or an array
// (or fragment) containing these types.
optionalNode: PropTypes.node,

// A React element.
optionalElement: PropTypes.element,

// A React element type (ie. MyComponent).
optionalElementType: PropTypes.elementType,

// You can also declare that a prop is an instance of a class. This uses
// JS's instanceof operator.
optionalMessage: PropTypes.instanceOf(Message),

// You can ensure that your prop is limited to specific values by treating
// it as an enum.
optionalEnum: PropTypes.oneOf(['News', 'Photos']),

// An object that could be one of many types
optionalUnion: PropTypes.oneOfType([
  PropTypes.string, PropTypes.number, PropTypes.instanceOf(Message)
]),

// An array of a certain type
optionalArrayOf: PropTypes.arrayOf(PropTypes.number),

// An object with property values of a certain type
optionalObjectOf: PropTypes.objectOf(PropTypes.number),

// An object taking on a particular shape
optionalObjectWithShape: PropTypes.shape({
  color: PropTypes.string,
  fontSize: PropTypes.number
}),

// An object with warnings on extra properties
optionalObjectWithStrictShape: PropTypes.exact({
  name: PropTypes.string,
  quantity: PropTypes.number
}),

// You can chain any of the above with `isRequired` to make sure a warning
// is shown if the prop isn't provided.
requiredFunc: PropTypes.func.isRequired,

// A required value of any data type
requiredAny: PropTypes.any.isRequired,

// You can also specify a custom validator. It should return an Error
// object if the validation fails. Don't `console.warn` or throw, as this
// won't work inside `oneOfType`.
customProp: function (props, propName, componentName) {
  if (!/matchme/.test(props[propName])) {
    return new Error('Invalid prop `' + propName + '` supplied to'
      + ' `' + componentName + '`. Validation failed.');
```

```

  }
},

// You can also supply a custom validator to `arrayOf` and `objectOf`.
// It should return an Error object if the validation fails. The validator
// will be called for each key in the array or object. The first two
// arguments of the validator are the array or object itself, and the
// current item's key.
customArrayProp: PropTypes.arrayOf(
  function (propValue, key, componentName, location, propFullName) {
    if (!/matchme/.test(propValue[key])) {
      return new Error(
        'Invalid prop `' + propFullName + '` supplied to' + ' `'
        + componentName + '`. Validation failed.');
```

Requiring Single Child

With `PropTypes.element` you can specify that only a single child can be passed to a component as children.

```

import PropTypes from 'prop-types';

class MyComponent extends React.Component {
  render() {
    // This must be exactly one element or it will warn.
```

```

    const children = this.props.children;
    return <div>{children}</div>;
  }
}

MyComponent.propTypes = {
  children: PropTypes.element.isRequired
};

```

Default Prop Values

You can define default values for your `props` by assigning to the special `defaultProps` property:

```

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

// Specifies the default values for props:
Greeting.defaultProps = {
  name: 'Stranger'
};

// Renders "Hello, Stranger":
const root = ReactDOM.createRoot(document.getElementById('example'));
root.render(<Greeting />);

```

Since ES2022 you can also declare `defaultProps` as static property within a React component class. For more information, see the [class public static fields](#). This modern syntax will require a compilation step to work within older browsers.

```

class Greeting extends React.Component {
  static defaultProps = {
    name: 'stranger'
  };

  render() {
    return <div>Hello, {this.props.name}</div>;
  }
}

```

The `defaultProps` will be used to ensure that `this.props.name` will have a value if it was not specified by the parent component. The `propTypes` typechecking happens after `defaultProps` are resolved, so typechecking will also apply to the `defaultProps`.

Function Components

If you are using function components in your regular development, you may want to make some small changes to allow `PropTypes` to be properly applied.

Let's say you have a component like this:

```

export default function HelloWorldComponent({ name }) {
  return <div>Hello, {name}</div>;
}

```

To add `PropTypes`, you may want to declare the component in a separate function before exporting, like this:

```

function HelloWorldComponent({ name }) {
  return <div>Hello, {name}</div>;
}

export default HelloWorldComponent;

```

Then, you can add `PropTypes` directly to the `HelloWorldComponent`:

```

import PropTypes from 'prop-types';

```

```
function HelloWorldComponent({ name }) {
  return <div>Hello, {name}</div>;
}

HelloWorldComponent.propTypes = {
  name: PropTypes.string
};

export default HelloWorldComponent;
```

21. Uncontrolled Components

In most cases, we recommend using [controlled components](#) to implement forms. In a controlled component, form data is handled by a React component. The alternative is uncontrolled components, where form data is handled by the DOM itself.

To write an uncontrolled component, instead of writing an event handler for every state update, you can [use a ref](#) to get form values from the DOM.

For example, this code accepts a single name in an uncontrolled component:

```
class NameForm extends React.Component {
  constructor(props) {
    super(props);
    this.handleSubmit = this.handleSubmit.bind(this);
    this.input = React.createRef();
  }

  handleSubmit(event) {
    alert('A name was submitted: ' + this.input.current.value);
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Name:
          <input type="text" ref={this.input} />
        </label>
        <input type="submit" value="Submit" />
      </form>
    );
  }
}
```

Since an uncontrolled component keeps the source of truth in the DOM, it is sometimes easier to integrate React and non-React code when using uncontrolled components. It can also be slightly less code if you want to be quick and dirty. Otherwise, you should usually use controlled components.

If it's still not clear which type of component you should use for a particular situation, you might find [this article on controlled versus uncontrolled inputs](#) to be helpful.

Default Values

In the React rendering lifecycle, the `value` attribute on form elements will override the value in the DOM. With an uncontrolled component, you often want React to specify the initial value, but leave subsequent updates uncontrolled. To handle this case, you can specify a `defaultValue` attribute instead of `value`. Changing the value of `defaultValue` attribute after a component has mounted will not cause any update of the value in the DOM.

```
render() {
  return (
    <form onSubmit={this.handleSubmit}>
      <label>
        Name:
        <input defaultValue="Bob" type="text" ref={this.input} />
      </label>
      <input type="submit" value="Submit" />
    </form>
  );
}
```

Likewise, `<input type="checkbox">` and `<input type="radio">` support `defaultChecked`, and `<select>` and `<textarea>` supports `defaultValue`.

The file input Tag

In HTML, an `<input type="file">` lets the user choose one or more files from their device storage to be uploaded to a server or manipulated by JavaScript via the [File API](#).

```
<input type="file" />
```

In React, an `<input type="file" />` is always an uncontrolled component because its value can only be set by a user, and not programmatically.

You should use the File API to interact with the files. The following example shows how to create a [ref to the DOM node](#) to access file(s) in a submit handler:

```
class FileInput extends React.Component {
  constructor(props) {
    super(props);
    this.handleSubmit = this.handleSubmit.bind(this);
    this.fileInput = React.createRef();
  }
  handleSubmit(event) {
    event.preventDefault();
    alert(`Selected file - ${this.fileInput.current.files[0].name}`);
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Upload file:
          <input type="file" ref={this.fileInput} />
        </label>
        <br />
        <button type="submit">Submit</button>
      </form>
    );
  }
}

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

22. Web Components

React and [Web Components](#) are built to solve different problems. Web Components provide strong encapsulation for reusable components, while React provides a declarative library that keeps the DOM in sync with your data. The two goals are complementary. As a developer, you are free to use React in your Web Components, or to use Web Components in React, or both.

Most people who use React don't use Web Components, but you may want to, especially if you are using third-party UI components that are written using Web Components.

Using Web Components in React

```
class HelloMessage extends React.Component {
  render() {
    return (
      <div>
        Hello <x-search>{this.props.name}</x-search>!
      </div>
    );
  }
}
```

Note:

Web Components often expose an imperative API. For instance, a `video` Web Component might expose `play()` and `pause()` functions. To access the imperative APIs of a Web Component, you

will need to use a ref to interact with the DOM node directly. If you are using third-party Web Components, the best solution is to write a React component that behaves as a wrapper for your Web Component.

Events emitted by a Web Component may not properly propagate through a React render tree. You will need to manually attach event handlers to handle these events within your React components.

One common confusion is that Web Components use “class” instead of “className”.

```
function BrickFlipbox() {
  return (
    <brick-flipbox class="demo">
      <div>front</div>
      <div>back</div>
    </brick-flipbox>
  );
}
```

Using React in your Web Components

```
class XSearch extends HTMLElement {
  connectedCallback() {
    const mountPoint = document.createElement('span');
    this.attachShadow({ mode: 'open' }).appendChild(mountPoint);

    const name = this.getAttribute('name');
    const url = 'https://www.google.com/search?q=' + encodeURIComponent(name);
    const root = ReactDOM.createRoot(mountPoint);
    root.render(<a href={url}>{name}</a>);
  }
}
customElements.define('x-search', XSearch);
```

Note:

This code **will not** work if you transform classes with Babel. See [this issue](#) for the discussion. Include the [custom-elements-es5-adapter](#) before you load your web components to fix this issue.

III. API REFERENCE

1. React Top-Level API

`React` is the entry point to the React library. If you load React from a `<script>` tag, these top-level APIs are available on the `React` global. If you use ES6 with npm, you can write `import React from 'react'`. If you use ES5 with npm, you can write `var React = require('react')`.

Overview

Components

React components let you split the UI into independent, reusable pieces, and think about each piece in isolation. React components can be defined by subclassing `React.Component` or `React.PureComponent`.

- `React.Component`
- `React.PureComponent`

If you don't use ES6 classes, you may use the `create-react-class` module instead. See [Using React without ES6](#) for more information.

React components can also be defined as functions which can be wrapped:

- `React.memo`

Creating React Elements

We recommend [using JSX](#) to describe what your UI should look like. Each JSX element is just syntactic sugar for calling `React.createElement()`. You will not typically invoke the following methods directly if you are using JSX.

- `createElement()`
- `createFactory()`

See [Using React without JSX](#) for more information.

Transforming Elements

`React` provides several APIs for manipulating elements:

- `cloneElement()`
- `isValidElement()`
- `React.Children`

Fragments

`React` also provides a component for rendering multiple elements without a wrapper.

- `React.Fragment`

Refs

- `React.createRef`
- `React.forwardRef`

Suspense

Suspense lets components “wait” for something before rendering. Today, Suspense only supports one use case: [loading components dynamically with `React.lazy`](#). In the future, it will support other use cases like data fetching.

- `React.lazy`
- `React.Suspense`

Transitions

Transitions are a new concurrent feature introduced in React 18. They allow you to mark updates as transitions, which tells React that they can be interrupted and avoid going back to Suspense fallbacks for already visible content.

- `React.startTransition`
- `React.useTransition`

Hooks

Hooks are a new addition in React 16.8. They let you use state and other React features without writing a class. Hooks have a [dedicated docs section](#) and a separate API reference:

- [Basic Hooks](#)
 - `useState`
 - `useEffect`
 - `useContext`
- [Additional Hooks](#)

- `useReducer`
- `useCallback`
- `useMemo`
- `useRef`
- `useImperativeHandle`
- `useLayoutEffect`
- `useDebugValue`
- `useDeferredValue`
- `useTransition`
- `useId`
- Library Hooks
 - `useSyncExternalStore`
 - `useInsertionEffect`

Reference

React.Component

`React.Component` is the base class for React components when they are defined using [ES6 classes](#):

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

See the [React.Component API Reference](#) for a list of methods and properties related to the base `React.Component` class.

React.PureComponent

`React.PureComponent` is similar to `React.Component`. The difference between them is that `React.Component` doesn't implement `shouldComponentUpdate()`, but `React.PureComponent` implements it with a shallow prop and state comparison.

If your React component's `render()` function renders the same result given the same props and state, you can use `React.PureComponent` for a performance boost in some cases.

Note

`React.PureComponent`'s `shouldComponentUpdate()` only shallowly compares the objects. If these contain complex data structures, it may produce false-negatives for deeper differences. Only extend `PureComponent` when you expect to have simple props and state, or use `forceUpdate()` when you know deep data structures have changed. Or, consider using [immutable objects](#) to facilitate fast comparisons of nested data.

Furthermore, `React.PureComponent`'s `shouldComponentUpdate()` skips prop updates for the whole component subtree. Make sure all the children components are also “pure”.

React.memo

```
const MyComponent = React.memo(function MyComponent(props) {
  /* render using props */
});
```

`React.memo` is a [higher order component](#).

If your component renders the same result given the same props, you can wrap it in a call to `React.memo` for a performance boost in some cases by memoizing the result. This means that React will skip rendering the component, and reuse the last rendered result.

`React.memo` only checks for prop changes. If your function component wrapped in `React.memo` has a `useState`, `useReducer` or `useContext` Hook in its implementation, it will still rerender when state or context change.

By default it will only shallowly compare complex objects in the props object. If you want control over the comparison, you can also provide a custom comparison function as the second argument.

```
function MyComponent(props) {
  /* render using props */
}
function areEqual(prevProps, nextProps) {
  /*
   * return true if passing nextProps to render would return
   * the same result as passing prevProps to render,
   * otherwise return false
   */
}
export default React.memo(MyComponent, areEqual);
```

This method only exists as a [performance optimization](#). Do not rely on it to “prevent” a render, as this can lead to bugs.

Note

Unlike the `shouldComponentUpdate()` method on class components, the `areEqual` function returns `true` if the props are equal and `false` if the props are not equal. This is the inverse from `shouldComponentUpdate`.

`createElement()`

```
React.createElement(type, [props], [...children]);
```

Create and return a new [React element](#) of the given type. The type argument can be either a tag name string (such as `'div'` or `'span'`), a [React component](#) type (a class or a function), or a [React fragment](#) type.

Code written with JSX will be converted to use `React.createElement()`. You will not typically invoke `React.createElement()` directly if you are using JSX. See [React Without JSX](#) to learn more.

`cloneElement()`

```
React.cloneElement(element, [config], [...children]);
```

Clone and return a new React element using `element` as the starting point. `config` should contain all new props, `key`, or `ref`. The resulting element will have the original element’s props with the new props merged in shallowly. New children will replace existing children. `key` and `ref` from the original element will be preserved if no `key` and `ref` present in the `config`.

`React.cloneElement()` is almost equivalent to:

```
<element.type {...element.props} {...props}>{children}</element.type>
```

However, it also preserves `ref`s. This means that if you get a child with a `ref` on it, you won’t accidentally steal it from your ancestor. You will get the same `ref` attached to your new element. The new `ref` or `key` will replace old ones if present.

This API was introduced as a replacement of the deprecated `React.addons.cloneWithProps()`.

createFactory()

`React.createFactory(type)`

Return a function that produces React elements of a given type. Like `React.createElement()`, the type argument can be either a tag name string (such as `'div'` or `'span'`), a [React component](#) type (a class or a function), or a [React fragment](#) type.

This helper is considered legacy, and we encourage you to either use JSX or use `React.createElement()` directly instead.

You will not typically invoke `React.createFactory()` directly if you are using JSX. See [React Without JSX](#) to learn more.

isValidElement()

`React.isValidElement(object)`

Verifies the object is a React element. Returns `true` or `false`.

React.Children

`React.Children` provides utilities for dealing with the `this.props.children` opaque data structure.

React.Children.map

`React.Children.map(children, function([thisArg]))`

Invokes a function on every immediate child contained within `children` with `this` set to `thisArg`. If `children` is an array it will be traversed and the function will be called for each child in the array. If `children` is `null` or `undefined`, this method will return `null` or `undefined` rather than an array.

Note

If `children` is a `Fragment` it will be treated as a single child and not traversed.

React.Children.forEach

`React.Children.forEach(children, function([thisArg]))`

Like `React.Children.map()` but does not return an array.

React.Children.count

`React.Children.count(children)`

Returns the total number of components in `children`, equal to the number of times that a callback passed to `map` or `forEach` would be invoked.

React.Children.only

`React.Children.only(children)`

Verifies that `children` has only one child (a React element) and returns it. Otherwise this method throws an error.

Note:

`React.Children.only()` does not accept the return value of `React.Children.map()` because it is an array rather than a React element.

React.Children.toArray

`React.Children.toArray(children)`

Returns the `children` opaque data structure as a flat array with keys assigned to each child. Useful if you want to manipulate collections of children in your render methods, especially if you want to reorder or slice `this.props.children` before passing it down.

Note:

`React.Children.toArray()` changes keys to preserve the semantics of nested arrays when flattening lists of children. That is, `toArray` prefixes each key in the returned array so that each element's key is scoped to the input array containing it.

React.Fragment

The `React.Fragment` component lets you return multiple elements in a `render()` method without creating an additional DOM element:

```
render() {
  return (
    <React.Fragment>
      Some text.
      <h2>A heading</h2>
    </React.Fragment>);
}
```

You can also use it with the shorthand `<</>` syntax. For more information, see [React v16.2.0: Improved Support for Fragments](#).

React.createRef

`React.createRef` creates a `ref` that can be attached to React elements via the `ref` attribute.

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);

    this.inputRef = React.createRef();
  }

  render() {
    return <input type="text" ref={this.inputRef} />;
  }

  componentDidMount() {
    this.inputRef.current.focus();
  }
}
```

React.forwardRef

`React.forwardRef` creates a React component that forwards the `ref` attribute it receives to another component below in the tree. This technique is not very common but is particularly useful in two scenarios:

- [Forwarding refs to DOM components](#)
- [Forwarding refs in higher-order-components](#)

`React.forwardRef` accepts a rendering function as an argument. React will call this function with `props` and `ref` as two arguments. This function should return a React node.

```
const FancyButton = React.forwardRef((props, ref) => (
  <button ref={ref} className="FancyButton">
    {props.children}
  </button>
));

// You can now get a ref directly to the DOM button:
const ref = React.createRef();
<FancyButton ref={ref}>Click me!</FancyButton>;
```

In the above example, React passes a `ref` given to `<FancyButton ref={ref}>` element as a second argument to the rendering function inside the `React.forwardRef` call. This rendering function passes the `ref` to the `<button ref={ref}>` element.

As a result, after React attaches the ref, `ref.current` will point directly to the `<button>` DOM element instance.

For more information, see [forwarding refs](#).

React.lazy

`React.lazy()` lets you define a component that is loaded dynamically. This helps reduce the bundle size to delay loading components that aren't used during the initial render.

You can learn how to use it from our [code splitting documentation](#). You might also want to check out [this article](#) explaining how to use it in more detail.

```
// This component is loaded dynamically
const SomeComponent = React.lazy(() => import('./SomeComponent'));
```

Note that rendering `lazy` components requires that there's a `<React.Suspense>` component higher in the rendering tree. This is how you specify a loading indicator.

React.Suspense

`React.Suspense` lets you specify the loading indicator in case some components in the tree below it are not yet ready to render. In the future we plan to let `Suspense` handle more scenarios such as data fetching. You can read about this in [our roadmap](#).

Today, lazy loading components is the **only** use case supported by `<React.Suspense>`:

```
// This component is loaded dynamically
const OtherComponent = React.lazy(() => import('./OtherComponent'));

function MyComponent() {
  return (
    // Displays <Spinner> until OtherComponent loads
    <React.Suspense fallback=<Spinner />>
      <div>
        <OtherComponent />
      </div>
    </React.Suspense>
  );
}
```

It is documented in our [code splitting guide](#). Note that `lazy` components can be deep inside the `Suspense` tree — it doesn't have to wrap every one of them. The best practice is to place `<Suspense>` where you want to see a loading indicator, but to use `lazy()` wherever you want to do code splitting.

Note

For content that is already shown to the user, switching back to a loading indicator can be disorienting. It is sometimes better to show the “old” UI while the new UI is being prepared. To do this, you can use the new transition APIs `startTransition` and `useTransition` to mark updates as transitions and avoid unexpected fallbacks.

React.Suspense in Server Side Rendering

During server side rendering `Suspense` Boundaries allow you to flush your application in smaller chunks by suspending. When a component suspends we schedule a low priority task to render the closest `Suspense` boundary's fallback. If the component unsuspends before we flush the fallback then we send down the actual content and throw away the fallback.

React.Suspense during hydration

`Suspense` boundaries depend on their parent boundaries being hydrated before they can hydrate, but they can hydrate independently from sibling boundaries. Events on a boundary before it is hydrated will cause the boundary to hydrate at a higher priority than neighboring boundaries. [Read more](#)

React.startTransition

`React.startTransition(callback)`

`React.startTransition` lets you mark updates inside the provided callback as transitions. This method is designed to be used when `React.useTransition` is not available.

Note:

Updates in a transition yield to more urgent updates such as clicks.

Updates in a transition will not show a fallback for re-suspended content, allowing the user to continue interacting while rendering the update.

`React.startTransition` does not provide an `isPending` flag. To track the pending status of a transition see `React.useTransition`.

React.Component

This page contains a detailed API reference for the React component class definition. It assumes you're familiar with fundamental React concepts, such as [Components and Props](#), as well as [State and Lifecycle](#). If you're not, read them first.

Overview

React lets you define components as classes or functions. Components defined as classes currently provide more features which are described in detail on this page. To define a React component class, you need to extend `React.Component`:

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

The only method you *must* define in a `React.Component` subclass is called `render()`. All the other methods described on this page are optional.

We strongly recommend against creating your own base component classes. In React components, [code reuse is primarily achieved through composition rather than inheritance](#).

Note:

React doesn't force you to use the ES6 class syntax. If you prefer to avoid it, you may use the `create-react-class` module or a similar custom abstraction instead. Take a look at [Using React without ES6](#) to learn more.

The Component Lifecycle

Each component has several "lifecycle methods" that you can override to run code at particular times in the process. **You can use [this lifecycle diagram as a cheat sheet](#)**. In the list below, commonly used lifecycle methods are marked as **bold**. The rest of them exist for relatively rare use cases.

Mounting

These methods are called in the following order when an instance of a component is being created and inserted into the DOM:

- `constructor()`
- `static getDerivedStateFromProps()`
- **`render()`**
- `componentDidMount()`

Note:

This method is considered legacy and you should [avoid it](#) in new code:

- `UNSAFE_componentWillMount()`

Updating

An update can be caused by changes to props or state. These methods are called in the following order when a component is being re-rendered:

- `static getDerivedStateFromProps()`
- `shouldComponentUpdate()`
- `render()`
- `getSnapshotBeforeUpdate()`
- `componentDidUpdate()`

Note:

These methods are considered legacy and you should avoid them in new code:

- `UNSAFE_componentWillUpdate()`
- `UNSAFE_componentWillReceiveProps()`

Unmounting

This method is called when a component is being removed from the DOM:

- `componentWillUnmount()`

Error Handling

These methods are called when there is an error during rendering, in a lifecycle method, or in the constructor of any child component.

- `static getDerivedStateFromError()`
- `componentDidCatch()`

Other APIs

Each component also provides some other APIs:

- `setState()`
- `forceUpdate()`

Class Properties

- `defaultProps`
- `displayName`

Instance Properties

- `props`
- `state`

Reference

Commonly Used Lifecycle Methods

The methods in this section cover the vast majority of use cases you'll encounter creating React components. **For a visual reference, check out [this lifecycle diagram](#).**

`render()`

`render()`

The `render()` method is the only required method in a class component.

When called, it should examine `this.props` and `this.state` and return one of the following types:

- **React elements.** Typically created via `JSX`. For example, `<div />` and `<MyComponent />` are React elements that instruct React to render a DOM node, or another user-defined component, respectively.
- **Arrays and fragments.** Let you return multiple elements from render. See the documentation on [fragments](#) for more details.
- **Portals.** Let you render children into a different DOM subtree. See the documentation on [portals](#) for more details.
- **String and numbers.** These are rendered as text nodes in the DOM.
- **Booleans or `null` or `undefined`.** Render nothing. (Mostly exists to support `return test && <Child />` pattern, where `test` is boolean).

The `render()` function should be pure, meaning that it does not modify component state, it returns the same result each time it's invoked, and it does not directly interact with the browser.

If you need to interact with the browser, perform your work in `componentDidMount()` or the other lifecycle methods instead. Keeping `render()` pure makes components easier to think about.

Note

`render()` will not be invoked if `shouldComponentUpdate()` returns false.

constructor()

`constructor(props)`

If you don't initialize state and you don't bind methods, you don't need to implement a constructor for your React component.

The constructor for a React component is called before it is mounted. When implementing the constructor for a `React.Component` subclass, you should call `super(props)` before any other statement. Otherwise, `this.props` will be undefined in the constructor, which can lead to bugs.

Typically, in React constructors are only used for two purposes:

- Initializing [local state](#) by assigning an object to `this.state`.
- Binding [event handler](#) methods to an instance.

You **should not call** `setState()` in the `constructor()`. Instead, if your component needs to use local state, **assign the initial state to** `this.state` directly in the constructor:

```
constructor(props) {
  super(props);
  // Don't call this.setState() here!
  this.state = { counter: 0 };
  this.handleClick = this.handleClick.bind(this);
}
```

Constructor is the only place where you should assign `this.state` directly. In all other methods, you need to use `this.setState()` instead.

Avoid introducing any side-effects or subscriptions in the constructor. For those use cases, use `componentDidMount()` instead.

Note

Avoid copying props into state! This is a common mistake:

```
constructor(props) {
  super(props);
  // Don't do this!
  this.state = { color: props.color };
}
```


The problem is that it's both unnecessary (you can use `this.props.color` directly instead), and creates bugs (updates to the `color` prop won't be reflected in the state).

Only use this pattern if you intentionally want to ignore prop updates. In that case, it makes sense to rename the prop to be called `initialColor` or `defaultColor`. You can then force a component to “reset” its internal state by changing its `key` when necessary.

Read our [blog post on avoiding derived state](#) to learn about what to do if you think you need some state to depend on the props.

`componentDidMount()`

`componentDidMount()`

`componentDidMount()` is invoked immediately after a component is mounted (inserted into the tree). Initialization that requires DOM nodes should go here. If you need to load data from a remote endpoint, this is a good place to instantiate the network request.

This method is a good place to set up any subscriptions. If you do that, don't forget to unsubscribe in `componentWillUnmount()`.

You **may call `setState()` immediately** in `componentDidMount()`. It will trigger an extra rendering, but it will happen before the browser updates the screen. This guarantees that even though the `render()` will be called twice in this case, the user won't see the intermediate state. Use this pattern with caution because it often causes performance issues. In most cases, you should be able to assign the initial state in the `constructor()` instead. It can, however, be necessary for cases like modals and tooltips when you need to measure a DOM node before rendering something that depends on its size or position.

`componentDidUpdate()`

`componentDidUpdate(prevProps, prevState, snapshot)`

`componentDidUpdate()` is invoked immediately after updating occurs. This method is not called for the initial render.

Use this as an opportunity to operate on the DOM when the component has been updated. This is also a good place to do network requests as long as you compare the current props to previous props (e.g. a network request may not be necessary if the props have not changed).

```
componentDidUpdate(prevProps) {
  // Typical usage (don't forget to compare props):
  if (this.props.userID !== prevProps.userID) {
    this.fetchData(this.props.userID);
  }
}
```

You **may call `setState()` immediately** in `componentDidUpdate()` but note that **it must be wrapped in a condition** like in the example above, or you'll cause an infinite loop. It would also cause an extra re-rendering which, while not visible to the user, can affect the component performance. If you're trying to “mirror” some state to a prop coming from above, consider using the prop directly instead. Read more about [why copying props into state causes bugs](#).

If your component implements the `getSnapshotBeforeUpdate()` lifecycle (which is rare), the value it returns will be passed as a third “snapshot” parameter to `componentDidUpdate()`. Otherwise this parameter will be undefined.

Note

`componentDidUpdate()` will not be invoked if `shouldComponentUpdate()` returns false.

`componentWillUnmount()`

`componentWillUnmount()`

`componentWillUnmount()` is invoked immediately before a component is unmounted and destroyed. Perform any necessary cleanup in this method, such as invalidating timers, canceling network requests, or cleaning up any subscriptions that were created in `componentDidMount()`.

You **should not call** `setState()` in `componentWillUnmount()` because the component will never be re-rendered. Once a component instance is unmounted, it will never be mounted again.

Rarely Used Lifecycle Methods

The methods in this section correspond to uncommon use cases. They're handy once in a while, but most of your components probably don't need any of them. **You can see most of the methods below on [this lifecycle diagram](#) if you click the “Show less common lifecycles” checkbox at the top of it.**

`shouldComponentUpdate()`

`shouldComponentUpdate(nextProps, nextState)`

Use `shouldComponentUpdate()` to let React know if a component's output is not affected by the current change in state or props. The default behavior is to re-render on every state change, and in the vast majority of cases you should rely on the default behavior.

`shouldComponentUpdate()` is invoked before rendering when new props or state are being received. Defaults to `true`. This method is not called for the initial render or when `forceUpdate()` is used.

This method only exists as a **performance optimization**. Do not rely on it to “prevent” a rendering, as this can lead to bugs. **Consider using the built-in `PureComponent`** instead of writing `shouldComponentUpdate()` by hand. `PureComponent` performs a shallow comparison of props and state, and reduces the chance that you'll skip a necessary update.

If you are confident you want to write it by hand, you may compare `this.props` with `nextProps` and `this.state` with `nextState` and return `false` to tell React the update can be skipped. Note that returning `false` does not prevent child components from re-rendering when *their* state changes.

We do not recommend doing deep equality checks or using `JSON.stringify()` in `shouldComponentUpdate()`. It is very inefficient and will harm performance.

Currently, if `shouldComponentUpdate()` returns `false`, then `UNSAFE_componentWillUpdate()`, `render()`, and `componentDidUpdate()` will not be invoked. In the future React may treat `shouldComponentUpdate()` as a hint rather than a strict directive, and returning `false` may still result in a re-rendering of the component.

`static getDerivedStateFromProps()`

`static getDerivedStateFromProps(props, state)`

`getDerivedStateFromProps` is invoked right before calling the render method, both on the initial mount and on subsequent updates. It should return an object to update the state, or `null` to update nothing.

This method exists for **rare use cases** where the state depends on changes in props over time. For example, it might be handy for implementing a `<Transition>` component that compares its previous and next children to decide which of them to animate in and out.

Deriving state leads to verbose code and makes your components difficult to think about. [Make sure you're familiar with simpler alternatives:](#)

- If you need to **perform a side effect** (for example, data fetching or an animation) in response to a change in props, use `componentDidUpdate` lifecycle instead.
- If you want to **re-compute some data only when a prop changes**, [use a memoization helper instead](#).
- If you want to **“reset” some state when a prop changes**, consider either making a component [fully controlled](#) or [fully uncontrolled with a key](#) instead.

This method doesn't have access to the component instance. If you'd like, you can reuse some code between `getDerivedStateFromProps()` and the other class methods by extracting pure functions of the component props and state outside the class definition.

Note that this method is fired on *every* render, regardless of the cause. This is in contrast to `UNSAFE_componentWillReceiveProps`, which only fires when the parent causes a re-render and not as a result of a local `setState`.

`getSnapshotBeforeUpdate()`

`getSnapshotBeforeUpdate(prevProps, prevState)`

`getSnapshotBeforeUpdate()` is invoked right before the most recently rendered output is committed to e.g. the DOM. It enables your component to capture some information from the DOM (e.g. scroll position) before it is potentially changed. Any value returned by this lifecycle method will be passed as a parameter to `componentDidUpdate()`.

This use case is not common, but it may occur in UIs like a chat thread that need to handle scroll position in a special way.

A snapshot value (or `null`) should be returned.

For example:

```
class ScrollingList extends React.Component {
  constructor(props) {
    super(props);
    this.listRef = React.createRef();
  }

  getSnapshotBeforeUpdate(prevProps, prevState) {
    // Are we adding new items to the list?
    // Capture the scroll position so we can adjust scroll later.
    if (prevProps.list.length < this.props.list.length) {
      const list = this.listRef.current;
      return list.scrollHeight - list.scrollTop;
    }
    return null;
  }

  componentDidUpdate(prevProps, prevState, snapshot) {
    // If we have a snapshot value, we've just added new items.
    // Adjust scroll so these new items don't push the old ones out of view.
    // (snapshot here is the value returned from getSnapshotBeforeUpdate)
    if (snapshot !== null) {
      const list = this.listRef.current;
      list.scrollTop = list.scrollHeight - snapshot;
    }
  }

  render() {
    return <div ref={this.listRef}>{/* ...contents... */}</div>;
  }
}
```

In the above examples, it is important to read the `scrollHeight` property in `getSnapshotBeforeUpdate` because there may be delays between “render” phase lifecycles (like `render`) and “commit” phase lifecycles (like `getSnapshotBeforeUpdate` and `componentDidUpdate`).

Error boundaries

Error boundaries are React components that catch JavaScript errors anywhere in their child component tree, log those errors, and display a fallback UI instead of the component tree that crashed. Error boundaries catch errors during rendering, in lifecycle methods, and in constructors of the whole tree below them.

A class component becomes an error boundary if it defines either (or both) of the lifecycle methods `static getDerivedStateFromError()` or `componentDidCatch()`. Updating state from these lifecycles lets you capture an unhandled JavaScript error in the below tree and display a fallback UI.

Only use error boundaries for recovering from unexpected exceptions; **don't try to use them for control flow**.

For more details, see [Error Handling in React 16](#).

Note

Error boundaries only catch errors in the components **below** them in the tree. An error boundary can't catch an error within itself.

`static getDerivedStateFromError()`

```
static getDerivedStateFromError(error)
```

This lifecycle is invoked after an error has been thrown by a descendant component. It receives the error that was thrown as a parameter and should return a value to update state.

```
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 };
  }

  render() {
    if (this.state.hasError) {
      // You can render any custom fallback UI
      return <h1>Something went wrong.</h1>;
    }
    return this.props.children;
  }
}
```

Note

`getDerivedStateFromError()` is called during the “render” phase, so side-effects are not permitted. For those use cases, use `componentDidCatch()` instead.

`componentDidCatch()`

`componentDidCatch(error, info)`

This lifecycle is invoked after an error has been thrown by a descendant component. It receives two parameters:

1. `error` - The error that was thrown.
2. `info` - An object with a `componentStack` key containing information about which component threw the error.

`componentDidCatch()` is called during the “commit” phase, so side-effects are permitted. It should be used for things like logging errors:

```
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, info) {
    // Example "componentStack":
    //   in ComponentThatThrows (created by App)
    //   in ErrorBoundary (created by App)
    //   in div (created by App)
    //   in App
    logComponentStackToMyService(info.componentStack);
  }

  render() {
    if (this.state.hasError) {
      // You can render any custom fallback UI
      return <h1>Something went wrong.</h1>;
    }

    return this.props.children;
  }
}
```

Production and development builds of React slightly differ in the way `componentDidCatch()` handles errors.

On development, the errors will bubble up to `window`, this means that any `window.onerror` or `window.addEventListener('error', callback)` will intercept the errors that have been caught by `componentDidCatch()`.

On production, instead, the errors will not bubble up, which means any ancestor error handler will only receive errors not explicitly caught by `componentDidCatch()`.

Note

In the event of an error, you can render a fallback UI with `componentDidCatch()` by calling `setState`, but this will be deprecated in a future release. Use `static getDerivedStateFromError()` to handle fallback rendering instead.

Legacy Lifecycle Methods

The lifecycle methods below are marked as “legacy”. They still work, but we don’t recommend using them in the new code. You can learn more about migrating away from legacy lifecycle methods in [this blog post](#).

`UNSAFE_componentWillMount()`

`UNSAFE_componentWillMount()`

Note

This lifecycle was previously named `componentWillMount`. That name will continue to work until version 17. Use the [rename-unsafe-lifecycles](#) [codemod](#) to automatically update your components.

`UNSAFE_componentWillMount()` is invoked just before mounting occurs. It is called before `render()`, therefore calling `setState()` synchronously in this method will not trigger an extra rendering. Generally, we recommend using the `constructor()` instead for initializing state.

Avoid introducing any side-effects or subscriptions in this method. For those use cases, use `componentDidMount()` instead.

This is the only lifecycle method called on server rendering.

`UNSAFE_componentWillReceiveProps()`

`UNSAFE_componentWillReceiveProps(nextProps)`

Note

This lifecycle was previously named `componentWillReceiveProps`. That name will continue to work until version 17. Use the [rename-unsafe-lifecycles](#) [codemod](#) to automatically update your components.

Note:

Using this lifecycle method often leads to bugs and inconsistencies

- If you need to **perform a side effect** (for example, data fetching or an animation) in response to a change in props, use `componentDidUpdate` lifecycle instead.
- If you used `componentWillReceiveProps` for **re-computing some data only when a prop changes**, [use a memoization helper instead](#).
- If you used `componentWillReceiveProps` to **“reset” some state when a prop changes**, consider either making a component [fully controlled](#) or [fully uncontrolled with a key](#) instead.

For other use cases, [follow the recommendations in this blog post about derived state](#).

`UNSAFE_componentWillReceiveProps()` is invoked before a mounted component receives new props. If you need to update the state in response to prop changes (for example, to reset it), you may compare `this.props` and `nextProps` and perform state transitions using `this.setState()` in this method.

Note that if a parent component causes your component to re-render, this method will be called even if props have not changed. Make sure to compare the current and next values if you only want to handle changes.

React doesn’t call `UNSAFE_componentWillReceiveProps()` with initial props during [mounting](#). It only calls this method if some of component’s props may update. Calling `this.setState()` generally doesn’t trigger `UNSAFE_componentWillReceiveProps()`.

UNSAFE_componentWillUpdate()

```
UNSAFE_componentWillUpdate(nextProps, nextState)
```

Note

This lifecycle was previously named `componentWillUpdate`. That name will continue to work until version 17. Use the [rename-unsafe-lifecycles](#) [codemod](#) to automatically update your components.

`UNSAFE_componentWillUpdate()` is invoked just before rendering when new props or state are being received. Use this as an opportunity to perform preparation before an update occurs. This method is not called for the initial render.

Note that you cannot call `this.setState()` here; nor should you do anything else (e.g. dispatch a Redux action) that would trigger an update to a React component before `UNSAFE_componentWillUpdate()` returns.

Typically, this method can be replaced by `componentDidUpdate()`. If you were reading from the DOM in this method (e.g. to save a scroll position), you can move that logic to `getSnapshotBeforeUpdate()`.

Note

`UNSAFE_componentWillUpdate()` will not be invoked if `shouldComponentUpdate()` returns false.

Other APIs

Unlike the lifecycle methods above (which React calls for you), the methods below are the methods *you* can call from your components.

There are just two of them: `setState()` and `forceUpdate()`.

setState()

```
setState(updater[, callback])
```

`setState()` enqueues changes to the component state and tells React that this component and its children need to be re-rendered with the updated state. This is the primary method you use to update the user interface in response to event handlers and server responses.

Think of `setState()` as a *request* rather than an immediate command to update the component. For better perceived performance, React may delay it, and then update several components in a single pass. In the rare case that you need to force the DOM update to be applied synchronously, you may wrap it in `flushSync`, but this may hurt performance.

`setState()` does not always immediately update the component. It may batch or defer the update until later. This makes reading `this.state` right after calling `setState()` a potential pitfall. Instead, use `componentDidUpdate` or a `setState` callback (`setState(updater, callback)`), either of which are guaranteed to fire after the update has been applied. If you need to set the state based on the previous state, read about the `updater` argument below.

`setState()` will always lead to a re-render unless `shouldComponentUpdate()` returns `false`. If mutable objects are being used and conditional rendering logic cannot be implemented in `shouldComponentUpdate()`, calling `setState()` only when the new state differs from the previous state will avoid unnecessary re-renders.

The first argument is an `updater` function with the signature:

```
(state, props) => stateChange
```

`state` is a reference to the component state at the time the change is being applied. It should not be directly mutated. Instead, changes should be represented by building a new object based on the input from `state` and `props`. For instance, suppose we wanted to increment a value in state by `props.step`:

```
this.setState((state, props) => {  
  return { counter: state.counter + props.step };  
});
```

Both `state` and `props` received by the updater function are guaranteed to be up-to-date. The output of the updater is shallowly merged with `state`.

The second parameter to `setState()` is an optional callback function that will be executed once `setState` is completed and the component is re-rendered. Generally we recommend using `componentDidUpdate()` for such logic instead.

You may optionally pass an object as the first argument to `setState()` instead of a function:

```
setState(stateChange[, callback])
```

This performs a shallow merge of `stateChange` into the new state, e.g., to adjust a shopping cart item quantity:

```
this.setState({quantity: 2})
```

This form of `setState()` is also asynchronous, and multiple calls during the same cycle may be batched together. For example, if you attempt to increment an item quantity more than once in the same cycle, that will result in the equivalent of:

```
Object.assign(
  previousState,
  {quantity: state.quantity + 1},
  {quantity: state.quantity + 1},
  ...
)
```

Subsequent calls will override values from previous calls in the same cycle, so the quantity will only be incremented once. If the next state depends on the current state, we recommend using the updater function form, instead:

```
this.setState((state) => {
  return { quantity: state.quantity + 1 };
});
```

For more detail, see:

- [State and Lifecycle guide](#)
- [In depth: When and why are `setState\(\)` calls batched?](#)
- [In depth: Why isn't `this.state` updated immediately?](#)

forceUpdate()

```
component.forceUpdate(callback)
```

By default, when your component's state or props change, your component will re-render. If your `render()` method depends on some other data, you can tell React that the component needs re-rendering by calling `forceUpdate()`.

Calling `forceUpdate()` will cause `render()` to be called on the component, skipping `shouldComponentUpdate()`. This will trigger the normal lifecycle methods for child components, including the `shouldComponentUpdate()` method of each child. React will still only update the DOM if the markup changes.

Normally you should try to avoid all uses of `forceUpdate()` and only read from `this.props` and `this.state` in `render()`.

Class Properties

defaultProps

`defaultProps` can be defined as a property on the component class itself, to set the default props for the class. This is used for `undefined` props, but not for `null` props. For example:

```
class CustomButton extends React.Component {
  // ...
}

CustomButton.defaultProps = {
  color: 'blue'
};
```

If `props.color` is not provided, it will be set by default to `'blue'`:

```
render() {
  return <CustomButton /> ; // props.color will be set to blue
```

```
}
```

If `props.color` is set to `null`, it will remain `null`:

```
render() {  
  return <CustomButton color={null} /> ; // props.color will remain null  
}
```

displayName

The `displayName` string is used in debugging messages. Usually, you don't need to set it explicitly because it's inferred from the name of the function or class that defines the component. You might want to set it explicitly if you want to display a different name for debugging purposes or when you create a higher-order component, see [Wrap the Display Name for Easy Debugging](#) for details.

Instance Properties

props

`this.props` contains the props that were defined by the caller of this component. See [Components and Props](#) for an introduction to props.

In particular, `this.props.children` is a special prop, typically defined by the child tags in the JSX expression rather than in the tag itself.

state

The state contains data specific to this component that may change over time. The state is user-defined, and it should be a plain JavaScript object.

If some value isn't used for rendering or data flow (for example, a timer ID), you don't have to put it in the state. Such values can be defined as fields on the component instance.

See [State and Lifecycle](#) for more information about the state.

Never mutate `this.state` directly, as calling `setState()` afterwards may replace the mutation you made. Treat `this.state` as if it were immutable.

2. ReactDOM

The `react-dom` package provides DOM-specific methods that can be used at the top level of your app and as an escape hatch to get outside the React model if you need to.

```
import * as ReactDOM from 'react-dom';
```

If you use ES5 with npm, you can write:

```
var ReactDOM = require('react-dom');
```

The `react-dom` package also provides modules specific to client and server apps:

- `react-dom/client`
- `react-dom/server`

Overview

The `react-dom` package exports these methods:

- `createPortal()`
- `flushSync()`

These `react-dom` methods are also exported, but are considered legacy:

- `render()`

- `hydrate()`
- `findDOMNode()`
- `unmountComponentAtNode()`

Note:

Both `render` and `hydrate` have been replaced with new [client methods](#) in React 18. These methods will warn that your app will behave as if it's running React 17 (learn more [here](#)).

Browser Support

React supports all modern browsers, although [some polyfills are required](#) for older versions.

Note

We do not support older browsers that don't support ES5 methods or microtasks such as Internet Explorer. You may find that your apps do work in older browsers if polyfills such as [es5-shim](#) and [es5-sham](#) are included in the page, but you're on your own if you choose to take this path.

Reference

`createPortal()`

`createPortal(child, container)`

Creates a portal. Portals provide a way to [render children into a DOM node that exists outside the hierarchy of the DOM component](#).

`flushSync()`

`flushSync(callback)`

Force React to flush any updates inside the provided callback synchronously. This ensures that the DOM is updated immediately.

```
// Force this state update to be synchronous.
flushSync(() => {
  setCount(count + 1);
});
// By this point, DOM is updated.
```

Note:

`flushSync` can significantly hurt performance. Use sparingly.

`flushSync` may force pending Suspense boundaries to show their `fallback` state.

`flushSync` may also run pending effects and synchronously apply any updates they contain before returning.

`flushSync` may also flush updates outside the callback when necessary to flush the updates inside the callback. For example, if there are pending updates from a click, React may flush those before flushing the updates inside the callback.

Legacy Reference

`render()`

`render(element, container[, callback])`

Note:

`render` has been replaced with `createRoot` in React 18. See [createRoot](#) for more info.

Render a React element into the DOM in the supplied `container` and return a [reference](#) to the component (or returns `null` for [stateless components](#)).

If the React element was previously rendered into `container`, this will perform an update on it and only mutate the DOM as necessary to reflect the latest React element.

If the optional callback is provided, it will be executed after the component is rendered or updated.

Note:

`render()` controls the contents of the container node you pass in. Any existing DOM elements inside are replaced when first called. Later calls use React's DOM diffing algorithm for efficient updates.

`render()` does not modify the container node (only modifies the children of the container). It may be possible to insert a component to an existing DOM node without overwriting the existing children.

`render()` currently returns a reference to the root `ReactComponent` instance. However, using this return value is legacy and should be avoided because future versions of React may render components asynchronously in some cases. If you need a reference to the root `ReactComponent` instance, the preferred solution is to attach a [callback ref](#) to the root element.

Using `render()` to hydrate a server-rendered container is deprecated. Use `hydrateRoot()` instead.

hydrate()

```
hydrate(element, container[, callback])
```

Note:

`hydrate` has been replaced with `hydrateRoot` in React 18. See [hydrateRoot](#) for more info.

Same as `render()`, but is used to hydrate a container whose HTML contents were rendered by `ReactDOMServer`. React will attempt to attach event listeners to the existing markup.

React expects that the rendered content is identical between the server and the client. It can patch up differences in text content, but you should treat mismatches as bugs and fix them. In development mode, React warns about mismatches during hydration. There are no guarantees that attribute differences will be patched up in case of mismatches. This is important for performance reasons because in most apps, mismatches are rare, and so validating all markup would be prohibitively expensive.

If a single element's attribute or text content is unavoidably different between the server and the client (for example, a timestamp), you may silence the warning by adding `suppressHydrationWarning={true}` to the element. It only works one level deep, and is intended to be an escape hatch. Don't overuse it. Unless it's text content, React still won't attempt to patch it up, so it may remain inconsistent until future updates.

If you intentionally need to render something different on the server and the client, you can do a two-pass rendering. Components that render something different on the client can read a state variable like `this.state.isClient`, which you can set to `true` in `componentDidMount()`. This way the initial render pass will render the same content as the server, avoiding mismatches, but an additional pass will happen synchronously right after hydration. Note that this approach will make your components slower because they have to render twice, so use it with caution.

Remember to be mindful of user experience on slow connections. The JavaScript code may load significantly later than the initial HTML render, so if you render something different in the client-only pass, the transition can be jarring. However, if executed well, it may be beneficial to render a "shell" of the application on the server, and only show some of the extra widgets on the client. To learn how to do this without getting the markup mismatch issues, refer to the explanation in the previous paragraph.

unmountComponentAtNode()

```
unmountComponentAtNode(container)
```

Note:

`unmountComponentAtNode` has been replaced with `root.unmount()` in React 18. See [createRoot](#) for more info.

Remove a mounted React component from the DOM and clean up its event handlers and state. If no component was mounted in the container, calling this function does nothing. Returns `true` if a component was unmounted and `false` if there was no component to unmount.

findDOMNode()

Note:

`findDOMNode` is an escape hatch used to access the underlying DOM node. In most cases, use of this escape hatch is discouraged because it pierces the component abstraction. It has been deprecated in `StrictMode`.

```
findDOMNode(component)
```

If this component has been mounted into the DOM, this returns the corresponding native browser DOM element. This method is useful for reading values out of the DOM, such as form field values and performing DOM measurements. **In most cases, you can attach a ref to the DOM node and avoid using `findDOMNode` at all.**

When a component renders to `null` or `false`, `findDOMNode` returns `null`. When a component renders to a string, `findDOMNode` returns a text DOM node containing that value. As of React 16, a component may return a fragment with multiple children, in which case `findDOMNode` will return the DOM node corresponding to the first non-empty child.

Note:

`findDOMNode` only works on mounted components (that is, components that have been placed in the DOM). If you try to call this on a component that has not been mounted yet (like calling `findDOMNode()` in `render()` on a component that has yet to be created) an exception will be thrown.

`findDOMNode` cannot be used on function components.

3. ReactDOMClient

The `react-dom/client` package provides client-specific methods used for initializing an app on the client. Most of your components should not need to use this module.

```
import * as ReactDOM from 'react-dom/client';
```

If you use ES5 with npm, you can write:

```
var ReactDOM = require('react-dom/client');
```

Overview

The following methods can be used in client environments:

- `createRoot()`
- `hydrateRoot()`

Browser Support

React supports all modern browsers, although some polyfills are required for older versions.

Note

We do not support older browsers that don't support ES5 methods or microtasks such as Internet Explorer. You may find that your apps do work in older browsers if polyfills such as [es5-shim and es5-sham](#) are included in the page, but you're on your own if you choose to take this path.

Reference

`createRoot()`

```
createRoot(container[, options]);
```

Create a React root for the supplied `container` and return the root. The root can be used to render a React element into the DOM with `render`:

```
const root = createRoot(container);
root.render(element);
```

`createRoot` accepts two options:

- `onRecoverableError`: optional callback called when React automatically recovers from errors.
- `identifierPrefix`: optional prefix React uses for ids generated by `React.useId`. Useful to avoid conflicts when using multiple roots on the same page. Must be the same prefix used on the server.

The root can also be unmounted with `unmount`:

```
root.unmount();
```

Note:

`createRoot()` controls the contents of the container node you pass in. Any existing DOM elements inside are replaced when render is called. Later calls use React's DOM diffing algorithm for efficient updates.

`createRoot()` does not modify the container node (only modifies the children of the container). It may be possible to insert a component to an existing DOM node without overwriting the existing children.

Using `createRoot()` to hydrate a server-rendered container is not supported.

Use `hydrateRoot()` instead.

`hydrateRoot()`

```
hydrateRoot(container, element[, options])
```

Same as `createRoot()`, but is used to hydrate a container whose HTML contents were rendered by `ReactDOMServer`. React will attempt to attach event listeners to the existing markup.

`hydrateRoot` accepts two options:

- `onRecoverableError`: optional callback called when React automatically recovers from errors.
- `identifierPrefix`: optional prefix React uses for ids generated by `React.useId`. Useful to avoid conflicts when using multiple roots on the same page. Must be the same prefix used on the server.

Note

React expects that the rendered content is identical between the server and the client. It can patch up differences in text content, but you should treat mismatches as bugs and fix them. In development mode, React warns about mismatches during hydration. There are no guarantees that attribute differences will be patched up in case of mismatches. This is important for

performance reasons because in most apps, mismatches are rare, and so validating all markup would be prohibitively expensive.

4. ReactDOMServer

The `ReactDOMServer` object enables you to render components to static markup. Typically, it's used on a Node server:

```
// ES modules
import * as ReactDOMServer from 'react-dom/server';
// CommonJS
var ReactDOMServer = require('react-dom/server');
```

Overview

These methods are only available in the **environments with Node.js Streams**:

- `renderToPipeableStream()`
- `renderToNodeStream()` (Deprecated)
- `renderToStaticNodeStream()`

These methods are only available in the **environments with Web Streams** (this includes browsers, Deno, and some modern edge runtimes):

- `renderToReadableStream()`

The following methods can be used in the environments that don't support streams:

- `renderToString()`
- `renderToStaticMarkup()`

Reference

`renderToPipeableStream()`

`ReactDOMServer.renderToPipeableStream(element, options)`

Render a React element to its initial HTML. Returns a stream with a `pipe(res)` method to pipe the output and `abort()` to abort the request. Fully supports Suspense and streaming of HTML with “delayed” content blocks “popping in” via inline `<script>` tags later. [Read more](#)

If you call `ReactDOM.hydrateRoot()` on a node that already has this server-rendered markup, React will preserve it and only attach event handlers, allowing you to have a very performant first-load experience.

```
let didError = false;
const stream = renderToPipeableStream(<App />, {
  onShellReady() {
    // The content above all Suspense boundaries is ready.
    // If something errored before we started streaming, we set the error
    // code appropriately.
    res.statusCode = didError ? 500 : 200;
    res.setHeader('Content-type', 'text/html');
    stream.pipe(res);
  },
  onShellError(error) {
    // Something errored before we could complete the shell so we emit an alternative
    // shell.
    res.statusCode = 500;
    res.send('<!doctype html><p>Loading...</p><script src="clientrender.js"></script>');
  },
  onAllReady() {
    // If you don't want streaming, use this instead of onShellReady.
    // This will fire after the entire page content is ready.
    // You can use this for crawlers or static generation.
    // res.statusCode = didError ? 500 : 200;
    // res.setHeader('Content-type', 'text/html');
    // stream.pipe(res);
  },
  onError(err) {
```

```

    didError = true;
    console.error(err);
  }
});

```

See the [full list of options](#).

Note:

This is a Node.js-specific API. Environments with [Web Streams](#), like Deno and modern edge runtimes, should use [renderToReadableStream](#) instead.

renderToReadableStream()

```
ReactDOMServer.renderToReadableStream(element, options);
```

Streams a React element to its initial HTML. Returns a Promise that resolves to a [Readable Stream](#). Fully supports Suspense and streaming of HTML. [Read more](#)

If you call [ReactDOM.hydrateRoot\(\)](#) on a node that already has this server-rendered markup, React will preserve it and only attach event handlers, allowing you to have a very performant first-load experience.

```

let controller = new AbortController();
let didError = false;
try {
  let stream = await renderToReadableStream(
    <html>
      <body>Success</body>
    </html>,
    {
      signal: controller.signal,
      onError(error) {
        didError = true;
        console.error(error);
      }
    }
  );

  // This is to wait for all Suspense boundaries to be ready. You can uncomment
  // this line if you want to buffer the entire HTML instead of streaming it.
  // You can use this for crawlers or static generation:

  // await stream.allReady;

  return new Response(stream, {
    status: didError ? 500 : 200,
    headers: { 'Content-Type': 'text/html' }
  });
} catch (error) {
  return new Response('<!doctype html><p>Loading...</p><script src="clientrender.js"></script>', {
    status: 500,
    headers: { 'Content-Type': 'text/html' }
  });
}

```

See the [full list of options](#).

Note:

This API depends on [Web Streams](#). For Node.js, use [renderToPipeableStream](#) instead.

renderToNodeStream() (Deprecated)

```
ReactDOMServer.renderToNodeStream(element)
```

Render a React element to its initial HTML. Returns a [Node.js Readable stream](#) that outputs an HTML string. The HTML output by this stream is exactly equal to what [ReactDOMServer.renderToString](#) would return. You can use this method to generate HTML on the server and send the markup down on the initial request for faster page loads and to allow search engines to crawl your pages for SEO purposes.

If you call `ReactDOM.hydrateRoot()` on a node that already has this server-rendered markup, React will preserve it and only attach event handlers, allowing you to have a very performant first-load experience.

Note:

Server-only. This API is not available in the browser.

The stream returned from this method will return a byte stream encoded in utf-8. If you need a stream in another encoding, take a look at a project like [iconv-lite](#), which provides transform streams for transcoding text.

`renderToStaticNodeStream()`

`ReactDOMServer.renderToStaticNodeStream(element)`

Similar to `renderToNodeStream`, except this doesn't create extra DOM attributes that React uses internally, such as `data-reactroot`. This is useful if you want to use React as a simple static page generator, as stripping away the extra attributes can save some bytes.

The HTML output by this stream is exactly equal to what `ReactDOMServer.renderToStaticMarkup` would return.

If you plan to use React on the client to make the markup interactive, do not use this method. Instead, use `renderToNodeStream` on the server and `ReactDOM.hydrateRoot()` on the client.

Note:

Server-only. This API is not available in the browser.

The stream returned from this method will return a byte stream encoded in utf-8. If you need a stream in another encoding, take a look at a project like [iconv-lite](#), which provides transform streams for transcoding text.

`renderToString()`

`ReactDOMServer.renderToString(element)`

Render a React element to its initial HTML. React will return an HTML string. You can use this method to generate HTML on the server and send the markup down on the initial request for faster page loads and to allow search engines to crawl your pages for SEO purposes.

If you call `ReactDOM.hydrateRoot()` on a node that already has this server-rendered markup, React will preserve it and only attach event handlers, allowing you to have a very performant first-load experience.

Note

This API has limited Suspense support and does not support streaming.

On the server, it is recommended to use either `renderToPipeableStream` (for Node.js) or `renderToReadableStream` (for Web Streams) instead.

`renderToStaticMarkup()`

`ReactDOMServer.renderToStaticMarkup(element)`

Similar to `renderToString`, except this doesn't create extra DOM attributes that React uses internally, such as `data-reactroot`. This is useful if you want to use React as a simple static page generator, as stripping away the extra attributes can save some bytes.

If you plan to use React on the client to make the markup interactive, do not use this method. Instead, use `renderToString` on the server and `ReactDOM.hydrateRoot()` on the client.

5. DOM Elements

React implements a browser-independent DOM system for performance and cross-browser compatibility. We took the opportunity to clean up a few rough edges in browser DOM implementations.

In React, all DOM properties and attributes (including event handlers) should be camelCased. For example, the HTML attribute `tabindex` corresponds to the attribute `tabIndex` in React. The exception is `aria-*` and `data-*` attributes, which should be lowercased. For example, you can keep `aria-label` as `aria-label`.

Differences In Attributes

There are a number of attributes that work differently between React and HTML:

checked

The `checked` attribute is supported by `<input>` components of type `checkbox` or `radio`. You can use it to set whether the component is checked. This is useful for building controlled components. `defaultChecked` is the uncontrolled equivalent, which sets whether the component is checked when it is first mounted.

className

To specify a CSS class, use the `className` attribute. This applies to all regular DOM and SVG elements like `<div>`, `<a>`, and others.

If you use React with Web Components (which is uncommon), use the `class` attribute instead.

dangerouslySetInnerHTML

`dangerouslySetInnerHTML` is React's replacement for using `innerHTML` in the browser DOM. In general, setting HTML from code is risky because it's easy to inadvertently expose your users to a [cross-site scripting \(XSS\)](#) attack. So, you can set HTML directly from React, but you have to type out `dangerouslySetInnerHTML` and pass an object with a `__html` key, to remind yourself that it's dangerous. For example:

```
function createMarkup() {
  return { __html: 'First &mdot; Second' };
}

function MyComponent() {
  return <div dangerouslySetInnerHTML={createMarkup()} />;
}
```

htmlFor

Since `for` is a reserved word in JavaScript, React elements use `htmlFor` instead.

onChange

The `onChange` event behaves as you would expect it to: whenever a form field is changed, this event is fired. We intentionally do not use the existing browser behavior because `onChange` is a misnomer for its behavior and React relies on this event to handle user input in real time.

selected

If you want to mark an `<option>` as selected, reference the value of that option in the `value` of its `<select>` instead. Check out [“The select Tag”](#) for detailed instructions.

style

Note

Some examples in the documentation use `style` for convenience, but **using the `style` attribute as the primary means of styling elements is generally not recommended**. In most cases, `className` should be used to reference classes defined in an external CSS stylesheet. `style` is most often used in React applications to add dynamically-computed styles at render time. See also [FAQ: Styling and CSS](#).

The `style` attribute accepts a JavaScript object with camelCased properties rather than a CSS string. This is consistent with the DOM `style` JavaScript property, is more efficient, and prevents XSS security holes. For example:

```
const divStyle = {
  color: 'blue',
  backgroundImage: 'url(' + imgUrl + ')'
};

function HelloWorldComponent() {
  return <div style={divStyle}>Hello World!</div>;
}
```

Note that styles are not autoprefixed. To support older browsers, you need to supply corresponding style properties:

```
const divStyle = {
  WebkitTransition: 'all', // note the capital 'W' here
  msTransition: 'all' // 'ms' is the only lowercase vendor prefix
};

function ComponentWithTransition() {
  return <div style={divStyle}>This should work cross-browser</div>;
}
```

Style keys are camelCased in order to be consistent with accessing the properties on DOM nodes from JS (e.g. `node.style.backgroundColor`). Vendor prefixes other than `ms` should begin with a capital letter. This is why `WebkitTransition` has an uppercase “W”.

React will automatically append a “px” suffix to certain numeric inline style properties. If you want to use units other than “px”, specify the value as a string with the desired unit. For example:

```
// Result style: '10px'
<div style={{ height: 10 }}>
  Hello World!
</div> // Result style: '10%'
<div style={{ height: '10%' }}>
  Hello World!
</div>
```

Not all style properties are converted to pixel strings though. Certain ones remain unitless (eg `zoom`, `order`, `flex`). A complete list of unitless properties can be seen [here](#).

suppressContentEditableWarning

Normally, there is a warning when an element with children is also marked as `contentEditable`, because it won't work. This attribute suppresses that warning. Don't use this unless you are building a library like [Draft.js](#) that manages `contentEditable` manually.

suppressHydrationWarning

If you use server-side React rendering, normally there is a warning when the server and the client render different content. However, in some rare cases, it is very hard or impossible to guarantee an exact match. For example, timestamps are expected to differ on the server and on the client.

If you set `suppressHydrationWarning` to `true`, React will not warn you about mismatches in the attributes and the content of that element. It only works one level deep, and is intended to be used as an escape hatch. Don't overuse it. You can read more about hydration in the [ReactDOM.hydrateRoot\(\). documentation](#).

value

The `value` attribute is supported by `<input>`, `<select>` and `<textarea>` components. You can use it to set the value of the component. This is useful for building controlled components. `defaultValue` is the uncontrolled equivalent, which sets the value of the component when it is first mounted.

All Supported HTML Attributes

As of React 16, any standard or custom DOM attributes are fully supported.

React has always provided a JavaScript-centric API to the DOM. Since React components often take both custom and DOM-related props, React uses the `camelCase` convention just like the DOM APIs:

```
<div tabIndex={-1} />      // Just like node.tabIndex DOM API
<div className="Button" /> // Just like node.className DOM API
<input readOnly={true} />  // Just like node.readOnly DOM API
```

These props work similarly to the corresponding HTML attributes, with the exception of the special cases documented above.

Some of the DOM attributes supported by React include:

```
accept acceptCharset accessKey action allowFullScreen alt async autoComplete
autoFocus autoPlay capture cellPadding cellSpacing challenge charSet checked
cite classID className colSpan cols content contentEditable contextMenu controls
controlsList coords crossOrigin data dateTime default defer dir disabled
download draggable encType form formAction formEncType formMethod formNoValidate
formTarget frameBorder headers height hidden high href hrefLang htmlFor
httpEquiv icon id inputMode integrity is keyParams keyType kind label lang list
loop low manifest marginHeight marginWidth max maxLength media mediaGroup method
min minLength multiple muted name noValidate nonce open optimum pattern
placeholder poster preload profile radioGroup readOnly rel required reversed
role rowSpan rows sandbox scope scoped scrolling seamless selected shape size
sizes span spellCheck src srcDoc srcLang srcSet start step style summary
tabIndex target title type useMap value width wmode wrap
```

Similarly, all SVG attributes are fully supported:

```
accentHeight accumulate additive alignmentBaseline allowReorder alphabetic
amplitude arabicForm ascent attributeName attributeType autoReverse azimuth
baseFrequency baseProfile baselineShift bbox begin bias by calcMode capHeight
clip clipPath clipPathUnits clipRule colorInterpolation
colorInterpolationFilters colorProfile colorRendering contentScriptType
contentStyleType cursor cx cy d decelerate descent diffuseConstant direction
display divisor dominantBaseline dur dx dy edgeMode elevation enableBackground
end exponent externalResourcesRequired fill fillOpacity fillRule filter
filterRes filterUnits floodColor floodOpacity focusable fontFamily fontSize
fontSizeAdjust fontStretch fontStyle fontVariant fontWeight format from fx fy
g1 g2 glyphName glyphOrientationHorizontal glyphOrientationVertical glyphRef
gradientTransform gradientUnits hanging horizAdvX horizOriginX ideographic
imageRendering in in2 intercept k k1 k2 k3 k4 kernelMatrix kernelUnitLength
kerning keyPoints keySplines keyTimes lengthAdjust letterSpacing lightingColor
limitingConeAngle local markerEnd markerHeight markerMid markerStart
markerUnits markerWidth mask maskContentUnits maskUnits mathematical mode
numOctaves offset opacity operator order orient orientation origin overflow
overlinePosition overlineThickness paintOrder panose1 pathLength
patternContentUnits patternTransform patternUnits pointerEvents points
pointsAtX pointsAtY pointsAtZ preserveAlpha preserveAspectRatio primitiveUnits
r radius refX refY renderingIntent repeatCount repeatDur requiredExtensions
requiredFeatures restart result rotate rx ry scale seed shapeRendering slope
spacing specularConstant specularExponent speed spreadMethod startOffset
stdDeviation stemh stemv stitchTiles stopColor stopOpacity
strikethroughPosition strikethroughThickness string stroke strokeDasharray
strokeDashoffset strokeLinecap strokeLinejoin strokeMiterlimit strokeOpacity
strokeWidth surfaceScale systemLanguage tableValues targetX targetY textAnchor
textDecoration textLength textRendering to transform u1 u2 underlinePosition
underlineThickness unicode unicodeBidi unicodeRange unitsPerEm vAlphabetic
vHanging vIdeographic vMathematical values vectorEffect version vertAdvY
vertOriginX vertOriginY viewBox viewTarget visibility widths wordSpacing
writingMode x x1 x2 xChannelSelector xHeight xlinkActuate xlinkArcrole
xlinkHref xlinkRole xlinkShow xlinkTitle xlinkType xmlns xmlnsXlink xmlBase
xmlLang xmlSpace y y1 y2 yChannelSelector z zoomAndPan
```

You may also use custom attributes as long as they're fully lowercase.

6. SyntheticEvent

This reference guide documents the `SyntheticEvent` wrapper that forms part of React's Event System. See the [Handling Events](#) guide to learn more.

Overview

Your event handlers will be passed instances of `SyntheticEvent`, a cross-browser wrapper around the browser's native event. It has the same interface as the browser's native event, including `stopPropagation()` and `preventDefault()`, except the events work identically across all browsers.

If you find that you need the underlying browser event for some reason, simply use the `nativeEvent` attribute to get it. The synthetic events are different from, and do not map directly to, the browser's native events. For example in `onMouseLeave` `event.nativeEvent` will point to a `mouseout` event. The specific mapping is not part of the public API and may change at any time. Every `SyntheticEvent` object has the following attributes:

```
boolean bubbles
boolean cancelable
DOMEventTarget currentTarget
boolean defaultPrevented
number eventPhase
boolean isTrusted
DOMEvent nativeEvent
void preventDefault()
boolean isDefaultPrevented()
void stopPropagation()
boolean isPropagationStopped()
void persist()
DOMEventTarget target
number timeStamp
string type
```

Note:

As of v17, `e.persist()` doesn't do anything because the `SyntheticEvent` is no longer pooled.

Note:

As of v0.14, returning `false` from an event handler will no longer stop event propagation.

Instead, `e.stopPropagation()` or `e.preventDefault()` should be triggered manually, as appropriate.

Supported Events

React normalizes events so that they have consistent properties across different browsers.

The event handlers below are triggered by an event in the bubbling phase. To register an event handler for the capture phase, append `Capture` to the event name; for example, instead of using `onClick`, you would use `onClickCapture` to handle the click event in the capture phase.

- [Clipboard Events](#)
- [Composition Events](#)
- [Keyboard Events](#)
- [Focus Events](#)
- [Form Events](#)
- [Generic Events](#)
- [Mouse Events](#)
- [Pointer Events](#)
- [Selection Events](#)
- [Touch Events](#)
- [UI Events](#)

- [Wheel Events](#)
 - [Media Events](#)
 - [Image Events](#)
 - [Animation Events](#)
 - [Transition Events](#)
 - [Other Events](#)
-

Reference

Clipboard Events

Event names:

`onCopy onCut onPaste`

Properties:

`DOMDataTransfer clipboardData`

Composition Events

Event names:

`onCompositionEnd onCompositionStart onCompositionUpdate`

Properties:

`string data`

Keyboard Events

Event names:

`onKeyDown onKeyPress onKeyUp`

Properties:

```
boolean altKey
number charCode
boolean ctrlKey
boolean getModifierState(key)
string key
number keyCode
string locale
number location
boolean metaKey
boolean repeat
boolean shiftKey
number which
```

The `key` property can take any of the values documented in the [DOM Level 3 Events spec](#).

Focus Events

Event names:

`onFocus onBlur`

These focus events work on all elements in the React DOM, not just form elements.

Properties:

`DOMEventTarget relatedTarget`

onFocus

The `onFocus` event is called when the element (or some element inside of it) receives focus. For example, it's called when the user clicks on a text input.

```
function Example() {
  return (
    <input
      onFocus={(e) => {
        console.log('Focused on input');
      }}
      placeholder="onFocus is triggered when you click this input."
    />
  );
}
```

onBlur

The `onBlur` event handler is called when focus has left the element (or left some element inside of it). For example, it's called when the user clicks outside of a focused text input.

```
function Example() {
  return (
    <input
      onBlur={(e) => {
        console.log('Triggered because this input lost focus');
      }}
      placeholder="onBlur is triggered when you click
        this input and then you click outside of it."
    />
  );
}
```

Detecting Focus Entering and Leaving

You can use the `currentTarget` and `relatedTarget` to differentiate if the focusing or blurring events originated from *outside* of the parent element. Here is a demo you can copy and paste that shows how to detect focusing a child, focusing the element itself, and focus entering or leaving the whole subtree.

```
function Example() {
  return (
    <div
      tabIndex={1}
      onFocus={(e) => {
        if (e.currentTarget === e.target) {
          console.log('focused self');
        } else {
          console.log('focused child', e.target);
        }
      }}
      if (!e.currentTarget.contains(e.relatedTarget)) {
        // Not triggered when swapping focus between children
        console.log('focus entered self');
      }
    >
    <input id="1" />
    <input id="2" />
  </div>
  );
}
```

Form Events

Event names:

`onChange` `onInput` `onInvalid` `onReset` `onSubmit`

For more information about the `onChange` event, see [Forms](#).

Generic Events

Event names:

`onError` `onLoad`

Mouse Events

Event names:

`onClick` `onContextMenu` `onDoubleClick` `onDrag` `onDragEnd` `onDragEnter` `onDragExit`
`onDragLeave` `onDragOver` `onDragStart` `onDrop` `onMouseDown` `onMouseEnter` `onMouseLeave`
`onMouseMove` `onMouseOut` `onMouseOver` `onMouseUp`

The `onMouseEnter` and `onMouseLeave` events propagate from the element being left to the one being entered instead of ordinary bubbling and do not have a capture phase.

Properties:

```
boolean altKey
number button
number buttons
number clientX
number clientY
boolean ctrlKey
boolean getModifierState(key)
boolean metaKey
number pageX
number pageY
DOMEventTarget relatedTarget
number screenX
number screenY
boolean shiftKey
```

Pointer Events

Event names:

`onPointerDown` `onPointerMove` `onPointerUp` `onPointerCancel` `onGotPointerCapture`
`onLostPointerCapture` `onPointerEnter` `onPointerLeave` `onPointerOver` `onPointerOut`

The `onPointerEnter` and `onPointerLeave` events propagate from the element being left to the one being entered instead of ordinary bubbling and do not have a capture phase.

Properties:

As defined in the [W3 spec](#), pointer events extend [Mouse Events](#) with the following properties:

```
number pointerId
number width
number height
number pressure
number tangentialPressure
number tiltX
number tiltY
number twist
string pointerType
boolean isPrimary
```

A note on cross-browser support:

Pointer events are not yet supported in every browser (at the time of writing this article, supported browsers include: Chrome, Firefox, Edge, and Internet Explorer). React deliberately does not polyfill support for other browsers because a standard-conform polyfill would significantly increase the bundle size of `react-dom`.

If your application requires pointer events, we recommend adding a third party pointer event polyfill.

Selection Events

Event names:

`onSelect`

Touch Events

Event names:

`onTouchCancel` `onTouchEnd` `onTouchMove` `onTouchStart`

Properties:

```
boolean altKey
DOMTouchList changedTouches
boolean ctrlKey
boolean getModifierState(key)
boolean metaKey
boolean shiftKey
DOMTouchList targetTouches
DOMTouchList touches
```

UI Events

Event names:

`onScroll`

Note

Starting with React 17, the `onScroll` event **does not bubble** in React. This matches the browser behavior and prevents the confusion when a nested scrollable element fires events on a distant parent.

Properties:

`number detail`
`DOMAbstractView view`

Wheel Events

Event names:

`onWheel`

Properties:

```
number deltaMode
number deltaX
number deltaY
number deltaZ
```

Media Events

Event names:

`onAbort` `onCanPlay` `onCanPlayThrough` `onDurationChange` `onEmptied` `onEncrypted`
`onEnded` `onError` `onLoadedData` `onLoadedMetadata` `onLoadStart` `onPause` `onPlay`
`onPlaying` `onProgress` `onRateChange` `onSeeked` `onSeeking` `onStalled` `onSuspend`
`onTimeUpdate` `onVolumeChange` `onWaiting`

Image Events

Event names:

`onLoad` `onError`

Animation Events

Event names:

`onAnimationStart` `onAnimationEnd` `onAnimationIteration`

Properties:

```
string animationName
string pseudoElement
float elapsedTime
```

Transition Events

Event names:

`onTransitionEnd`

Properties:

```
string propertyName
string pseudoElement
float elapsedTime
```

Other Events

Event names:

`onToggle`

7. Test Utilities

Importing

```
import ReactTestUtils from 'react-dom/test-utils'; // ES6
var ReactTestUtils = require('react-dom/test-utils'); // ES5 with npm
```

Overview

`ReactTestUtils` makes it easy to test React components in the testing framework of your choice. At Facebook we use [Jest](#) for painless JavaScript testing. Learn how to get started with Jest through the Jest website's [React Tutorial](#).

Note:

We recommend using [React Testing Library](#), which is designed to enable and encourage writing tests that use your components as the end users do.

For React versions ≤ 16 , the [Enzyme](#) library makes it easy to assert, manipulate, and traverse your React Components' output.

- `act()`
- `mockComponent()`

- `isElement()`
- `isElementOfType()`
- `isDOMComponent()`
- `isCompositeComponent()`
- `isCompositeComponentWithType()`
- `findAllInRenderedTree()`
- `scryRenderedDOMComponentsWithClass()`
- `findRenderedDOMComponentWithClass()`
- `scryRenderedDOMComponentsWithTag()`
- `findRenderedDOMComponentWithTag()`
- `scryRenderedComponentsWithType()`
- `findRenderedComponentWithType()`
- `renderIntoDocument()`
- `Simulate`

Reference

`act()`

To prepare a component for assertions, wrap the code rendering it and performing updates inside an `act()` call. This makes your test run closer to how React works in the browser.

Note

If you use `react-test-renderer`, it also provides an `act` export that behaves the same way.

For example, let's say we have this `Counter` component:

```
class Counter extends React.Component {
  constructor(props) {
    super(props);
    this.state = { count: 0 };
    this.handleClick = this.handleClick.bind(this);
  }
  componentDidMount() {
    document.title = `You clicked ${this.state.count} times`;
  }
  componentDidUpdate() {
    document.title = `You clicked ${this.state.count} times`;
  }
  handleClick() {
    this.setState((state) => ({
      count: state.count + 1
    }));
  }
  render() {
    return (
      <div>
        <p>You clicked {this.state.count} times</p>
        <button onClick={this.handleClick}>Click me</button>
      </div>
    );
  }
}
```

Here is how we can test it:

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import { act } from 'react-dom/test-utils';
import Counter from './Counter';
```

```

let container;

beforeEach(() => {
  container = document.createElement('div');
  document.body.appendChild(container);
});

afterEach(() => {
  document.body.removeChild(container);
  container = null;
});

it('can render and update a counter', () => {
  // Test first render and componentDidMount
  act(() => {
    ReactDOM.createRoot(container).render(<Counter />);
  });
  const button = container.querySelector('button');
  const label = container.querySelector('p');
  expect(label.textContent).toBe('You clicked 0 times');
  expect(document.title).toBe('You clicked 0 times');

  // Test second render and componentDidUpdate
  act(() => {
    button.dispatchEvent(new MouseEvent('click', { bubbles: true }));
  });
  expect(label.textContent).toBe('You clicked 1 times');
  expect(document.title).toBe('You clicked 1 times');
});

```

- Don't forget that dispatching DOM events only works when the DOM container is added to the `document`. You can use a library like [React Testing Library](#) to reduce the boilerplate code.
- The [recipes](#) document contains more details on how `act()` behaves, with examples and usage.

mockComponent()

```
mockComponent(componentClass, [mockTagName]);
```

Pass a mocked component module to this method to augment it with useful methods that allow it to be used as a dummy React component. Instead of rendering as usual, the component will become a simple `<div>` (or other tag if `mockTagName` is provided) containing any provided children.

Note:

`mockComponent()` is a legacy API. We recommend using `jest.mock()` instead.

isElement()

```
isElement(element)
```

Returns `true` if `element` is any React element.

isElementOfType()

```
isElementOfType(element, componentClass)
```

Returns `true` if `element` is a React element whose type is of a React `componentClass`.

isDOMComponent()

```
isDOMComponent(instance)
```

Returns `true` if `instance` is a DOM component (such as a `<div>` or ``).

isCompositeComponent()

```
isCompositeComponent(instance)
```

Returns `true` if `instance` is a user-defined component, such as a class or a function.

isCompositeComponentWithType()

`isCompositeComponentWithType(instance, componentClass)`

Returns `true` if `instance` is a component whose type is of a React `componentClass`.

findAllInRenderedTree()

`findAllInRenderedTree(tree, test)`

Traverse all components in `tree` and accumulate all components where `test(component)` is `true`. This is not that useful on its own, but it's used as a primitive for other test utils.

scryRenderedDOMComponentsWithClass()

`scryRenderedDOMComponentsWithClass(tree, className)`

Finds all DOM elements of components in the rendered tree that are DOM components with the class name matching `className`.

findRenderedDOMComponentWithClass()

`findRenderedDOMComponentWithClass(tree, className)`

Like `scryRenderedDOMComponentsWithClass()`, but expects there to be one result, and returns that one result, or throws exception if there is any other number of matches besides one.

scryRenderedDOMComponentsWithTag()

`scryRenderedDOMComponentsWithTag(tree, tagName)`

Finds all DOM elements of components in the rendered tree that are DOM components with the tag name matching `tagName`.

findRenderedDOMComponentWithTag()

`findRenderedDOMComponentWithTag(tree, tagName)`

Like `scryRenderedDOMComponentsWithTag()`, but expects there to be one result, and returns that one result, or throws exception if there is any other number of matches besides one.

scryRenderedComponentsWithType()

`scryRenderedComponentsWithType(tree, componentClass)`

Finds all instances of components with type equal to `componentClass`.

findRenderedComponentWithType()

`findRenderedComponentWithType(tree, componentClass)`

Same as `scryRenderedComponentsWithType()`, but expects there to be one result and returns that one result, or throws exception if there is any other number of matches besides one.

renderIntoDocument()

`renderIntoDocument(element)`

Render a React element into a detached DOM node in the document. **This function requires a DOM.** It is effectively equivalent to:

```
const domContainer = document.createElement('div');
ReactDOM.createRoot(domContainer).render(element);
```

Note:

You will need to have `window`, `window.document` and `window.document.createElement` globally available **before** you import `React`. Otherwise React will think it can't access the DOM and

methods like `setState` won't work.

Other Utilities

Simulate

```
Simulate.{eventName}(element, [eventData])
```

Simulate an event dispatch on a DOM node with optional `eventData` event data.

`Simulate` has a method for every event that React understands.

Clicking an element

```
// <button ref={(node) => this.button = node}>...</button>
const node = this.button;
ReactTestUtils.Simulate.click(node);
```

Changing the value of an input field and then pressing ENTER.

```
// <input ref={(node) => this.textInput = node} />
const node = this.textInput;
node.value = 'giraffe';
ReactTestUtils.Simulate.change(node);
ReactTestUtils.Simulate.keyDown(node, {key: "Enter", keyCode: 13, which: 13});
```

Note

You will have to provide any event property that you're using in your component (e.g. `keyCode`, `which`, etc...) as React is not creating any of these for you.

8. Test Renderer

Importing

```
import TestRenderer from 'react-test-renderer'; // ES6
const TestRenderer = require('react-test-renderer'); // ES5 with npm
```

Overview

This package provides a React renderer that can be used to render React components to pure JavaScript objects, without depending on the DOM or a native mobile environment.

Essentially, this package makes it easy to grab a snapshot of the platform view hierarchy (similar to a DOM tree) rendered by a React DOM or React Native component without using a browser or [jsdom](#).

Example:

```
import TestRenderer from 'react-test-renderer';

function Link(props) {
  return <a href={props.page}>{props.children}</a>;
}

const testRenderer = TestRenderer.create(
  <Link page="https://www.facebook.com/">Facebook</Link>;
);

console.log(testRenderer.toJSON());
// { type: 'a',
//   props: { href: 'https://www.facebook.com/' },
//   children: [ 'Facebook' ] }
```

You can use Jest's snapshot testing feature to automatically save a copy of the JSON tree to a file and check in your tests that it hasn't changed: [Learn more about it](#).

You can also traverse the output to find specific nodes and make assertions about them.

```
import TestRenderer from 'react-test-renderer';

function MyComponent() {
  return (
    <div>
      <SubComponent foo="bar" />
      <p className="my">Hello</p>
    </div>
  );
}

function SubComponent() {
  return <p className="sub">Sub</p>;
}

const testRenderer = TestRenderer.create(<MyComponent />);
const testInstance = testRenderer.root;

expect(testInstance.findByType(SubComponent).props.foo).toBe('bar');
expect(testInstance.findByProps({ className: 'sub' }).children).toEqual(['Sub']);
```

TestRenderer

- `TestRenderer.create()`
- `TestRenderer.act()`

TestRenderer instance

- `testRenderer.toJSON()`
- `testRenderer.toTree()`
- `testRenderer.update()`
- `testRenderer.unmount()`
- `testRenderer.getInstance()`
- `testRenderer.root`

TestInstance

- `testInstance.find()`
- `testInstance.findByType()`
- `testInstance.findByProps()`
- `testInstance.findAll()`
- `testInstance.findAllByType()`
- `testInstance.findAllByProps()`
- `testInstance.instance`
- `testInstance.type`
- `testInstance.props`
- `testInstance.parent`
- `testInstance.children`

Reference

TestRenderer.create()

```
TestRenderer.create(element, options);
```

Create a `TestRenderer` instance with the passed React element. It doesn't use the real DOM, but it still fully renders the component tree into memory so you can make assertions about it. Returns a TestRenderer instance.

TestRenderer.act()

```
TestRenderer.act(callback);
```

Similar to the `act()` helper from `react-dom/test-utils`, `TestRenderer.act` prepares a component for assertions. Use this version of `act()` to wrap calls to `TestRenderer.create` and `testRenderer.update`.

```
import { create, act } from 'react-test-renderer';
import App from './app.js'; // The component being tested

// render the component
let root;
act(() => {
  root = create(<App value={1} />);
});

// make assertions on root
expect(root.toJSON()).toMatchSnapshot();

// update with some different props
act(() => {
  root.update(<App value={2} />);
});

// make assertions on root
expect(root.toJSON()).toMatchSnapshot();
```

testRenderer.toJSON()

```
testRenderer.toJSON()
```

Return an object representing the rendered tree. This tree only contains the platform-specific nodes like `<div>` or `<View>` and their props, but doesn't contain any user-written components. This is handy for snapshot testing.

testRenderer.toTree()

```
testRenderer.toTree()
```

Return an object representing the rendered tree. The representation is more detailed than the one provided by `toJSON()`, and includes the user-written components. You probably don't need this method unless you're writing your own assertion library on top of the test renderer.

testRenderer.update()

```
testRenderer.update(element)
```

Re-render the in-memory tree with a new root element. This simulates a React update at the root. If the new element has the same type and key as the previous element, the tree will be updated; otherwise, it will re-mount a new tree.

testRenderer.unmount()

```
testRenderer.unmount()
```

Unmount the in-memory tree, triggering the appropriate lifecycle events.

testRenderer.getInstance()

```
testRenderer.getInstance()
```

Return the instance corresponding to the root element, if available. This will not work if the root element is a function component because they don't have instances.

testRenderer.root

```
testRenderer.root
```

Returns the root “test instance” object that is useful for making assertions about specific nodes in the tree. You can use it to find other “test instances” deeper below.

testInstance.find()

`testInstance.find(test)`

Find a single descendant test instance for which `test(testInstance)` returns `true`. If `test(testInstance)` does not return `true` for exactly one test instance, it will throw an error.

testInstance.findByType()

`testInstance.findByType(type)`

Find a single descendant test instance with the provided `type`. If there is not exactly one test instance with the provided `type`, it will throw an error.

testInstance.findByProps()

`testInstance.findByProps(props)`

Find a single descendant test instance with the provided `props`. If there is not exactly one test instance with the provided `props`, it will throw an error.

testInstance.findAll()

`testInstance.findAll(test)`

Find all descendant test instances for which `test(testInstance)` returns `true`.

testInstance.findAllByType()

`testInstance.findAllByType(type)`

Find all descendant test instances with the provided `type`.

testInstance.findAllByProps()

`testInstance.findAllByProps(props)`

Find all descendant test instances with the provided `props`.

testInstance.instance

`testInstance.instance`

The component instance corresponding to this test instance. It is only available for class components, as function components don't have instances. It matches the `this` value inside the given component.

testInstance.type

`testInstance.type`

The component type corresponding to this test instance. For example, a `<Button />` component has a type of `Button`.

testInstance.props

`testInstance.props`

The props corresponding to this test instance. For example, a `<Button size="small" />` component has `{size: 'small'}` as props.

testInstance.parent

`testInstance.parent`

The parent test instance of this test instance.

testInstance.children

`testInstance.children`

The children test instances of this test instance.

Ideas

You can pass `createNodeMock` function to `TestRenderer.create` as the option, which allows for custom mock refs. `createNodeMock` accepts the current element and should return a mock ref object. This is useful when you test a component that relies on refs.

```
import TestRenderer from 'react-test-renderer';

class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.input = null;
  }
  componentDidMount() {
    this.input.focus();
  }
  render() {
    return <input type="text" ref={el => (this.input = el)} />;
  }
}

let focused = false;
TestRenderer.create(<MyComponent />, {
  createNodeMock: (element) => {
    if (element.type === 'input') {
      // mock a focus function
      return {
        focus: () => {
          focused = true;
        }
      };
    }
    return null;
  }
});
expect(focused).toBe(true);
```

9. JavaScript Environment Requirements

React 18 supports all modern browsers (Edge, Firefox, Chrome, Safari, etc).

If you support older browsers and devices such as Internet Explorer which do not provide modern browser features natively or have non-compliant implementations, consider including a global polyfill in your bundled application.

Here is a list of the modern features React 18 uses:

- `Promise`
- `Symbol`
- `Object.assign`

The correct polyfill for these features depend on your environment. For many users, you can configure your `Browserslist` settings. For others, you may need to import polyfills like `core-js` directly.

10. Glossary of React Terms

Single-page Application

A single-page application is an application that loads a single HTML page and all the necessary assets (such as JavaScript and CSS) required for the application to run. Any interactions with the page or subsequent pages do not require a round trip to the server which means the page is not reloaded.

Though you may build a single-page application in React, it is not a requirement. React can also be used for enhancing small parts of existing websites with additional interactivity. Code written in React can coexist peacefully with markup rendered on the server by something like PHP, or with other client-side libraries. In fact, this is exactly how React is being used at Facebook.

ES6, ES2015, ES2016, etc

These acronyms all refer to the most recent versions of the ECMAScript Language Specification standard, which the JavaScript language is an implementation of. The ES6 version (also known as ES2015) includes many additions to the previous versions such as: arrow functions, classes, template literals, `let` and `const` statements. You can learn more about specific versions [here](#).

Compilers

A JavaScript compiler takes JavaScript code, transforms it and returns JavaScript code in a different format. The most common use case is to take ES6 syntax and transform it into syntax that older browsers are capable of interpreting. [Babel](#) is the compiler most commonly used with React.

Bundlers

Bundlers take JavaScript and CSS code written as separate modules (often hundreds of them), and combine them together into a few files better optimized for the browsers. Some bundlers commonly used in React applications include [Webpack](#) and [Browserify](#).

Package Managers

Package managers are tools that allow you to manage dependencies in your project. [npm](#) and [Yarn](#) are two package managers commonly used in React applications. Both of them are clients for the same npm package registry.

CDN

CDN stands for Content Delivery Network. CDNs deliver cached, static content from a network of servers across the globe.

JSX

JSX is a syntax extension to JavaScript. It is similar to a template language, but it has full power of JavaScript. JSX gets compiled to `React.createElement()` calls which return plain JavaScript objects called “React elements”. To get a basic introduction to JSX [see the docs here](#) and find a more in-depth tutorial on JSX [here](#).

React DOM uses camelCase property naming convention instead of HTML attribute names. For example, `tabindex` becomes `tabIndex` in JSX. The attribute `class` is also written as `className` since `class` is a reserved word in JavaScript:

```
<h1 className="hello">My name is Clementine!</h1>
```

Elements

React elements are the building blocks of React applications. One might confuse elements with a more widely known concept of “components”. An element describes what you want to see on the screen. React elements are immutable.

```
const element = <h1>Hello, world</h1>;
```

Typically, elements are not used directly, but get returned from components.

Components

React components are small, reusable pieces of code that return a React element to be rendered to the page. The simplest version of React component is a plain JavaScript function that returns a React element:

```
function Welcome(props) {  
  return <h1>Hello, {props.name}</h1>;  
}
```

Components can also be ES6 classes:

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

```
}  
}
```

Components can be broken down into distinct pieces of functionality and used within other components. Components can return other components, arrays, strings and numbers. A good rule of thumb is that if a part of your UI is used several times (Button, Panel, Avatar), or is complex enough on its own (App, FeedStory, Comment), it is a good candidate to be a reusable component. Component names should also always start with a capital letter (`<Wrapper/>` **not** `<wrapper/>`). See [this documentation](#) for more information on rendering components.

props

`props` are inputs to a React component. They are data passed down from a parent component to a child component.

Remember that `props` are readonly. They should not be modified in any way:

```
// Wrong!  
props.number = 42;
```

If you need to modify some value in response to user input or a network response, use `state` instead.

props.children

`props.children` is available on every component. It contains the content between the opening and closing tags of a component. For example:

```
<Welcome>Hello world!</Welcome>
```

The string `Hello world!` is available in `props.children` in the `Welcome` component:

```
function Welcome(props) {  
  return <p>{props.children}</p>;  
}
```

For components defined as classes, use `this.props.children`:

```
class Welcome extends React.Component {  
  render() {  
    return <p>{this.props.children}</p>;  
  }  
}
```

state

A component needs `state` when some data associated with it changes over time. For example, a `Checkbox` component might need `isChecked` in its state, and a `NewsFeed` component might want to keep track of `fetchPosts` in its state.

The most important difference between `state` and `props` is that `props` are passed from a parent component, but `state` is managed by the component itself. A component cannot change its `props`, but it can change its `state`.

For each particular piece of changing data, there should be just one component that “owns” it in its state. Don’t try to synchronize states of two different components. Instead, [lift it up](#) to their closest shared ancestor, and pass it down as props to both of them.

Lifecycle Methods

Lifecycle methods are custom functionality that gets executed during the different phases of a component. There are methods available when the component gets created and inserted into the DOM ([mounting](#)), when the component updates, and when the component gets unmounted or removed from the DOM.

Controlled vs. Uncontrolled Components

React has two different approaches to dealing with form inputs.

An input form element whose value is controlled by React is called a *controlled component*. When a user enters data into a controlled component a change event handler is triggered and your code decides whether the input is valid (by re-rendering with the updated value). If you do not re-render then the form element will remain unchanged.

An *uncontrolled component* works like form elements do outside of React. When a user inputs data into a form field (an input box, dropdown, etc) the updated information is reflected without React needing to do anything. However, this also means that you can't force the field to have a certain value.

In most cases you should use controlled components.

Keys

A “key” is a special string attribute you need to include when creating arrays of elements. Keys help React identify which items have changed, are added, or are removed. Keys should be given to the elements inside an array to give the elements a stable identity.

Keys only need to be unique among sibling elements in the same array. They don't need to be unique across the whole application or even a single component.

Don't pass something like `Math.random()` to keys. It is important that keys have a “stable identity” across re-renders so that React can determine when items are added, removed, or re-ordered. Ideally, keys should correspond to unique and stable identifiers coming from your data, such as `post.id`.

Refs

React supports a special attribute that you can attach to any component. The `ref` attribute can be an object created by `React.createRef()`, function or a callback function, or a string (in legacy API). When the `ref` attribute is a callback function, the function receives the underlying DOM element or class instance (depending on the type of element) as its argument. This allows you to have direct access to the DOM element or component instance.

Use refs sparingly. If you find yourself often using refs to “make things happen” in your app, consider getting more familiar with top-down data flow.

Events

Handling events with React elements has some syntactic differences:

- React event handlers are named using camelCase, rather than lowercase.
- With JSX you pass a function as the event handler, rather than a string.

Reconciliation

When a component's props or state change, React decides whether an actual DOM update is necessary by comparing the newly returned element with the previously rendered one. When they are not equal, React will update the DOM. This process is called “reconciliation”.

IV. HOOKS

1. Introducing Hooks

Hooks are a new addition in React 16.8. They let you use state and other React features without writing a class.

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

function Example() {
  // Declare a new state variable, which we'll call "count"
  const [count, setCount] = useState(0);
  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}> Click me</button>
    </div>
  );
}
```

This new function `useState` is the first “Hook” we’ll learn about, but this example is just a teaser. Don’t worry if it doesn’t make sense yet!

You can start learning Hooks on the next page. On this page, we’ll continue by explaining why we’re adding Hooks to React and how they can help you write great applications.

Note

React 16.8.0 is the first release to support Hooks. When upgrading, don’t forget to update all packages, including React DOM. React Native has supported Hooks since the 0.59 release of React Native.

Video Introduction

At React Conf 2018, Sophie Alpert and Dan Abramov introduced Hooks, followed by Ryan Florence demonstrating how to refactor an application to use them. Watch the video here:

No Breaking Changes

Before we continue, note that Hooks are:

- **Completely opt-in.** You can try Hooks in a few components without rewriting any existing code. But you don’t have to learn or use Hooks right now if you don’t want to.
- **100% backwards-compatible.** Hooks don’t contain any breaking changes.
- **Available now.** Hooks are now available with the release of v16.8.0.

There are no plans to remove classes from React. You can read more about the gradual adoption strategy for Hooks in the bottom section of this page.

Hooks don’t replace your knowledge of React concepts. Instead, Hooks provide a more direct API to the React concepts you already know: props, state, context, refs, and lifecycle. As we will show later, Hooks also offer a new powerful way to combine them.

If you just want to start learning Hooks, feel free to jump directly to the next page! You can also keep reading this page to learn more about why we’re adding Hooks, and how we’re going to start using them without rewriting our applications.

Motivation

Hooks solve a wide variety of seemingly unconnected problems in React that we’ve encountered over five years of writing and maintaining tens of thousands of components. Whether you’re learning React, use it daily, or even prefer a different library with a similar component model, you might recognize some of these problems.

It’s hard to reuse stateful logic between components

React doesn’t offer a way to “attach” reusable behavior to a component (for example, connecting it to a store). If you’ve worked with React for a while, you may be familiar with patterns like render props and higher-order components that try to solve this. But these patterns require you to restructure your components when you use them, which can be cumbersome and make code harder to follow. If you look at a typical React application in React DevTools, you will likely find a “wrapper hell” of components surrounded by layers of providers, consumers, higher-order components, render props, and other abstractions. While we could filter them out in DevTools, this points to a deeper underlying problem: React needs a better primitive for sharing stateful logic.

With Hooks, you can extract stateful logic from a component so it can be tested independently and reused. **Hooks allow you to reuse stateful logic without changing your component hierarchy.** This makes it easy to share Hooks among many components or with the community.

We’ll discuss this more in Building Your Own Hooks.

Complex components become hard to understand

We've often had to maintain components that started out simple but grew into an unmanageable mess of stateful logic and side effects. Each lifecycle method often contains a mix of unrelated logic. For example, components might perform some data fetching in `componentDidMount` and `componentDidUpdate`. However, the same `componentDidMount` method might also contain some unrelated logic that sets up event listeners, with cleanup performed in `componentWillUnmount`. Mutually related code that changes together gets split apart, but completely unrelated code ends up combined in a single method. This makes it too easy to introduce bugs and inconsistencies.

In many cases it's not possible to break these components into smaller ones because the stateful logic is all over the place. It's also difficult to test them. This is one of the reasons many people prefer to combine React with a separate state management library. However, that often introduces too much abstraction, requires you to jump between different files, and makes reusing components more difficult.

To solve this, **Hooks let you split one component into smaller functions based on what pieces are related (such as setting up a subscription or fetching data)**, rather than forcing a split based on lifecycle methods. You may also opt into managing the component's local state with a reducer to make it more predictable.

We'll discuss this more in [Using the Effect Hook](#).

Classes confuse both people and machines

In addition to making code reuse and code organization more difficult, we've found that classes can be a large barrier to learning React. You have to understand how `this` works in JavaScript, which is very different from how it works in most languages. You have to remember to bind the event handlers. Without [ES2022 public class fields](#), the code is very verbose. People can understand props, state, and top-down data flow perfectly well but still struggle with classes. The distinction between function and class components in React and when to use each one leads to disagreements even between experienced React developers.

Additionally, React has been out for about five years, and we want to make sure it stays relevant in the next five years. As [Svelte](#), [Angular](#), [Glimmer](#), and others show, [ahead-of-time compilation](#) of components has a lot of future potential. Especially if it's not limited to templates. Recently, we've been experimenting with [component folding](#) using [Prepack](#), and we've seen promising early results. However, we found that class components can encourage unintentional patterns that make these optimizations fall back to a slower path. Classes present issues for today's tools, too. For example, classes don't minify very well, and they make hot reloading flaky and unreliable. We want to present an API that makes it more likely for code to stay on the optimizable path.

To solve these problems, **Hooks let you use more of React's features without classes**. Conceptually, React components have always been closer to functions. Hooks embrace functions, but without sacrificing the practical spirit of React. Hooks provide access to imperative escape hatches and don't require you to learn complex functional or reactive programming techniques.

Examples

[Hooks at a Glance](#) is a good place to start learning Hooks.

Gradual Adoption Strategy

TLDR: There are no plans to remove classes from React.

We know that React developers are focused on shipping products and don't have time to look into every new API that's being released. Hooks are very new, and it might be better to wait for more examples and tutorials before considering learning or adopting them.

We also understand that the bar for adding a new primitive to React is extremely high. For curious readers, we have prepared a [detailed RFC](#) that dives into the motivation with more details, and provides extra perspective on the specific design decisions and related prior art.

Crucially, Hooks work side-by-side with existing code so you can adopt them gradually. There is no rush to migrate to Hooks. We recommend avoiding any "big rewrites", especially for existing, complex class components. It takes a bit of a mind shift to start "thinking in Hooks". In our experience, it's best to practice using Hooks in new and non-critical components first, and ensure that everybody on your team feels comfortable with them. After you give Hooks a try, please feel free to [send us feedback](#), positive or negative.

We intend for Hooks to cover all existing use cases for classes, but **we will keep supporting class components for the foreseeable future**. At Facebook, we have tens of thousands of components written as classes, and we have absolutely no plans to rewrite them. Instead, we are starting to use Hooks in the new code side by side with classes.

Frequently Asked Questions

We've prepared a [Hooks FAQ page](#) that answers the most common questions about Hooks.

Next Steps

By the end of this page, you should have a rough idea of what problems Hooks are solving, but many details are probably unclear. Don't worry! **Let's now go to [the next page](#) where we start learning about Hooks by example.**

2. Hooks at a Glance

Hooks are a new addition in React 16.8. They let you use state and other React features without writing a class.

Hooks are [backwards-compatible](#). This page provides an overview of Hooks for experienced React users. This is a fast-paced overview. If you get confused, look for a yellow box like this:

Detailed Explanation

Read the [Motivation](#) to learn why we're introducing Hooks to React.

↑↑↑ **Each section ends with a yellow box like this.** They link to detailed explanations.

State Hook

This example renders a counter. When you click the button, it increments the value:

```
import React, { useState } from 'react';
function Example() {
  // Declare a new state variable, which we'll call "count"
  const [count, setCount] = useState(0);
  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}> Click me</button>
    </div>
  );
}
```

Here, `useState` is a *Hook* (we'll talk about what this means in a moment). We call it inside a function component to add some local state to it. React will preserve this state between re-renders. `useState` returns a pair: the *current* state value and a function that lets you update it. You can call this function from an event handler or somewhere else. It's similar to `this.setState` in a class, except it doesn't merge the old and new state together. (We'll show an example comparing `useState` to `this.state` in [Using the State Hook](#).)

The only argument to `useState` is the initial state. In the example above, it is `0` because our counter starts from zero. Note that unlike `this.state`, the state here doesn't have to be an object — although it can be if you want. The initial state argument is only used during the first render.

Declaring multiple state variables

You can use the State Hook more than once in a single component:

```
function ExampleWithManyStates() {
  // Declare multiple state variables!
  const [age, setAge] = useState(42);
  const [fruit, setFruit] = useState('banana');
  const [todos, setTodos] = useState([{ text: 'Learn Hooks' }]);
  // ...
}
```

The [array destructuring](#) syntax lets us give different names to the state variables we declared by calling `useState`. These names aren't a part of the `useState` API. Instead, React assumes that if you call `useState` many times, you do it in the same order during every render. We'll come back to why this works and when this is useful later.

But what is a Hook?

Hooks are functions that let you “hook into” React state and lifecycle features from function components. Hooks don't work inside classes — they let you use React without classes. (We [don't recommend](#) rewriting your existing components overnight but you can start using Hooks in the new ones if you'd like.)

React provides a few built-in Hooks like `useState`. You can also create your own Hooks to reuse stateful behavior between different components. We'll look at the built-in Hooks first.

Detailed Explanation

You can learn more about the State Hook on a dedicated page: [Using the State Hook](#).

Effect Hook

You've likely performed data fetching, subscriptions, or manually changing the DOM from React components before. We call these operations “side effects” (or “effects” for short) because they can affect other components and can't be done during rendering.

The Effect Hook, `useEffect`, adds the ability to perform side effects from a function component. It serves the same purpose as `componentDidMount`, `componentDidUpdate`, and `componentWillUnmount` in React classes, but unified into a single API. (We'll show examples comparing `useEffect` to these methods in [Using the Effect Hook](#).)

For example, this component sets the document title after React updates the DOM:

```
import React, { useState, useEffect } from 'react';
function Example() {
  const [count, setCount] = useState(0);

  // Similar to componentDidMount and componentDidUpdate:
  useEffect(() => {
    // Update the document title using the browser API
    document.title = `You clicked ${count} times`;
  });
  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}> Click me</button>
    </div>
  );
}
```

When you call `useEffect`, you're telling React to run your “effect” function after flushing changes to the DOM. Effects are declared inside the component so they have access to its props and state. By default, React runs the effects after every render — *including* the first render. (We'll talk more about how this compares to class lifecycles in [Using the Effect Hook](#).)

Effects may also optionally specify how to “clean up” after them by returning a function. For example, this component uses an effect to subscribe to a friend's online status, and cleans up by unsubscribing from it:

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

function FriendStatus(props) {
  const [isOnline, setIsOnline] = useState(null);

  function handleStatusChange(status) {
    setIsOnline(status.isOnline);
  }

  useEffect(() => {
    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  });
  if (isOnline === null) {
    return 'Loading...';
  }
}
```

```

    return isOnline ? 'Online' : 'Offline';
  }

```

In this example, React would unsubscribe from our `ChatAPI` when the component unmounts, as well as before re-running the effect due to a subsequent render. (If you want, there's a way to [tell React to skip re-subscribing](#) if the `props.friend.id` we passed to `ChatAPI` didn't change.)

Just like with `useState`, you can use more than a single effect in a component:

```

function FriendStatusWithCounter(props) {
  const [count, setCount] = useState(0);
  useEffect(() => {
    document.title = `You clicked ${count} times`;
  });

  const [isOnline, setIsOnline] = useState(null);
  useEffect(() => {
    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  });

  function handleStatusChange(status) {
    setIsOnline(status.isOnline);
  }
  // ...
}

```

Hooks let you organize side effects in a component by what pieces are related (such as adding and removing a subscription), rather than forcing a split based on lifecycle methods.

Detailed Explanation

You can learn more about `useEffect` on a dedicated page: [Using the Effect Hook](#).

👉 Rules of Hooks

Hooks are JavaScript functions, but they impose two additional rules:

- Only call Hooks **at the top level**. Don't call Hooks inside loops, conditions, or nested functions.
- Only call Hooks **from React function components**. Don't call Hooks from regular JavaScript functions. (There is just one other valid place to call Hooks — your own custom Hooks. We'll learn about them in a moment.)

We provide a [linter plugin](#) to enforce these rules automatically. We understand these rules might seem limiting or confusing at first, but they are essential to making Hooks work well.

Detailed Explanation

You can learn more about these rules on a dedicated page: [Rules of Hooks](#).

💡 Building Your Own Hooks

Sometimes, we want to reuse some stateful logic between components. Traditionally, there were two popular solutions to this problem: [higher-order components](#) and [render props](#). Custom Hooks let you do this, but without adding more components to your tree.

Earlier on this page, we introduced a `FriendStatus` component that calls the `useState` and `useEffect` Hooks to subscribe to a friend's online status. Let's say we also want to reuse this subscription logic in another component.

First, we'll extract this logic into a custom Hook called `useFriendStatus`:

```

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

function useFriendStatus(friendID) {
  const [isOnline, setIsOnline] = useState(null);

  function handleStatusChange(status) {

```



```

    setIsOnline(status.isOnline);
  }

  useEffect(() => {
    ChatAPI.subscribeToFriendStatus(friendID, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(friendID, handleStatusChange);
    };
  });

  return isOnline;
}

```

It takes `friendID` as an argument, and returns whether our friend is online.

Now we can use it from both components:

```

function FriendStatus(props) {
  const isOnline = useFriendStatus(props.friend.id);
  if (isOnline === null) {
    return 'Loading...';
  }
  return isOnline ? 'Online' : 'Offline';
}

function FriendListItem(props) {
  const isOnline = useFriendStatus(props.friend.id);
  return <li style={{ color: isOnline ? 'green' : 'black' }}> {props.friend.name}</li>;
}

```

The state of each component is completely independent. Hooks are a way to reuse *stateful logic*, not state itself. In fact, each *call* to a Hook has a completely isolated state — so you can even use the same custom Hook twice in one component.

Custom Hooks are more of a convention than a feature. If a function's name starts with "`use`" and it calls other Hooks, we say it is a custom Hook. The `useSomething` naming convention is how our linter plugin is able to find bugs in the code using Hooks.

You can write custom Hooks that cover a wide range of use cases like form handling, animation, declarative subscriptions, timers, and probably many more we haven't considered. We are excited to see what custom Hooks the React community will come up with.

Detailed Explanation

You can learn more about custom Hooks on a dedicated page: [Building Your Own Hooks](#).



Other Hooks

There are a few less commonly used built-in Hooks that you might find useful. For example, `useContext` lets you subscribe to React context without introducing nesting:

```

function Example() {
  const locale = useContext(LocaleContext);
  const theme = useContext(ThemeContext); // ...
}

```

And `useReducer` lets you manage local state of complex components with a reducer:

```

function Todos() {
  const [todos, dispatch] = useReducer(todosReducer); // ...
}

```

Detailed Explanation

You can learn more about all the built-in Hooks on a dedicated page: [Hooks API Reference](#).

Next Steps

Phew, that was fast! If some things didn't quite make sense or you'd like to learn more in detail, you can read the next pages, starting with the [State Hook](#) documentation.

You can also check out the [Hooks API reference](#) and the [Hooks FAQ](#).

Finally, don't miss the [introduction page](#) which explains *why* we're adding Hooks and how we'll start using them side by side with classes — without rewriting our apps.

3. Using the State Hook

Hooks are a new addition in React 16.8. They let you use state and other React features without writing a class.

The [introduction page](#) used this example to get familiar with Hooks:

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

function Example() {
  // Declare a new state variable, which we'll call "count"
  const [count, setCount] = useState(0);
  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}> Click me</button>
    </div>
  );
}
```

We'll start learning about Hooks by comparing this code to an equivalent class example.

Equivalent Class Example

If you used classes in React before, this code should look familiar:

```
class Example extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
  }

  render() {
    return (
      <div>
        <p>You clicked {this.state.count} times</p>
        <button onClick={() => this.setState({ count: this.state.count + 1 })}>
          Click me
        </button>
      </div>
    );
  }
}
```

The state starts as `{ count: 0 }`, and we increment `state.count` when the user clicks a button by calling `this.setState()`. We'll use snippets from this class throughout the page.

Note

You might be wondering why we're using a counter here instead of a more realistic example. This is to help us focus on the API while we're still making our first steps with Hooks.

Hooks and Function Components

As a reminder, function components in React look like this:

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

or this:

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

You might have previously known these as “stateless components”. We’re now introducing the ability to use React state from these, so we prefer the name “function components”.

Hooks **don’t** work inside classes. But you can use them instead of writing classes.

What’s a Hook?

Our new example starts by importing the `useState` Hook from React:

```
import React, { useState } from 'react';
function Example() {
  // ...
}
```

What is a Hook? A Hook is a special function that lets you “hook into” React features. For example, `useState` is a Hook that lets you add React state to function components. We’ll learn other Hooks later.

When would I use a Hook? If you write a function component and realize you need to add some state to it, previously you had to convert it to a class. Now you can use a Hook inside the existing function component. We’re going to do that right now!

Note:

There are some special rules about where you can and can’t use Hooks within a component. We’ll learn them in [Rules of Hooks](#).

Declaring a State Variable

In a class, we initialize the `count` state to `0` by setting `this.state` to `{ count: 0 }` in the constructor:

```
class Example extends React.Component {
  constructor(props) {
    super(props);
    this.state = { count: 0 };
  }
}
```

In a function component, we have no `this`, so we can’t assign or read `this.state`. Instead, we call the `useState` Hook directly inside our component:

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

function Example() {
  // Declare a new state variable, which we'll call "count"
  const [count, setCount] = useState(0);
}
```

What does calling `useState` do? It declares a “state variable”. Our variable is called `count` but we could call it anything else, like `banana`. This is a way to “preserve” some values between the function calls — `useState` is a new way to use the exact same capabilities that `this.state` provides in a class. Normally, variables “disappear” when the function exits but state variables are preserved by React.

What do we pass to `useState` as an argument? The only argument to the `useState()` Hook is the initial state. Unlike with classes, the state doesn’t have to be an object. We can keep a number or a string if that’s all we need. In our example, we just want a number for how many times the user clicked, so pass `0` as initial state for our variable. (If we wanted to store two different values in state, we would call `useState()` twice.)

What does `useState` return? It returns a pair of values: the current state and a function that updates it. This is why we write `const [count, setCount] = useState()`. This is similar to `this.state.count` and `this.setState` in a class, except you get them in a pair. If you're not familiar with the syntax we used, we'll come back to it [at the bottom of this page](#).

Now that we know what the `useState` Hook does, our example should make more sense:

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

function Example() {
  // Declare a new state variable, which we'll call "count"
  const [count, setCount] = useState(0);
}
```

We declare a state variable called `count`, and set it to `0`. React will remember its current value between re-renders, and provide the most recent one to our function. If we want to update the current `count`, we can call `setCount`.

Note

You might be wondering: why is `useState` not named `createState` instead?

“Create” wouldn’t be quite accurate because the state is only created the first time our component renders. During the next renders, `useState` gives us the current state. Otherwise it wouldn’t be “state” at all! There’s also a reason why Hook names *always* start with `use`. We’ll learn why later in the [Rules of Hooks](#).

Reading State

When we want to display the current count in a class, we read `this.state.count`:

```
<p>You clicked {this.state.count} times</p>
```

In a function, we can use `count` directly:

```
<p>You clicked {count} times</p>
```

Updating State

In a class, we need to call `this.setState()` to update the `count` state:

```
<button onClick={() => this.setState({ count: this.state.count + 1 })}>
  Click me
</button>
```

In a function, we already have `setCount` and `count` as variables so we don't need `this`:

```
<button onClick={() => setCount(count + 1)}>
  Click me
</button>
```

Recap

Let's now **recap what we learned line by line** and check our understanding.

- **Line 1:** We import the `useState` Hook from React. It lets us keep local state in a function component.
- **Line 4:** Inside the `Example` component, we declare a new state variable by calling the `useState` Hook. It returns a pair of values, to which we give names. We're calling our variable `count` because it holds the number of button clicks. We initialize it to zero by passing `0` as the only `useState` argument. The second returned item is itself a function. It lets us update the `count` so we'll name it `setCount`.
- **Line 9:** When the user clicks, we call `setCount` with a new value. React will then re-render the `Example` component, passing the new `count` value to it.

This might seem like a lot to take in at first. Don't rush it! If you're lost in the explanation, look at the code above again and try to read it from top to bottom. We promise that once you try to “forget” how state works in classes, and look at this code with fresh eyes, it will make sense.

Tip: What Do Square Brackets Mean?

You might have noticed the square brackets when we declare a state variable:

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

The names on the left aren't a part of the React API. You can name your own state variables:

```
const [fruit, setFruit] = useState('banana');
```

This JavaScript syntax is called “[array destructuring](#)”. It means that we're making two new variables `fruit` and `setFruit`, where `fruit` is set to the first value returned by `useState`, and `setFruit` is the second. It is equivalent to this code:

```
var fruitStateVariable = useState('banana'); // Returns a pair
var fruit = fruitStateVariable[0]; // First item in a pair
var setFruit = fruitStateVariable[1]; // Second item in a pair
```

When we declare a state variable with `useState`, it returns a pair — an array with two items. The first item is the current value, and the second is a function that lets us update it. Using `[0]` and `[1]` to access them is a bit confusing because they have a specific meaning. This is why we use array destructuring instead.

Note

You might be curious how React knows which component `useState` corresponds to since we're not passing anything like `this` back to React. We'll answer [this question](#) and many others in the FAQ section.

Tip: Using Multiple State Variables

Declaring state variables as a pair of `[something, setSomething]` is also handy because it lets us give *different* names to different state variables if we want to use more than one:

```
function ExampleWithManyStates() {
  // Declare multiple state variables!
  const [age, setAge] = useState(42);
  const [fruit, setFruit] = useState('banana');
  const [todos, setTodos] = useState([{ text: 'Learn Hooks' }]);
}
```

In the above component, we have `age`, `fruit`, and `todos` as local variables, and we can update them individually:

```
function handleOrangeClick() {
  // Similar to this.setState({ fruit: 'orange' })
  setFruit('orange');
}
```

You **don't have to** use many state variables. State variables can hold objects and arrays just fine, so you can still group related data together. However, unlike `this.setState` in a class, updating a state variable always *replaces* it instead of merging it.

We provide more recommendations on splitting independent state variables [in the FAQ](#).

Next Steps

On this page we've learned about one of the Hooks provided by React, called `useState`. We're also sometimes going to refer to it as the “State Hook”. It lets us add local state to React function components — which we did for the first time ever!

We also learned a little bit more about what Hooks are. Hooks are functions that let you “hook into” React features from function components. Their names always start with `use`, and there are more Hooks we haven't seen yet.

Now let's continue by **learning the next Hook: `useEffect`**. It lets you perform side effects in components, and is similar to lifecycle methods in classes.

4. Using the Effect Hook

Hooks are a new addition in React 16.8. They let you use state and other React features without writing a class.

The *Effect Hook* lets you perform side effects in function components:

```
import React, { useState, useEffect } from 'react';
function Example() {
  const [count, setCount] = useState(0);

  // Similar to componentDidMount and componentDidUpdate:
  useEffect(() => {
    // Update the document title using the browser API
    document.title = `You clicked ${count} times`;
  });
  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}> Click me</button>
    </div>
  );
}
```

This snippet is based on the [counter example from the previous page](#), but we added a new feature to it: we set the document title to a custom message including the number of clicks.

Data fetching, setting up a subscription, and manually changing the DOM in React components are all examples of side effects. Whether or not you're used to calling these operations "side effects" (or just "effects"), you've likely performed them in your components before.

Tip

If you're familiar with React class lifecycle methods, you can think of `useEffect` Hook as `componentDidMount`, `componentDidUpdate`, and `componentWillUnmount` combined.

There are two common kinds of side effects in React components: those that don't require cleanup, and those that do. Let's look at this distinction in more detail.

Effects Without Cleanup

Sometimes, we want to **run some additional code after React has updated the DOM**. Network requests, manual DOM mutations, and logging are common examples of effects that don't require a cleanup. We say that because we can run them and immediately forget about them. Let's compare how classes and Hooks let us express such side effects.

Example Using Classes

In React class components, the `render` method itself shouldn't cause side effects. It would be too early — we typically want to perform our effects *after* React has updated the DOM.

This is why in React classes, we put side effects into `componentDidMount` and `componentDidUpdate`. Coming back to our example, here is a React counter class component that updates the document title right after React makes changes to the DOM:

```
class Example extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
  }

  componentDidMount() {
    document.title = `You clicked ${this.state.count} times`;
  }
  componentDidUpdate() {
    document.title = `You clicked ${this.state.count} times`;
  }
}
```

```

    }
    render() {
      return (
        <div>
          <p>You clicked {this.state.count} times</p>
          <button onClick={() => this.setState({ count: this.state.count + 1 })}>
            Click me
          </button>
        </div>
      );
    }
  }
}

```

Note how **we have to duplicate the code between these two lifecycle methods in class**.

This is because in many cases we want to perform the same side effect regardless of whether the component just mounted, or if it has been updated. Conceptually, we want it to happen after every render — but React class components don't have a method like this. We could extract a separate method but we would still have to call it in two places.

Now let's see how we can do the same with the `useEffect` Hook.

Example Using Hooks

We've already seen this example at the top of this page, but let's take a closer look at it:

```

import React, { useState, useEffect } from 'react';
function Example() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    document.title = `You clicked ${count} times`;
  });
  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}> Click me</button>
    </div>
  );
}

```

What does `useEffect` do? By using this Hook, you tell React that your component needs to do something after render. React will remember the function you passed (we'll refer to it as our “effect”), and call it later after performing the DOM updates. In this effect, we set the document title, but we could also perform data fetching or call some other imperative API.

Why is `useEffect` called inside a component? Placing `useEffect` inside the component lets us access the `count` state variable (or any props) right from the effect. We don't need a special API to read it — it's already in the function scope. Hooks embrace JavaScript closures and avoid introducing React-specific APIs where JavaScript already provides a solution.

Does `useEffect` run after every render? Yes! By default, it runs both after the first render *and* after every update. (We will later talk about [how to customize this](#).) Instead of thinking in terms of “mounting” and “updating”, you might find it easier to think that effects happen “after render”. React guarantees the DOM has been updated by the time it runs the effects.

Detailed Explanation

Now that we know more about effects, these lines should make sense:

```

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

  useEffect(() => {
    document.title = `You clicked ${count} times`;
  });
}

```

We declare the `count` state variable, and then we tell React we need to use an effect. We pass a function to the `useEffect` Hook. This function we pass *is* our effect. Inside our effect, we set the document title using the `document.title` browser API. We can read the latest `count` inside the effect because it's in the scope of our function. When React renders our component, it will remember the effect we used, and then run our effect after updating the DOM. This happens for every render, including the first one.

Experienced JavaScript developers might notice that the function passed to `useEffect` is going to be different on every render. This is intentional. In fact, this is what lets us read the `count` value from inside the effect without worrying about it getting stale. Every time we re-render, we schedule a *different* effect, replacing the previous one. In a way, this makes the effects behave more like a part of the render result — each effect “belongs” to a particular render. We will see more clearly why this is useful [later on this page](#).

Tip

Unlike `componentDidMount` or `componentDidUpdate`, effects scheduled with `useEffect` don't block the browser from updating the screen. This makes your app feel more responsive. The majority of effects don't need to happen synchronously. In the uncommon cases where they do (such as measuring the layout), there is a separate `useLayoutEffect` Hook with an API identical to `useEffect`.

Effects with Cleanup

Earlier, we looked at how to express side effects that don't require any cleanup. However, some effects do. For example, **we might want to set up a subscription** to some external data source. In that case, it is important to clean up so that we don't introduce a memory leak! Let's compare how we can do it with classes and with Hooks.

Example Using Classes

In a React class, you would typically set up a subscription in `componentDidMount`, and clean it up in `componentWillUnmount`. For example, let's say we have a `ChatAPI` module that lets us subscribe to a friend's online status. Here's how we might subscribe and display that status using a class:

```
class FriendStatus extends React.Component {
  constructor(props) {
    super(props);
    this.state = { isOnline: null };
    this.handleStatusChange = this.handleStatusChange.bind(this);
  }

  componentDidMount() {
    ChatAPI.subscribeToFriendStatus(this.props.friend.id, this.handleStatusChange);
  }

  componentWillUnmount() {
    ChatAPI.unsubscribeFromFriendStatus(this.props.friend.id, this.handleStatusChange);
  }

  handleStatusChange(status) {
    this.setState({ isOnline: status.isOnline });
  }

  render() {
    if (this.state.isOnline === null) {
      return 'Loading...';
    }
    return this.state.isOnline ? 'Online' : 'Offline';
  }
}
```

Notice how `componentDidMount` and `componentWillUnmount` need to mirror each other. Lifecycle methods force us to split this logic even though conceptually code in both of them is related to the same effect.

Note

Eagle-eyed readers may notice that this example also needs a `componentDidUpdate` method to be fully correct. We'll ignore this for now but will come back to it in a [later section](#) of this page.

Example Using Hooks

Let's see how we could write this component with Hooks.

You might be thinking that we'd need a separate effect to perform the cleanup. But code for adding and removing a subscription is so tightly related that `useEffect` is designed to keep it together. If your effect returns a function, React will run it when it is time to clean up:


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

function FriendStatus(props) {
  const [isOnline, setIsOnline] = useState(null);

  useEffect(() => {
    function handleStatusChange(status) {
      setIsOnline(status.isOnline);
    }
    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    // Specify how to clean up after this effect:
    return function cleanup() {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  });
  if (isOnline === null) {
    return 'Loading...';
  }
  return isOnline ? 'Online' : 'Offline';
}
```

Why did we return a function from our effect? This is the optional cleanup mechanism for effects. Every effect may return a function that cleans up after it. This lets us keep the logic for adding and removing subscriptions close to each other. They're part of the same effect!

When exactly does React clean up an effect? React performs the cleanup when the component unmounts. However, as we learned earlier, effects run for every render and not just once. This is why React *also* cleans up effects from the previous render before running the effects next time. We'll discuss [why this helps avoid bugs](#) and [how to opt out of this behavior in case it creates performance issues](#) later below.

Note

We don't have to return a named function from the effect. We called it `cleanup` here to clarify its purpose, but you could return an arrow function or call it something different.

Recap

We've learned that `useEffect` lets us express different kinds of side effects after a component renders. Some effects might require cleanup so they return a function:

```
useEffect(() => {
  function handleStatusChange(status) {
    setIsOnline(status.isOnline);
  }

  ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
  return () => {
    ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
  };
});
```

Other effects might not have a cleanup phase, and don't return anything.

```
useEffect(() => {
  document.title = `You clicked ${count} times`;
});
```

The Effect Hook unifies both use cases with a single API.

If you feel like you have a decent grasp on how the Effect Hook works, or if you feel overwhelmed, you can jump to the [next page about Rules of Hooks](#) now.

Tips for Using Effects

We'll continue this page with an in-depth look at some aspects of `useEffect` that experienced React users will likely be curious about. Don't feel obligated to dig into them now. You can always come back to this page to learn more details about the Effect

Hook.

Tip: Use Multiple Effects to Separate Concerns

One of the problems we outlined in the [Motivation](#) for Hooks is that class lifecycle methods often contain unrelated logic, but related logic gets broken up into several methods. Here is a component that combines the counter and the friend status indicator logic from the previous examples:

```
class FriendStatusWithCounter extends React.Component {
  constructor(props) {
    super(props);
    this.state = { count: 0, isOnline: null };
    this.handleStatusChange = this.handleStatusChange.bind(this);
  }

  componentDidMount() {
    document.title = `You clicked ${this.state.count} times`;
    ChatAPI.subscribeToFriendStatus(this.props.friend.id, this.handleStatusChange);
  }

  componentDidUpdate() {
    document.title = `You clicked ${this.state.count} times`;
  }

  componentWillUnmount() {
    ChatAPI.unsubscribeFromFriendStatus(this.props.friend.id, this.handleStatusChange);
  }

  handleStatusChange(status) {
    this.setState({
      isOnline: status.isOnline
    });
  }
  // ...
}
```

Note how the logic that sets `document.title` is split between `componentDidMount` and `componentDidUpdate`. The subscription logic is also spread between `componentDidMount` and `componentWillUnmount`. And `componentDidMount` contains code for both tasks.

So, how can Hooks solve this problem? Just like [you can use the `State` Hook more than once](#), you can also use several effects. This lets us separate unrelated logic into different effects:

```
function FriendStatusWithCounter(props) {
  const [count, setCount] = useState(0);
  useEffect(() => {
    document.title = `You clicked ${count} times`;
  });

  const [isOnline, setIsOnline] = useState(null);
  useEffect(() => {
    function handleStatusChange(status) {
      setIsOnline(status.isOnline);
    }

    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  });
  // ...
}
```

Hooks let us split the code based on what it is doing rather than a lifecycle method name. React will apply every effect used by the component, in the order they were specified.

Explanation: Why Effects Run on Each Update

If you're used to classes, you might be wondering why the effect cleanup phase happens after every re-render, and not just once during unmounting. Let's look at a practical example to see why this design helps us create components with fewer bugs.

[Earlier on this page](#), we introduced an example `FriendStatus` component that displays whether a friend is online or not. Our class reads `friend.id` from `this.props`, subscribes to the friend status after the component mounts, and unsubscribes during

unmounting:

```
componentDidMount() {
  ChatAPI.subscribeToFriendStatus(
    this.props.friend.id,
    this.handleStatusChange
  );
}

componentWillUnmount() {
  ChatAPI.unsubscribeFromFriendStatus(
    this.props.friend.id,
    this.handleStatusChange
  );
}
```

But what happens if the `friend` prop changes while the component is on the screen? Our component would continue displaying the online status of a different friend. This is a bug. We would also cause a memory leak or crash when unmounting since the unsubscribe call would use the wrong friend ID.

In a class component, we would need to add `componentDidUpdate` to handle this case:

```
componentDidMount() {
  ChatAPI.subscribeToFriendStatus(
    this.props.friend.id,
    this.handleStatusChange
  );
}

componentDidUpdate(prevProps) {
  // Unsubscribe from the previous friend.id
  ChatAPI.unsubscribeFromFriendStatus(
    prevProps.friend.id,
    this.handleStatusChange
  );
  // Subscribe to the next friend.id
  ChatAPI.subscribeToFriendStatus(
    this.props.friend.id,
    this.handleStatusChange
  );
}

componentWillUnmount() {
  ChatAPI.unsubscribeFromFriendStatus(
    this.props.friend.id,
    this.handleStatusChange
  );
}
```

Forgetting to handle `componentDidUpdate` properly is a common source of bugs in React applications.

Now consider the version of this component that uses Hooks:

```
function FriendStatus(props) {
  // ...
  useEffect(() => {
    // ...
    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  });
}
```

It doesn't suffer from this bug. (But we also didn't make any changes to it.)

There is no special code for handling updates because `useEffect` handles them *by default*. It cleans up the previous effects before applying the next effects. To illustrate this, here is a sequence of subscribe and unsubscribe calls that this component could produce over time:

```
// Mount with { friend: { id: 100 } } props
ChatAPI.subscribeToFriendStatus(100, handleStatusChange); // Run first effect

// Update with { friend: { id: 200 } } props
ChatAPI.unsubscribeFromFriendStatus(100, handleStatusChange); // Clean up previous effect
```

```

ChatAPI.subscribeToFriendStatus(200, handleStatusChange); // Run next effect

// Update with { friend: { id: 300 } } props
ChatAPI.unsubscribeFromFriendStatus(200, handleStatusChange); // Clean up previous effect
ChatAPI.subscribeToFriendStatus(300, handleStatusChange); // Run next effect

// Unmount
ChatAPI.unsubscribeFromFriendStatus(300, handleStatusChange); // Clean up last effect

```

This behavior ensures consistency by default and prevents bugs that are common in class components due to missing update logic.

Tip: Optimizing Performance by Skipping Effects

In some cases, cleaning up or applying the effect after every render might create a performance problem. In class components, we can solve this by writing an extra comparison with `prevProps` or `prevState` inside `componentDidUpdate`:

```

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

```

This requirement is common enough that it is built into the `useEffect` Hook API. You can tell React to *skip* applying an effect if certain values haven't changed between re-renders. To do so, pass an array as an optional second argument to `useEffect`:

```

useEffect(() => {
  document.title = `You clicked ${count} times`;
}, [count]); // Only re-run the effect if count changes

```

In the example above, we pass `[count]` as the second argument. What does this mean? If the `count` is `5`, and then our component re-renders with `count` still equal to `5`, React will compare `[5]` from the previous render and `[5]` from the next render. Because all items in the array are the same (`5 === 5`), React would skip the effect. That's our optimization.

When we render with `count` updated to `6`, React will compare the items in the `[5]` array from the previous render to items in the `[6]` array from the next render. This time, React will re-apply the effect because `5 !== 6`. If there are multiple items in the array, React will re-run the effect even if just one of them is different.

This also works for effects that have a cleanup phase:

```

useEffect(() => {
  function handleStatusChange(status) {
    setIsOnline(status.isOnline);
  }

  ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
  return () => {
    ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
  };
}, [props.friend.id]); // Only re-subscribe if props.friend.id changes

```

In the future, the second argument might get added automatically by a build-time transformation.

Note

If you use this optimization, make sure the array includes **all values from the component scope (such as props and state) that change over time and that are used by the effect**. Otherwise, your code will reference stale values from previous renders. Learn more about [how to deal with functions](#) and [what to do when the array changes too often](#).

If you want to run an effect and clean it up only once (on mount and unmount), you can pass an empty array (`[]`) as a second argument. This tells React that your effect doesn't depend on *any* values from props or state, so it never needs to re-run. This isn't handled as a special case — it follows directly from how the dependencies array always works.

If you pass an empty array (`[]`), the props and state inside the effect will always have their initial values. While passing `[]` as the second argument is closer to the familiar `componentDidMount` and `componentWillUnmount` mental model, there are usually better solutions to avoid re-running effects too often. Also, don't forget that React defers running `useEffect` until after the browser has painted, so doing extra work is less of a problem.

We recommend using the `exhaustive-deps` rule as part of our `eslint-plugin-react-hooks` package. It warns when dependencies are specified incorrectly and suggests a fix.

Next Steps

Congratulations! This was a long page, but hopefully by the end most of your questions about effects were answered. You've learned both the State Hook and the Effect Hook, and there is a *lot* you can do with both of them combined. They cover most of the use cases for classes — and where they don't, you might find the [additional Hooks](#) helpful.

We're also starting to see how Hooks solve problems outlined in [Motivation](#). We've seen how effect cleanup avoids duplication in `componentDidUpdate` and `componentWillUnmount`, brings related code closer together, and helps us avoid bugs. We've also seen how we can separate effects by their purpose, which is something we couldn't do in classes at all.

At this point you might be questioning how Hooks work. How can React know which `useState` call corresponds to which state variable between re-renders? How does React “match up” previous and next effects on every update? **On the next page we will learn about the [Rules of Hooks](#) — they're essential to making Hooks work.**

5. Rules of Hooks

Hooks are a new addition in React 16.8. They let you use state and other React features without writing a class.



Hooks are JavaScript functions, but you need to follow two rules when using them. We provide a [linter plugin](#) to enforce these rules automatically:

Only Call Hooks at the Top Level

Don't call Hooks inside loops, conditions, or nested functions. Instead, always use Hooks at the top level of your React function, before any early returns. By following this rule, you ensure that Hooks are called in the same order each time a component renders. That's what allows React to correctly preserve the state of Hooks between multiple `useState` and `useEffect` calls. (If you're curious, we'll explain this in depth [below](#).)

Only Call Hooks from React Functions

Don't call Hooks from regular JavaScript functions. Instead, you can:

-  Call Hooks from React function components.
-  Call Hooks from custom Hooks (we'll learn about them [on the next page](#)).

By following this rule, you ensure that all stateful logic in a component is clearly visible from its source code.

ESLint Plugin

We released an ESLint plugin called `eslint-plugin-react-hooks` that enforces these two rules. You can add this plugin to your project if you'd like to try it:

This plugin is included by default in [Create React App](#).

```
npm install eslint-plugin-react-hooks --save-dev
```

```
// Your ESLint configuration
{
  "plugins": [
    // ...
    "react-hooks"
  ],
  "rules": {
    // ...
  }
}
```

```

    "react-hooks/rules-of-hooks": "error", // Checks rules of Hooks
    "react-hooks/exhaustive-deps": "warn" // Checks effect dependencies
  }
}

```

You can skip to the next page explaining how to write [your own Hooks](#) now. On this page, we'll continue by explaining the reasoning behind these rules.

Explanation

As we [learned earlier](#), we can use multiple State or Effect Hooks in a single component:

```

function Form() {
  // 1. Use the name state variable
  const [name, setName] = useState('Mary');

  // 2. Use an effect for persisting the form
  useEffect(function persistForm() {
    localStorage.setItem('formData', name);
  });

  // 3. Use the surname state variable
  const [surname, setSurname] = useState('Poppins');

  // 4. Use an effect for updating the title
  useEffect(function updateTitle() {
    document.title = name + ' ' + surname;
  });

  // ...
}

```

So how does React know which state corresponds to which `useState` call? The answer is that **React relies on the order in which Hooks are called**. Our example works because the order of the Hook calls is the same on every render:

```

// -----
// First render
// -----
useState('Mary'); // 1. Initialize the name state variable with 'Mary'
useEffect(persistForm); // 2. Add an effect for persisting the form
useState('Poppins'); // 3. Initialize the surname state variable with 'Poppins'
useEffect(updateTitle); // 4. Add an effect for updating the title

// -----
// Second render
// -----
useState('Mary'); // 1. Read the name state variable (argument is ignored)
useEffect(persistForm); // 2. Replace the effect for persisting the form
useState('Poppins'); // 3. Read the surname state variable (argument is ignored)
useEffect(updateTitle); // 4. Replace the effect for updating the title

// ...

```

As long as the order of the Hook calls is the same between renders, React can associate some local state with each of them. But what happens if we put a Hook call (for example, the `persistForm` effect) inside a condition?

```

// 🚫 We're breaking the first rule by using a Hook in a condition
if (name !== '') {
  useEffect(function persistForm() {
    localStorage.setItem('formData', name);
  });
}

```

The `name !== ''` condition is `true` on the first render, so we run this Hook. However, on the next render the user might clear the form, making the condition `false`. Now that we skip this Hook during rendering, the order of the Hook calls becomes different:

```

useState('Mary'); // 1. Read the name state variable (argument is ignored)
// useEffect(persistForm) // 🚫 This Hook was skipped!
useState('Poppins'); // 🚫 2 (but was 3). Fail to read the surname state variable
useEffect(updateTitle); // 🚫 3 (but was 4). Fail to replace the effect

```

React wouldn't know what to return for the second `useState` Hook call. React expected that the second Hook call in this component corresponds to the `persistForm` effect, just like during the previous render, but it doesn't anymore. From that point, every next Hook call after the one we skipped would also shift by one, leading to bugs.

This is why Hooks must be called on the top level of our components. If we want to run an effect conditionally, we can put that condition *inside* our Hook:

```
useEffect(function persistForm() {
  // 🚫 We're not breaking the first rule anymore
  if (name !== '') {
    localStorage.setItem('formData', name);
  }
});
```

Note that you don't need to worry about this problem if you use the [provided lint rule](#). But now you also know *why* Hooks work this way, and which issues the rule is preventing.

Next Steps

Finally, we're ready to learn about [writing your own Hooks](#)! Custom Hooks let you combine Hooks provided by React into your own abstractions, and reuse common stateful logic between different components.

6. Building Your Own Hooks

Hooks are a new addition in React 16.8. They let you use state and other React features without writing a class.

Building your own Hooks lets you extract component logic into reusable functions.

When we were learning about [using the Effect Hook](#), we saw this component from a chat application that displays a message indicating whether a friend is online or offline:

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

function FriendStatus(props) {
  const [isOnline, setIsOnline] = useState(null);
  useEffect(() => {
    function handleStatusChange(status) {
      setIsOnline(status.isOnline);
    }
    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  });
  if (isOnline === null) {
    return 'Loading...';
  }
  return isOnline ? 'Online' : 'Offline';
}
```

Now let's say that our chat application also has a contact list, and we want to render names of online users with a green color. We could copy and paste similar logic above into our `FriendListItem` component but it wouldn't be ideal:

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

function FriendListItem(props) {
  const [isOnline, setIsOnline] = useState(null);
  useEffect(() => {
    function handleStatusChange(status) {
      setIsOnline(status.isOnline);
    }
    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  });
  return <li style={{ color: isOnline ? 'green' : 'black' }}> {props.friend.name}</li>;
}
```

Instead, we'd like to share this logic between `FriendStatus` and `FriendListItem`.

Traditionally in React, we've had two popular ways to share stateful logic between components: [render props](#) and [higher-order components](#). We will now look at how Hooks solve many of the same problems without forcing you to add more components to the tree.

Extracting a Custom Hook

When we want to share logic between two JavaScript functions, we extract it to a third function. Both components and Hooks are functions, so this works for them too!

A custom Hook is a JavaScript function whose name starts with "use" and that may call other Hooks. For example, `useFriendStatus` below is our first custom Hook:

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

function useFriendStatus(friendID) {
  const [isOnline, setIsOnline] = useState(null);

  useEffect(() => {
    function handleStatusChange(status) {
      setIsOnline(status.isOnline);
    }

    ChatAPI.subscribeToFriendStatus(friendID, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(friendID, handleStatusChange);
    };
  });

  return isOnline;
}
```

There's nothing new inside of it — the logic is copied from the components above. Just like in a component, make sure to only call other Hooks unconditionally at the top level of your custom Hook.

Unlike a React component, a custom Hook doesn't need to have a specific signature. We can decide what it takes as arguments, and what, if anything, it should return. In other words, it's just like a normal function. Its name should always start with `use` so that you can tell at a glance that the [rules of Hooks](#) apply to it.

The purpose of our `useFriendStatus` Hook is to subscribe us to a friend's status. This is why it takes `friendID` as an argument, and returns whether this friend is online:

```
function useFriendStatus(friendID) {
  const [isOnline, setIsOnline] = useState(null);

  // ...

  return isOnline;
}
```

Now let's see how we can use our custom Hook.

Using a Custom Hook

In the beginning, our stated goal was to remove the duplicated logic from the `FriendStatus` and `FriendListItem` components. Both of them want to know whether a friend is online.

Now that we've extracted this logic to a `useFriendStatus` hook, we can *just use it*:

```
function FriendStatus(props) {
  const isOnline = useFriendStatus(props.friend.id);
  if (isOnline === null) {
    return 'Loading...';
  }
  return isOnline ? 'Online' : 'Offline';
}

function FriendListItem(props) {
  const isOnline = useFriendStatus(props.friend.id);
  return <li style={{ color: isOnline ? 'green' : 'black' }}> {props.friend.name}</li>;
}
```


Is this code equivalent to the original examples? Yes, it works in exactly the same way. If you look closely, you'll notice we didn't make any changes to the behavior. All we did was to extract some common code between two functions into a separate function. **Custom Hooks are a convention that naturally follows from the design of Hooks, rather than a React feature.**

Do I have to name my custom Hooks starting with “use”? Please do. This convention is very important. Without it, we wouldn't be able to automatically check for violations of [rules of Hooks](#) because we couldn't tell if a certain function contains calls to Hooks inside of it.

Do two components using the same Hook share state? No. Custom Hooks are a mechanism to reuse *stateful logic* (such as setting up a subscription and remembering the current value), but every time you use a custom Hook, all state and effects inside of it are fully isolated.

How does a custom Hook get isolated state? Each *call* to a Hook gets isolated state. Because we call `useFriendStatus` directly, from React's point of view our component just calls `useState` and `useEffect`. And as we [learned earlier](#), we can call `useState` and `useEffect` many times in one component, and they will be completely independent.

Tip: Pass Information Between Hooks

Since Hooks are functions, we can pass information between them.

To illustrate this, we'll use another component from our hypothetical chat example. This is a chat message recipient picker that displays whether the currently selected friend is online:

```
const friendList = [
  { id: 1, name: 'Phoebe' },
  { id: 2, name: 'Rachel' },
  { id: 3, name: 'Ross' }
];

function ChatRecipientPicker() {
  const [recipientID, setRecipientID] = useState(1);
  const isRecipientOnline = useFriendStatus(recipientID);
  return (
    <>
      <Circle color={isRecipientOnline ? 'green' : 'red'} />
      <select value={recipientID} onChange={(e) => setRecipientID(Number(e.target.value))>
        {friendList.map((friend) => (
          <option key={friend.id} value={friend.id}>
            {friend.name}
          </option>
        ))}
      </select>
    </>
  );
}
```

We keep the currently chosen friend ID in the `recipientID` state variable, and update it if the user chooses a different friend in the `<select>` picker.

Because the `useState` Hook call gives us the latest value of the `recipientID` state variable, we can pass it to our custom `useFriendStatus` Hook as an argument:

```
const [recipientID, setRecipientID] = useState(1);
const isRecipientOnline = useFriendStatus(recipientID);
```

This lets us know whether the *currently selected* friend is online. If we pick a different friend and update the `recipientID` state variable, our `useFriendStatus` Hook will unsubscribe from the previously selected friend, and subscribe to the status of the newly selected one.

useYourImagination()

Custom Hooks offer the flexibility of sharing logic that wasn't possible in React components before. You can write custom Hooks that cover a wide range of use cases like form handling, animation, declarative subscriptions, timers, and probably many more we haven't considered. What's more, you can build Hooks that are just as easy to use as React's built-in features.

Try to resist adding abstraction too early. Now that function components can do more, it's likely that the average function component in your codebase will become longer. This is normal — don't feel like you *have to* immediately split it into Hooks.

But we also encourage you to start spotting cases where a custom Hook could hide complex logic behind a simple interface, or help untangle a messy component.

For example, maybe you have a complex component that contains a lot of local state that is managed in an ad-hoc way. `useState` doesn't make centralizing the update logic any easier so you might prefer to write it as a [Redux](#) reducer:

```
function todosReducer(state, action) {
  switch (action.type) {
    case 'add':
      return [
        ...state,
        {
          text: action.text,
          completed: false
        }
      ];
    // ... other actions ...
    default:
      return state;
  }
}
```

Reducers are very convenient to test in isolation, and scale to express complex update logic. You can further break them apart into smaller reducers if necessary. However, you might also enjoy the benefits of using React local state, or might not want to install another library.

So what if we could write a `useReducer` Hook that lets us manage the *local* state of our component with a reducer? A simplified version of it might look like this:

```
function useReducer(reducer, initialState) {
  const [state, setState] = useState(initialState);

  function dispatch(action) {
    const nextState = reducer(state, action);
    setState(nextState);
  }

  return [state, dispatch];
}
```

Now we could use it in our component, and let the reducer drive its state management:

```
function Todos() {
  const [todos, dispatch] = useReducer(todosReducer, []);
  function handleAddClick(text) {
    dispatch({ type: 'add', text });
  }

  // ...
}
```

The need to manage local state with a reducer in a complex component is common enough that we've built the `useReducer` Hook right into React. You'll find it together with other built-in Hooks in the [Hooks API reference](#).

7. Hooks API Reference

Hooks are a new addition in React 16.8. They let you use state and other React features without writing a class.

This page describes the APIs for the built-in Hooks in React.

If you're new to Hooks, you might want to check out [the overview](#) first. You may also find useful information in the [frequently asked questions](#) section.

- [Basic Hooks](#)
 - `useState`
 - `useEffect`
 - `useContext`
- [Additional Hooks](#)

- `useReducer`
- `useCallback`
- `useMemo`
- `useRef`
- `useImperativeHandle`
- `useLayoutEffect`
- `useDebugValue`
- `useDeferredValue`
- `useTransition`
- `useId`
- Library Hooks
 - `useSyncExternalStore`
 - `useInsertionEffect`

Basic Hooks

`useState`

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

Returns a stateful value, and a function to update it.

During the initial render, the returned state (`state`) is the same as the value passed as the first argument (`initialState`).

The `setState` function is used to update the state. It accepts a new state value and enqueues a re-render of the component.

```
setState(newState);
```

During subsequent re-renders, the first value returned by `useState` will always be the most recent state after applying updates.

Note

React guarantees that `setState` function identity is stable and won't change on re-renders. This is why it's safe to omit from the `useEffect` or `useCallback` dependency list.

Functional updates

If the new state is computed using the previous state, you can pass a function to `setState`. The function will receive the previous value, and return an updated value. Here's an example of a counter component that uses both forms of `setState`:

```
function Counter({ initialCount }) {
  const [count, setCount] = useState(initialCount);
  return (
    <>
      Count: {count}
      <button onClick={() => setCount(initialCount)}>Reset</button>
      <button onClick={() => setCount((prevCount) => prevCount - 1)}>-</button>
      <button onClick={() => setCount((prevCount) => prevCount + 1)}>+</button>
    </>
  );
}
```

The "+" and "-" buttons use the functional form, because the updated value is based on the previous value. But the "Reset" button uses the normal form, because it always sets the count back to the initial value.

If your update function returns the exact same value as the current state, the subsequent rerender will be skipped completely.

Note

Unlike the `setState` method found in class components, `useState` does not automatically merge update objects. You can replicate this behavior by combining the function updater form with object

spread syntax:

```
const [state, setState] = useState({});
setState((prevState) => {
  // Object.assign would also work
  return { ...prevState, ...updatedValues };
});
```

Another option is `useReducer`, which is more suited for managing state objects that contain multiple sub-values.

Lazy initial state

The `initialState` argument is the state used during the initial render. In subsequent renders, it is disregarded. If the initial state is the result of an expensive computation, you may provide a function instead, which will be executed only on the initial render:

```
const [state, setState] = useState(() => {
  const initialState = someExpensiveComputation(props);
  return initialState;
});
```

Bailing out of a state update

If you update a State Hook to the same value as the current state, React will bail out without rendering the children or firing effects. (React uses the [Object.is comparison algorithm](#).)

Note that React may still need to render that specific component again before bailing out. That shouldn't be a concern because React won't unnecessarily go "deeper" into the tree. If you're doing expensive calculations while rendering, you can optimize them with `useMemo`.

Batching of state updates

React may group several state updates into a single re-render to improve performance. Normally, this improves performance and shouldn't affect your application's behavior.

Before React 18, only updates inside React event handlers were batched. Starting with React 18, [batching is enabled for all updates by default](#). Note that React makes sure that updates from several *different* user-initiated events — for example, clicking a button twice — are always processed separately and do not get batched. This prevents logical mistakes.

In the rare case that you need to force the DOM update to be applied synchronously, you may wrap it in `flushSync`. However, this can hurt performance so do this only where needed.

useEffect

```
useEffect(() => {
```

Accepts a function that contains imperative, possibly effectful code.

Mutations, subscriptions, timers, logging, and other side effects are not allowed inside the main body of a function component (referred to as React's *render phase*). Doing so will lead to confusing bugs and inconsistencies in the UI.

Instead, use `useEffect`. The function passed to `useEffect` will run after the render is committed to the screen. Think of effects as an escape hatch from React's purely functional world into the imperative world.

By default, effects run after every completed render, but you can choose to fire them [only when certain values have changed](#).

Cleaning up an effect

Often, effects create resources that need to be cleaned up before the component leaves the screen, such as a subscription or timer ID. To do this, the function passed to `useEffect` may return a clean-up function. For example, to create a subscription:

```
useEffect(() => {
  const subscription = props.source.subscribe();
  return () => {
    // Clean up the subscription
    subscription.unsubscribe();
  };
});
```

```
};  
});
```

The clean-up function runs before the component is removed from the UI to prevent memory leaks. Additionally, if a component renders multiple times (as they typically do), the **previous effect is cleaned up before executing the next effect**. In our example, this means a new subscription is created on every update. To avoid firing an effect on every update, refer to the next section.

Timing of effects

Unlike `componentDidMount` and `componentDidUpdate`, the function passed to `useEffect` fires **after** layout and paint, during a deferred event. This makes it suitable for the many common side effects, like setting up subscriptions and event handlers, because most types of work shouldn't block the browser from updating the screen.

However, not all effects can be deferred. For example, a DOM mutation that is visible to the user must fire synchronously before the next paint so that the user does not perceive a visual inconsistency. (The distinction is conceptually similar to passive versus active event listeners.) For these types of effects, React provides one additional Hook called `useLayoutEffect`. It has the same signature as `useEffect`, and only differs in when it is fired.

Additionally, starting in React 18, the function passed to `useEffect` will fire synchronously **before** layout and paint when it's the result of a discrete user input such as a click, or when it's the result of an update wrapped in `flushSync`. This behavior allows the result of the effect to be observed by the event system, or by the caller of `flushSync`.

Note

This only affects the timing of when the function passed to `useEffect` is called - updates scheduled inside these effects are still deferred. This is different than `useLayoutEffect`, which fires the function and processes the updates inside of it immediately.

Even in cases where `useEffect` is deferred until after the browser has painted, it's guaranteed to fire before any new renders. React will always flush a previous render's effects before starting a new update.

Conditionally firing an effect

The default behavior for effects is to fire the effect after every completed render. That way an effect is always recreated if one of its dependencies changes.

However, this may be overkill in some cases, like the subscription example from the previous section. We don't need to create a new subscription on every update, only if the `source` prop has changed.

To implement this, pass a second argument to `useEffect` that is the array of values that the effect depends on. Our updated example now looks like this:

```
useEffect(() => {  
  const subscription = props.source.subscribe();  
  return () => {  
    subscription.unsubscribe();  
  };  
}, [props.source]);
```

Now the subscription will only be recreated when `props.source` changes.

Note

If you use this optimization, make sure the array includes **all values from the component scope (such as props and state) that change over time and that are used by the effect**. Otherwise, your code will reference stale values from previous renders. Learn more about [how to deal with functions](#) and what to do when the [array values change too often](#).

If you want to run an effect and clean it up only once (on mount and unmount), you can pass an empty array (`[]`) as a second argument. This tells React that your effect doesn't depend

on *any* values from props or state, so it never needs to re-run. This isn't handled as a special case — it follows directly from how the dependencies array always works.

If you pass an empty array (`[]`), the props and state inside the effect will always have their initial values. While passing `[]` as the second argument is closer to the familiar `componentDidMount` and `componentWillUnmount` mental model, there are usually better solutions to avoid re-running effects too often. Also, don't forget that React defers running `useEffect` until after the browser has painted, so doing extra work is less of a problem.

We recommend using the `exhaustive-deps` rule as part of our `eslint-plugin-react-hooks` package. It warns when dependencies are specified incorrectly and suggests a fix.

The array of dependencies is not passed as arguments to the effect function. Conceptually, though, that's what they represent: every value referenced inside the effect function should also appear in the dependencies array. In the future, a sufficiently advanced compiler could create this array automatically.

useContext

```
const value = useContext(MyContext);
```

Accepts a context object (the value returned from `React.createContext`) and returns the current context value for that context. The current context value is determined by the `value` prop of the nearest `<MyContext.Provider>` above the calling component in the tree.

When the nearest `<MyContext.Provider>` above the component updates, this Hook will trigger a rerender with the latest context `value` passed to that `MyContext` provider. Even if an ancestor uses `React.memo` or `shouldComponentUpdate`, a rerender will still happen starting at the component itself using `useContext`.

Don't forget that the argument to `useContext` must be the *context object itself*:

- **Correct:** `useContext(MyContext)`
- **Incorrect:** `useContext(MyContext.Consumer)`
- **Incorrect:** `useContext(MyContext.Provider)`

A component calling `useContext` will always re-render when the context value changes. If re-rendering the component is expensive, you can optimize it by using memoization.

Tip

If you're familiar with the context API before Hooks, `useContext(MyContext)` is equivalent to `static contextType = MyContext` in a class, or to `<MyContext.Consumer>`.

`useContext(MyContext)` only lets you *read* the context and subscribe to its changes. You still need a `<MyContext.Provider>` above in the tree to *provide* the value for this context.

Putting it together with Context.Provider

```
const themes = {
  light: {
    foreground: '#000000',
    background: '#eeeeee'
  },
  dark: {
    foreground: '#ffffff',
    background: '#222222'
  }
};

const ThemeContext = React.createContext(themes.light);

function App() {
  return (
    <ThemeContext.Provider value={themes.dark}>
      <Toolbar />
    </ThemeContext.Provider>
  );
}
```

```

}

function Toolbar(props) {
  return (
    <div>
      <ThemedButton />
    </div>
  );
}

function ThemedButton() {
  const theme = useContext(ThemeContext);
  return (
    <button style={{ background: theme.background, color: theme.foreground }}>
      I am styled by theme context!
    </button>
  );
}

```

This example is modified for hooks from a previous example in the [Context Advanced Guide](#), where you can find more information about when and how to use Context.

Additional Hooks

The following Hooks are either variants of the basic ones from the previous section, or only needed for specific edge cases. Don't stress about learning them up front.

useReducer

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

An alternative to `useState`. Accepts a reducer of type `(state, action) => newState`, and returns the current state paired with a `dispatch` method. (If you're familiar with Redux, you already know how this works.)

`useReducer` is usually preferable to `useState` when you have complex state logic that involves multiple sub-values or when the next state depends on the previous one. `useReducer` also lets you optimize performance for components that trigger deep updates because [you can pass `dispatch` down instead of callbacks](#).

Here's the counter example from the [useState](#) section, rewritten to use a reducer:

```

const initialState = { count: 0 };

function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return { count: state.count + 1 };
    case 'decrement':
      return { count: state.count - 1 };
    default:
      throw new Error();
  }
}

function Counter() {
  const [state, dispatch] = useReducer(reducer, initialState);
  return (
    <>
      Count: {state.count}
      <button onClick={() => dispatch({ type: 'decrement' })}>-</button>
      <button onClick={() => dispatch({ type: 'increment' })}>+</button>
    </>
  );
}

```

Note

React guarantees that `dispatch` function identity is stable and won't change on re-renders. This is why it's safe to omit from the `useEffect` or `useCallback` dependency list.

Specifying the initial state

There are two different ways to initialize `useReducer` state. You may choose either one depending on the use case. The simplest way is to pass the initial state as a second argument:

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

Note

React doesn't use the `state = initialState` argument convention popularized by Redux. The initial value sometimes needs to depend on props and so is specified from the Hook call instead. If you feel strongly about this, you can call `useReducer(reducer, undefined, reducer)` to emulate the Redux behavior, but it's not encouraged.

Lazy initialization

You can also create the initial state lazily. To do this, you can pass an `init` function as the third argument. The initial state will be set to `init(initialArg)`.

It lets you extract the logic for calculating the initial state outside the reducer. This is also handy for resetting the state later in response to an action:

```
function init(initialCount) {
  return { count: initialCount };
}

function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return { count: state.count + 1 };
    case 'decrement':
      return { count: state.count - 1 };
    case 'reset':
      return init(action.payload);
    default:
      throw new Error();
  }
}

function Counter({ initialCount }) {
  const [state, dispatch] = useReducer(reducer, initialCount, init);
  return (
    <>
      Count: {state.count}
      <button onClick={() => dispatch({ type: 'reset', payload: initialCount })}>
        Reset
      </button>
      <button onClick={() => dispatch({ type: 'decrement' })}>-</button>
      <button onClick={() => dispatch({ type: 'increment' })}>+</button>
    </>
  );
}
```

Bailing out of a dispatch

If you return the same value from a Reducer Hook as the current state, React will bail out without rendering the children or firing effects. (React uses the [Object.is comparison algorithm](#).)

Note that React may still need to render that specific component again before bailing out. That shouldn't be a concern because React won't unnecessarily go "deeper" into the tree. If you're doing expensive calculations while rendering, you can optimize them with `useMemo`.

useCallback

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

Returns a [memoized](#) callback.

Pass an inline callback and an array of dependencies. `useCallback` will return a memoized version of the callback that only changes if one of the dependencies has changed. This is useful when passing callbacks to optimized child components that rely on reference equality to prevent unnecessary renders (e.g. `shouldComponentUpdate`).

`useCallback(fn, deps)` is equivalent to `useMemo(() => fn, deps)`.

Note

The array of dependencies is not passed as arguments to the callback. Conceptually, though, that's what they represent: every value referenced inside the callback should also appear in the dependencies array. In the future, a sufficiently advanced compiler could create this array automatically.

We recommend using the `exhaustive-deps` rule as part of our `eslint-plugin-react-hooks` package. It warns when dependencies are specified incorrectly and suggests a fix.

useMemo

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

Returns a memoized value.

Pass a “create” function and an array of dependencies. `useMemo` will only recompute the memoized value when one of the dependencies has changed. This optimization helps to avoid expensive calculations on every render.

Remember that the function passed to `useMemo` runs during rendering. Don't do anything there that you wouldn't normally do while rendering. For example, side effects belong in `useEffect`, not `useMemo`.

If no array is provided, a new value will be computed on every render.

You may rely on `useMemo` as a performance optimization, not as a semantic guarantee. In the future, React may choose to “forget” some previously memoized values and recalculate them on next render, e.g. to free memory for offscreen components. Write your code so that it still works without `useMemo` — and then add it to optimize performance.

Note

The array of dependencies is not passed as arguments to the function. Conceptually, though, that's what they represent: every value referenced inside the function should also appear in the dependencies array. In the future, a sufficiently advanced compiler could create this array automatically.

We recommend using the `exhaustive-deps` rule as part of our `eslint-plugin-react-hooks` package. It warns when dependencies are specified incorrectly and suggests a fix.

useRef

```
const refContainer = useRef(initialValue);
```

`useRef` returns a mutable ref object whose `.current` property is initialized to the passed argument (`initialValue`). The returned object will persist for the full lifetime of the component.

A common use case is to access a child imperatively:

```
function TextInputWithFocusButton() {
  const inputEl = useRef(null);
  const onButtonClick = () => {
    // 'current' points to the mounted text input element
    inputEl.current.focus();
  };
  return (
    <>
      <input ref={inputEl} type="text" />
      <button onClick={onButtonClick}>Focus the input</button>
    </>
  );
}
```

Essentially, `useRef` is like a “box” that can hold a mutable value in its `.current` property.

You might be familiar with refs primarily as a way to [access the DOM](#). If you pass a ref object to React with `<div ref={myRef} />`, React will set its `.current` property to the corresponding DOM node whenever that node changes.

However, `useRef()` is useful for more than the `ref` attribute. It’s [handy for keeping any mutable value around](#) similar to how you’d use instance fields in classes.

This works because `useRef()` creates a plain JavaScript object. The only difference between `useRef()` and creating a `{current: ...}` object yourself is that `useRef` will give you the same ref object on every render.

Keep in mind that `useRef` *doesn’t* notify you when its content changes. Mutating the `.current` property doesn’t cause a re-render. If you want to run some code when React attaches or detaches a ref to a DOM node, you may want to use a [callback ref](#) instead.

useImperativeHandle

```
useImperativeHandle(ref, createHandle, [deps])
```

`useImperativeHandle` customizes the instance value that is exposed to parent components when using `ref`. As always, imperative code using refs should be avoided in most cases. `useImperativeHandle` should be used with `forwardRef`:

```
function FancyInput(props, ref) {
  const inputRef = useRef();
  useImperativeHandle(ref, () => ({
    focus: () => {
      inputRef.current.focus();
    }
  }));
  return <input ref={inputRef} ... />;
}
FancyInput = forwardRef(FancyInput);
```

In this example, a parent component that renders `<FancyInput ref={inputRef} />` would be able to call `inputRef.current.focus()`.

useLayoutEffect

The signature is identical to `useEffect`, but it fires synchronously after all DOM mutations. Use this to read layout from the DOM and synchronously re-render. Updates scheduled inside `useLayoutEffect` will be flushed synchronously, before the browser has a chance to paint.

Prefer the standard `useEffect` when possible to avoid blocking visual updates.

Tip

If you’re migrating code from a class component, note `useLayoutEffect` fires in the same phase as `componentDidMount` and `componentDidUpdate`. However, **we recommend starting with `useEffect` first** and only trying `useLayoutEffect` if that causes a problem.

If you use server rendering, keep in mind that *neither* `useLayoutEffect` nor `useEffect` can run until the JavaScript is downloaded. This is why React warns when a server-rendered component contains `useLayoutEffect`. To fix this, either move that logic to `useEffect` (if it isn’t necessary for the first render), or delay showing that component until after the client renders (if the HTML looks broken until `useLayoutEffect` runs).

To exclude a component that needs layout effects from the server-rendered HTML, render it conditionally with `showChild && <Child />` and defer showing it with `useEffect(() => { setShowChild(true); }, [])`. This way, the UI doesn’t appear broken before hydration.

useDebugValue

```
useDebugValue(value)
```

`useDebugValue` can be used to display a label for custom hooks in React DevTools.

For example, consider the `useFriendStatus` custom Hook described in [“Building Your Own Hooks”](#):

```
function useFriendStatus(friendID) {
  const [isOnline, setIsOnline] = useState(null);

  // ...

  // Show a label in DevTools next to this Hook // e.g.
  ('FriendStatus: Online');
  useDebugValue(isOnline ? 'Online' : 'Offline');
  return isOnline;
}
```

Tip

We don't recommend adding debug values to every custom Hook. It's most valuable for custom Hooks that are part of shared libraries.

Defer formatting debug values

In some cases formatting a value for display might be an expensive operation. It's also unnecessary unless a Hook is actually inspected.

For this reason `useDebugValue` accepts a formatting function as an optional second parameter. This function is only called if the Hooks are inspected. It receives the debug value as a parameter and should return a formatted display value.

For example a custom Hook that returned a `Date` value could avoid calling the `toDateString` function unnecessarily by passing the following formatter:

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

useDeferredValue

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

`useDeferredValue` accepts a value and returns a new copy of the value that will defer to more urgent updates. If the current render is the result of an urgent update, like user input, React will return the previous value and then render the new value after the urgent render has completed.

This hook is similar to user-space hooks which use debouncing or throttling to defer updates. The benefits to using `useDeferredValue` is that React will work on the update as soon as other work finishes (instead of waiting for an arbitrary amount of time), and like [startTransition](#), deferred values can suspend without triggering an unexpected fallback for existing content.

Memoizing deferred children

`useDeferredValue` only defers the value that you pass to it. If you want to prevent a child component from re-rendering during an urgent update, you must also memoize that component with `React.memo` or `React.useMemo`:

```
function Typeahead() {
  const query = useSearchQuery('');
  const deferredQuery = useDeferredValue(query);

  // Memoizing tells React to only re-render when deferredQuery changes,
  // not when query changes.
  const suggestions = useMemo(() => {
    return (<SearchSuggestions query={deferredQuery} />), [deferredQuery];
  });

  return (
    <>
      <SearchInput query={query} />
      <Suspense fallback="Loading results...">{suggestions}</Suspense>
    </>
  );
}
```

Memoizing the children tells React that it only needs to re-render them when `deferredQuery` changes and not when `query` changes. This caveat is not unique to `useDeferredValue`, and it's the same pattern you would use with similar hooks that use debouncing or throttling.

useTransition

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

Returns a stateful value for the pending state of the transition, and a function to start it.

`startTransition` lets you mark updates in the provided callback as transitions:

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

`isPending` indicates when a transition is active to show a pending state:

```
function App() {  
  const [isPending, startTransition] = useTransition();  
  const [count, setCount] = useState(0);  
  
  function handleClick() {  
    startTransition(() => {  
      setCount((c) => c + 1);  
    });  
  }  
  
  return (  
    <div>  
      {isPending && <Spinner />}  
      <button onClick={handleClick}>{count}</button>  
    </div>  
  );  
}
```

Note:

Updates in a transition yield to more urgent updates such as clicks.

Updates in a transition will not show a fallback for re-suspended content. This allows the user to continue interacting with the current content while rendering the update.

useId

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

`useId` is a hook for generating unique IDs that are stable across the server and client, while avoiding hydration mismatches.

Note

`useId` is **not** for generating keys in a list. Keys should be generated from your data.

For a basic example, pass the `id` directly to the elements that need it:

```
function Checkbox() {  
  const id = useId();  
  return (  
    <>  
      <label htmlFor={id}>Do you like React?</label>  
      <input id={id} type="checkbox" name="react" />  
    </>  
  );  
}
```

For multiple IDs in the same component, append a suffix using the same `id`:

```
function NameFields() {
  const id = useId();
  return (
    <div>
      <label htmlFor={id + '-firstName'}>First Name</label>
      <div>
        <input id={id + '-firstName'} type="text" />
      </div>
      <label htmlFor={id + '-lastName'}>Last Name</label>
      <div>
        <input id={id + '-lastName'} type="text" />
      </div>
    </div>
  );
}
```

Note:

`useId` generates a string that includes the `:` token. This helps ensure that the token is unique, but is not supported in CSS selectors or APIs like `querySelectorAll`.

`useId` supports an `identifierPrefix` to prevent collisions in multi-root apps. To configure, see the options for `hydrateRoot` and `ReactDOMServer`.

Library Hooks

The following Hooks are provided for library authors to integrate libraries deeply into the React model, and are not typically used in application code.

`useSyncExternalStore`

```
const state = useSyncExternalStore(subscribe, getSnapshot[, getServerSnapshot]);
```

`useSyncExternalStore` is a hook recommended for reading and subscribing from external data sources in a way that's compatible with concurrent rendering features like selective hydration and time slicing.

This method returns the value of the store and accepts three arguments:

- `subscribe`: function to register a callback that is called whenever the store changes.
- `getSnapshot`: function that returns the current value of the store.
- `getServerSnapshot`: function that returns the snapshot used during server rendering.

The most basic example simply subscribes to the entire store:

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

However, you can also subscribe to a specific field:

```
const selectedField = useSyncExternalStore(store.subscribe, () => {
  return store.getSnapshot().selectedField;
});
```

When server rendering, you must serialize the store value used on the server, and provide it to `useSyncExternalStore`. React will use this snapshot during hydration to prevent server mismatches:

```
const selectedField = useSyncExternalStore(
  store.subscribe,
  () => store.getSnapshot().selectedField,
  () => INITIAL_SERVER_SNAPSHOT.selectedField
);
```

Note:

`getSnapshot` must return a cached value. If `getSnapshot` is called multiple times in a row, it must return the same exact value unless there was a store update in between.

A shim is provided for supporting multiple React versions published as `use-sync-external-store/shim`. This shim will prefer `useSyncExternalStore` when available, and fallback to a user-space implementation when it's not.

As a convenience, we also provide a version of the API with automatic support for memoizing the result of `getSnapshot` published as `use-sync-external-store/with-selector`.

useInsertionEffect

```
useInsertionEffect(didUpdate);
```

The signature is identical to `useEffect`, but it fires synchronously *before* all DOM mutations. Use this to inject styles into the DOM before reading layout in `useLayoutEffect`. Since this hook is limited in scope, this hook does not have access to refs and cannot schedule updates.

Note:

`useInsertionEffect` should be limited to css-in-js library authors.

Prefer `useEffect` or `useLayoutEffect` instead.

8. Hooks FAQ

Hooks are a new addition in React 16.8. They let you use state and other React features without writing a class.

This page answers some of the frequently asked questions about [Hooks](#).

- **Adoption Strategy.**

- [Which versions of React include Hooks?](#)
- [Do I need to rewrite all my class components?](#)
- [What can I do with Hooks that I couldn't with classes?](#)
- [How much of my React knowledge stays relevant?](#)
- [Should I use Hooks, classes, or a mix of both?](#)
- [Do Hooks cover all use cases for classes?](#)
- [Do Hooks replace render props and higher-order components?](#)
- [What do Hooks mean for popular APIs like Redux connect\(\) and React Router?](#)
- [Do Hooks work with static typing?](#)
- [How to test components that use Hooks?](#)
- [What exactly do the lint rules enforce?](#)

- **From Classes to Hooks**

- [How do lifecycle methods correspond to Hooks?](#)
- [How can I do data fetching with Hooks?](#)
- [Is there something like instance variables?](#)
- [Should I use one or many state variables?](#)
- [Can I run an effect only on updates?](#)
- [How to get the previous props or state?](#)

- [Why am I seeing stale props or state inside my function?](#)
- [How do I implement `getDerivedStateFromProps`?](#)
- [Is there something like `forceUpdate`?](#)
- [Can I make a ref to a function component?](#)
- [How can I measure a DOM node?](#)
- [What does `const \[thing, setThing\] = useState\(\)` mean?](#)
- **[Performance Optimizations](#)**
 - [Can I skip an effect on updates?](#)
 - [Is it safe to omit functions from the list of dependencies?](#)
 - [What can I do if my effect dependencies change too often?](#)
 - [How do I implement `shouldComponentUpdate`?](#)
 - [How to memoize calculations?](#)
 - [How to create expensive objects lazily?](#)
 - [Are Hooks slow because of creating functions in render?](#)
 - [How to avoid passing callbacks down?](#)
 - [How to read an often-changing value from `useCallback`?](#)
- **[Under the Hood](#)**
 - [How does React associate Hook calls with components?](#)
 - [What is the prior art for Hooks?](#)

Adoption Strategy

Which versions of React include Hooks?

Starting with 16.8.0, React includes a stable implementation of React Hooks for:

- React DOM
- React Native
- React DOM Server
- React Test Renderer
- React Shallow Renderer

Note that **to enable Hooks, all React packages need to be 16.8.0 or higher**. Hooks won't work if you forget to update, for example, React DOM.

[React Native 0.59](#) and above support Hooks.

Do I need to rewrite all my class components?

No. There are [no plans](#) to remove classes from React — we all need to keep shipping products and can't afford rewrites. We recommend trying Hooks in new code.

What can I do with Hooks that I couldn't with classes?

Hooks offer a powerful and expressive new way to reuse functionality between components. [“Building Your Own Hooks”](#) provides a glimpse of what's possible. [This article](#) by a React core team member dives deeper into the new capabilities unlocked by Hooks.

How much of my React knowledge stays relevant?

Hooks are a more direct way to use the React features you already know — such as state, lifecycle, context, and refs. They don't fundamentally change how React works, and your knowledge of components, props, and top-down data flow is just as relevant.

Hooks do have a learning curve of their own. If there's something missing in this documentation, [raise an issue](#) and we'll try to help.

Should I use Hooks, classes, or a mix of both?

When you're ready, we'd encourage you to start trying Hooks in new components you write. Make sure everyone on your team is on board with using them and familiar with this documentation. We don't recommend rewriting your existing classes to Hooks unless you planned to rewrite them anyway (e.g. to fix bugs).

You can't use Hooks *inside* a class component, but you can definitely mix classes and function components with Hooks in a single tree. Whether a component is a class or a function that uses Hooks is an implementation detail of that component. In the longer term, we expect Hooks to be the primary way people write React components.

Do Hooks cover all use cases for classes?

Our goal is for Hooks to cover all use cases for classes as soon as possible. There are no Hook equivalents to the uncommon `getSnapshotBeforeUpdate`, `getDerivedStateFromError` and `componentDidCatch` lifecycles yet, but we plan to add them soon.

Do Hooks replace render props and higher-order components?

Often, render props and higher-order components render only a single child. We think Hooks are a simpler way to serve this use case. There is still a place for both patterns (for example, a virtual scroller component might have a `renderItem` prop, or a visual container component might have its own DOM structure). But in most cases, Hooks will be sufficient and can help reduce nesting in your tree.

What do Hooks mean for popular APIs like Redux `connect()` and React Router?

You can continue to use the exact same APIs as you always have; they'll continue to work.

React Redux since v7.1.0 [supports Hooks API](#) and exposes hooks like `useDispatch` or `useSelector`.

React Router [supports hooks](#) since v5.1.

Other libraries might support hooks in the future too.

Do Hooks work with static typing?

Hooks were designed with static typing in mind. Because they're functions, they are easier to type correctly than patterns like higher-order components. The latest Flow and TypeScript React definitions include support for React Hooks.

Importantly, custom Hooks give you the power to constrain React API if you'd like to type them more strictly in some way. React gives you the primitives, but you can combine them in different ways than what we provide out of the box.

How to test components that use Hooks?

From React's point of view, a component using Hooks is just a regular component. If your testing solution doesn't rely on React internals, testing components with Hooks shouldn't be different from how you normally test components.

Note

[Testing Recipes](#) include many examples that you can copy and paste.

For example, let's say we have this counter component:

```
function Example() {
  const [count, setCount] = useState(0);
  useEffect(() => {
```



```

    document.title = `You clicked ${count} times`;
  });
  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>Click me</button>
    </div>
  );
}

```

We'll test it using React DOM. To make sure that the behavior matches what happens in the browser, we'll wrap the code rendering and updating it into `ReactTestUtils.act()` calls:

```

import React from 'react';
import ReactDOM from 'react-dom/client';
import { act } from 'react-dom/test-utils';
import Counter from './Counter';

let container;

beforeEach(() => {
  container = document.createElement('div');
  document.body.appendChild(container);
});

afterEach(() => {
  document.body.removeChild(container);
  container = null;
});

it('can render and update a counter', () => {
  // Test first render and effect
  act(() => {
    ReactDOM.createRoot(container).render(<Counter />);
  });
  const button = container.querySelector('button');
  const label = container.querySelector('p');
  expect(label.textContent).toBe('You clicked 0 times');
  expect(document.title).toBe('You clicked 0 times');

  // Test second render and effect
  act(() => {
    button.dispatchEvent(new MouseEvent('click', { bubbles: true }));
  });
  expect(label.textContent).toBe('You clicked 1 times');
  expect(document.title).toBe('You clicked 1 times');
});

```

The calls to `act()` will also flush the effects inside of them.

If you need to test a custom Hook, you can do so by creating a component in your test, and using your Hook from it. Then you can test the component you wrote.

To reduce the boilerplate, we recommend using [React Testing Library](#) which is designed to encourage writing tests that use your components as the end users do.

For more information, check out [Testing Recipes](#).

What exactly do the lint rules enforce?

We provide an [ESLint plugin](#) that enforces [rules of Hooks](#) to avoid bugs. It assumes that any function starting with "`use`" and a capital letter right after it is a Hook. We recognize this heuristic isn't perfect and there may be some false positives, but without an ecosystem-wide convention there is just no way to make Hooks work well — and longer names will discourage people from either adopting Hooks or following the convention.

In particular, the rule enforces that:

- Calls to Hooks are either inside a `PascalCase` function (assumed to be a component) or another `useSomething` function (assumed to be a custom Hook).
- Hooks are called in the same order on every render.

There are a few more heuristics, and they might change over time as we fine-tune the rule to balance finding bugs with avoiding false positives.

From Classes to Hooks

How do lifecycle methods correspond to Hooks?

- `constructor`: Function components don't need a constructor. You can initialize the state in the `useState` call. If computing the initial state is expensive, you can pass a function to `useState`.
- `getDerivedStateFromProps`: Schedule an update while rendering instead.
- `shouldComponentUpdate`: See `React.memo` below.
- `render`: This is the function component body itself.
- `componentDidMount`, `componentDidUpdate`, `componentWillUnmount`: The `useEffect` Hook can express all combinations of these (including less common cases).
- `getSnapshotBeforeUpdate`, `componentDidCatch` and `getDerivedStateFromError`: There are no Hook equivalents for these methods yet, but they will be added soon.

How can I do data fetching with Hooks?

Here is a [small demo](#) to get you started. To learn more, check out [this article](#) about data fetching with Hooks.

Is there something like instance variables?

Yes! The `useRef()` Hook isn't just for DOM refs. The "ref" object is a generic container whose `current` property is mutable and can hold any value, similar to an instance property on a class.

You can write to it from inside `useEffect`:

```
function Timer() {
  const intervalRef = useRef();
  useEffect(() => {
    const id = setInterval(() => {
      // ...
    });
    intervalRef.current = id;
    return () => {
      clearInterval(intervalRef.current);
    };
  });
  // ...
}
```

If we just wanted to set an interval, we wouldn't need the ref (`id` could be local to the effect), but it's useful if we want to clear the interval from an event handler:

```
// ...
function handleCancelClick() {
  clearInterval(intervalRef.current);
}
// ...
```

Conceptually, you can think of refs as similar to instance variables in a class. Unless you're doing [lazy initialization](#), avoid setting refs during rendering — this can lead to surprising behavior. Instead, typically you want to modify refs in event handlers and effects.

Should I use one or many state variables?

If you're coming from classes, you might be tempted to always call `useState()` once and put all state into a single object. You can do it if you'd like. Here is an example of a component that follows the mouse movement. We keep its position and size in the local state:

```
function Box() {
  const [state, setState] = useState({ left: 0, top: 0, width: 100, height: 100 });
  // ...
}
```

Now let's say we want to write some logic that changes `left` and `top` when the user moves their mouse. Note how we have to merge these fields into the previous state object manually:

```
// ...
useEffect(() => {
  function handleWindowMouseMove(e) {
    // Spreading "...state" ensures we don't "lose" width and height
    setState((state) => ({ ...state, left: e.pageX, top: e.pageY }));
  }
  // Note: this implementation is a bit simplified
  window.addEventListener('mousemove', handleWindowMouseMove);
  return () => window.removeEventListener('mousemove', handleWindowMouseMove);
}, []);
// ...
```

This is because when we update a state variable, we *replace* its value. This is different from `this.setState` in a class, which *merges* the updated fields into the object.

If you miss automatic merging, you could write a custom `useLegacyState` Hook that merges object state updates. However, **we recommend to split state into multiple state variables based on which values tend to change together**.

For example, we could split our component state into `position` and `size` objects, and always replace the `position` with no need for merging:

```
function Box() {
  const [position, setPosition] = useState({ left: 0, top: 0 });
  const [size, setSize] = useState({ width: 100, height: 100 });

  useEffect(() => {
    function handleWindowMouseMove(e) {
      setPosition({ left: e.pageX, top: e.pageY });
    }
    // ...
  });
}
```

Separating independent state variables also has another benefit. It makes it easy to later extract some related logic into a custom Hook, for example:

```
function Box() {
  const position = useWindowPosition();
  const [size, setSize] = useState({ width: 100, height: 100 });
  // ...
}

function useWindowPosition() {
  const [position, setPosition] = useState({ left: 0, top: 0 });
  useEffect(() => {
    // ...
  }, []);
  return position;
}
```

Note how we were able to move the `useState` call for the `position` state variable and the related effect into a custom Hook without changing their code. If all state was in a single object, extracting it would be more difficult.

Both putting all state in a single `useState` call, and having a `useState` call per each field can work. Components tend to be most readable when you find a balance between these two extremes, and group related state into a few independent state variables. If the state logic becomes complex, we recommend [managing it with a reducer](#) or a custom Hook.

Can I run an effect only on updates?

This is a rare use case. If you need it, you can [use a mutable ref](#) to manually store a boolean value corresponding to whether you are on the first or a subsequent render, then check that flag in your effect. (If you find yourself doing this often, you could

create a custom Hook for it.)

How to get the previous props or state?

There are two cases in which you might want to get previous props or state.

Sometimes, you need previous props to **clean up an effect**. For example, you might have an effect that subscribes to a socket based on the `userId` prop. If the `userId` prop changes, you want to unsubscribe from the *previous* `userId` and subscribe to the *next* one. You don't need to do anything special for this to work:

```
useEffect(() => {
  ChatAPI.subscribeToSocket(props.userId);
  return () => ChatAPI.unsubscribeFromSocket(props.userId);
}, [props.userId]);
```

In the above example, if `userId` changes from `3` to `4`, `ChatAPI.unsubscribeFromSocket(3)` will run first, and then `ChatAPI.subscribeToSocket(4)` will run. There is no need to get “previous” `userId` because the cleanup function will capture it in a closure.

Other times, you might need to **adjust state based on a change in props or other state**. This is rarely needed and is usually a sign you have some duplicate or redundant state. However, in the rare case that you need this pattern, you can [store previous state or props in state and update them during rendering](#).

We have previously suggested a custom Hook called `usePrevious` to hold the previous value. However, we've found that most use cases fall into the two patterns described above. If your use case is different, you can [hold a value in a ref](#) and manually update it when needed. Avoid reading and updating refs during rendering because this makes your component's behavior difficult to predict and understand.

Why am I seeing stale props or state inside my function?

Any function inside a component, including event handlers and effects, “sees” the props and state from the render it was created in. For example, consider code like this:

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

  function handleAlertClick() {
    setTimeout(() => {
      alert('You clicked on: ' + count);
    }, 3000);
  }

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>Click me</button>
      <button onClick={handleAlertClick}>Show alert</button>
    </div>
  );
}
```

If you first click “Show alert” and then increment the counter, the alert will show the `count` variable **at the time you clicked the “Show alert” button**. This prevents bugs caused by the code assuming props and state don't change.

If you intentionally want to read the *latest* state from some asynchronous callback, you could keep it in [a ref](#), mutate it, and read from it.

Finally, another possible reason you're seeing stale props or state is if you use the “dependency array” optimization but didn't correctly specify all the dependencies. For example, if an effect specifies `[]` as the second argument but reads `someProp` inside, it will keep “seeing” the initial value of `someProp`. The solution is to either remove the dependency array, or to fix it. Here's [how you can deal with functions](#), and here's [other common strategies](#) to run effects less often without incorrectly skipping dependencies.

Note

We provide an `exhaustive-deps` ESLint rule as a part of the `eslint-plugin-react-hooks` package. It warns when dependencies are specified incorrectly and suggests a fix.

How do I implement `getDerivedStateFromProps`?

While you probably don't need it, in rare cases that you do (such as implementing a `<Transition>` component), you can update the state right during rendering. React will re-run the component with updated state immediately after exiting the first render so it wouldn't be expensive.

Here, we store the previous value of the `row` prop in a state variable so that we can compare:

```
function ScrollView({ row }) {
  const [isScrollingDown, setIsScrollingDown] = useState(false);
  const [prevRow, setPrevRow] = useState(null);

  if (row !== prevRow) {
    // Row changed since last render. Update isScrollingDown.
    setIsScrollingDown(prevRow !== null && row > prevRow);
    setPrevRow(row);
  }

  return `Scrolling down: ${isScrollingDown}`;
}
```

This might look strange at first, but an update during rendering is exactly what `getDerivedStateFromProps` has always been like conceptually.

Is there something like `forceUpdate`?

Both `useState` and `useReducer` Hooks bail out of updates if the next value is the same as the previous one. Mutating state in place and calling `setState` will not cause a re-render.

Normally, you shouldn't mutate local state in React. However, as an escape hatch, you can use an incrementing counter to force a re-render even if the state has not changed:

```
const [ignored, forceUpdate] = useReducer((x) => x + 1, 0);

function handleClick() {
  forceUpdate();
}
```

Try to avoid this pattern if possible.

Can I make a ref to a function component?

While you shouldn't need this often, you may expose some imperative methods to a parent component with the `useImperativeHandle` Hook.

How can I measure a DOM node?

One rudimentary way to measure the position or size of a DOM node is to use a callback ref. React will call that callback whenever the ref gets attached to a different node. Here is a small demo:

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

  const measuredRef = useCallback((node) => {
    if (node !== null) {
      setHeight(node.getBoundingClientRect().height);
    }
  }, []);

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

```

    );
  }

```

We didn't choose `useRef` in this example because an object ref doesn't notify us about *changes* to the current ref value. 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.

Note that we pass `[]` as a dependency array to `useCallback`. This ensures that our ref callback doesn't change between the renders, and so React won't call it unnecessarily.

In this example, the callback ref will be called only when the component mounts and unmounts, since the rendered `<h1>` component stays present throughout any rerenders. If you want to be notified any time a component resizes, you may want to use `ResizeObserver` or a third-party Hook built on it.

If you want, you can extract this logic into a reusable Hook:

```

function MeasureExample() {
  const [rect, ref] = useClientRect();
  return (
    <>
      <h1 ref={ref}>Hello, world</h1>
      {rect !== null && <h2>The above header is {Math.round(rect.height)}px tall</h2>}
    </>
  );
}

function useClientRect() {
  const [rect, setRect] = useState(null);
  const ref = useCallback((node) => {
    if (node !== null) {
      setRect(node.getBoundingClientRect());
    }
  }, []);
  return [rect, ref];
}

```

What does `const [thing, setThing] = useState()` mean?

If you're not familiar with this syntax, check out the [explanation](#) in the State Hook documentation.

Performance Optimizations

Can I skip an effect on updates?

Yes. See [conditionally firing an effect](#). Note that forgetting to handle updates often introduces bugs, which is why this isn't the default behavior.

Is it safe to omit functions from the list of dependencies?

Generally speaking, no.

```

function Example({ someProp }) {
  function doSomething() {
    console.log(someProp);
  }

  useEffect(() => {
    doSomething();
  }, []); // 🚫 This is not safe (it calls `doSomething` which uses `someProp`)
}

```

It's difficult to remember which props or state are used by functions outside of the effect. This is why **usually you'll want to declare functions needed by an effect *inside* of it**. Then it's easy to see what values from the component scope that effect depends on:

```
function Example({ someProp }) {
  useEffect(() => {
    function doSomething() {
      console.log(someProp);
    }

    doSomething();
  }, [someProp]); // ✅ OK (our effect only uses `someProp`)
}
```

If after that we still don't use any values from the component scope, it's safe to specify `[]`:

```
useEffect(() => {
  function doSomething() {
    console.log('hello');
  }

  doSomething();
}, []); // ✅ OK in this example because we don't use *any* values from component scope
```

Depending on your use case, there are a few more options described below.

Note

We provide the [exhaustive-deps](#) ESLint rule as a part of the [eslint-plugin-react-hooks](#) package. It helps you find components that don't handle updates consistently.

Let's see why this matters.

If you specify a [list of dependencies](#) as the last argument to `useEffect`, `useLayoutEffect`, `useMemo`, `useCallback`, or `useImperativeHandle`, it must include all values that are used inside the callback and participate in the React data flow. That includes props, state, and anything derived from them.

It is **only** safe to omit a function from the dependency list if nothing in it (or the functions called by it) references props, state, or values derived from them. This example has a bug:

```
function ProductPage({ productId }) {
  const [product, setProduct] = useState(null);

  async function fetchProduct() {
    const response = await fetch('http://myapi/product/' + productId); // Uses productId prop
    const json = await response.json();
    setProduct(json);
  }

  useEffect(() => {
    fetchProduct();
  }, []); // ❌ Invalid because `fetchProduct` uses `productId` // ...
}
```

The recommended fix is to move that function *inside* of your effect. That makes it easy to see which props or state your effect uses, and to ensure they're all declared:

```
function ProductPage({ productId }) {
  const [product, setProduct] = useState(null);

  useEffect(() => {
    // By moving this function inside the effect, we can clearly see the values it uses.
    async function fetchProduct() {
      const response = await fetch('http://myapi/product/' + productId);
      const json = await response.json();
      setProduct(json);
    }
    fetchProduct();
  }, [productId]); // ✅ Valid because our effect only uses productId // ...
}
```

This also allows you to handle out-of-order responses with a local variable inside the effect:

```
useEffect(() => {
  let ignore = false;
  async function fetchProduct() {
    const response = await fetch('http://myapi/product/' + productId);
    const json = await response.json();
    if (!ignore) setProduct(json);
  }
  fetchProduct();
  // ...
}
```

```

    fetchProduct();
    return () => {
      ignore = true;
    };
  }, [productId]);

```

We moved the function inside the effect so it doesn't need to be in its dependency list.

Tip

Check out [this small demo](#) and [this article](#) to learn more about data fetching with Hooks.

If for some reason you *can't* move a function inside an effect, there are a few more options:

- **You can try moving that function outside of your component.** In that case, the function is guaranteed to not reference any props or state, and also doesn't need to be in the list of dependencies.
- If the function you're calling is a pure computation and is safe to call while rendering, you may **call it outside of the effect instead**, and make the effect depend on the returned value.
- As a last resort, you can **add a function to effect dependencies but wrap its definition** into the `useCallback` Hook. This ensures it doesn't change on every render unless *its own* dependencies also change:

```

function ProductPage({ productId }) {
  // ✅ Wrap with useCallback to avoid change on every render
  const fetchProduct = useCallback(() => {
    // ... Does something with productId ...
  }, [productId]);
  // ✅ All useCallback dependencies are specified
  return (
    <ProductDetails fetchProduct={fetchProduct} />
  );
}

function ProductDetails({ fetchProduct }) {
  useEffect(() => {
    fetchProduct();
  }, [fetchProduct]); // ✅ All useEffect dependencies are specified
  // ...
}

```

Note that in the above example we **need** to keep the function in the dependencies list. This ensures that a change in the `productId` prop of `ProductPage` automatically triggers a refetch in the `ProductDetails` component.

What can I do if my effect dependencies change too often?

Sometimes, your effect may be using state that changes too often. You might be tempted to omit that state from a list of dependencies, but that usually leads to bugs:

```

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

  useEffect(() => {
    const id = setInterval(() => {
      setCount(count + 1); // This effect depends on the `count` state
    }, 1000);
    return () => clearInterval(id);
  }, []); // 🚫 Bug: `count` is not specified as a dependency
  return (
    <h1>{count}</h1>
  );
}

```

The empty set of dependencies, `[]`, means that the effect will only run once when the component mounts, and not on every re-render. The problem is that inside the `setInterval` callback, the value of `count` does not change, because we've created a closure with the value of `count` set to `0` as it was when the effect callback ran. Every second, this callback then calls `setCount(0 + 1)`, so the count never goes above 1.

Specifying `[count]` as a list of dependencies would fix the bug, but would cause the interval to be reset on every change. Effectively, each `setInterval` would get one chance to execute before being cleared (similar to a `setTimeout`.) That may not be

desirable. To fix this, we can use the [functional update form of `setCount`](#). It lets us specify *how* the state needs to change without referencing the *current* state:

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

  useEffect(() => {
    const id = setInterval(() => {
      setCount((c) => c + 1); // ✅ This doesn't depend on `count` variable outside
    }, 1000);
    return () => clearInterval(id);
  }, []); // ✅ Our effect doesn't use any variables in the component scope
  return <h1>{count}</h1>;
}
```

(The identity of the `setCount` function is guaranteed to be stable so it's safe to omit.)

Now, the `setInterval` callback executes once a second, but each time the inner call to `setCount` can use an up-to-date value for `count` (called `c` in the callback here.)

In more complex cases (such as if one state depends on another state), try moving the state update logic outside the effect with the [`useReducer` Hook](#). [This article](#) offers an example of how you can do this. **The identity of the `dispatch` function from `useReducer` is always stable** — even if the reducer function is declared inside the component and reads its props.

As a last resort, if you want something like `this` in a class, you can [use a ref](#) to hold a mutable variable. Then you can write and read to it. For example:

```
function Example(props) {
  // Keep latest props in a ref.
  const latestProps = useRef(props);
  useEffect(() => {
    latestProps.current = props;
  });
  useEffect(() => {
    function tick() {
      // Read latest props at any time
      console.log(latestProps.current);
    }

    const id = setInterval(tick, 1000);
    return () => clearInterval(id);
  }, []); // This effect never re-runs
}
```

Only do this if you couldn't find a better alternative, as relying on mutation makes components less predictable. If there's a specific pattern that doesn't translate well, [file an issue](#) with a runnable example code and we can try to help.

How do I implement `shouldComponentUpdate`?

You can wrap a function component with `React.memo` to shallowly compare its props:

```
const Button = React.memo((props) => {
  // your component
});
```

It's not a Hook because it doesn't compose like Hooks do. `React.memo` is equivalent to `PureComponent`, but it only compares props. (You can also add a second argument to specify a custom comparison function that takes the old and new props. If it returns true, the update is skipped.)

`React.memo` doesn't compare state because there is no single state object to compare. But you can make children pure too, or even [optimize individual children with `useMemo`](#).

How to memoize calculations?

The `useMemo` Hook lets you cache calculations between multiple renders by “remembering” the previous computation:

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

This code calls `computeExpensiveValue(a, b)`. But if the dependencies `[a, b]` haven't changed since the last value, `useMemo` skips calling it a second time and simply reuses the last value it returned.

Remember that the function passed to `useMemo` runs during rendering. Don't do anything there that you wouldn't normally do while rendering. For example, side effects belong in `useEffect`, not `useMemo`.

You may rely on `useMemo` as a performance optimization, not as a semantic guarantee. In the future, React may choose to “forget” some previously memoized values and recalculate them on next render, e.g. to free memory for offscreen components. Write your code so that it still works without `useMemo` — and then add it to optimize performance. (For rare cases when a value must *never* be recomputed, you can [lazily initialize](#) a ref.)

Conveniently, `useMemo` also lets you skip an expensive re-render of a child:

```
function Parent({ a, b }) {
  // Only re-rendered if `a` changes:
  const child1 = useMemo(() => <Child1 a={a} />, [a]);
  // Only re-rendered if `b` changes:
  const child2 = useMemo(() => <Child2 b={b} />, [b]);
  return (
    <>
      {child1}
      {child2}
    </>
  );
}
```

Note that this approach won't work in a loop because Hook calls can't be placed inside loops. But you can extract a separate component for the list item, and call `useMemo` there.

How to create expensive objects lazily?

`useMemo` lets you [memoize an expensive calculation](#) if the dependencies are the same. However, it only serves as a hint, and doesn't *guarantee* the computation won't re-run. But sometimes you need to be sure an object is only created once.

The first common use case is when creating the initial state is expensive:

```
function Table(props) {
  // ⚠ createRows() is called on every render
  const [rows, setRows] = useState(createRows(props.count));
  // ...
}
```

To avoid re-creating the ignored initial state, we can pass a **function** to `useState`:

```
function Table(props) {
  // ✅ createRows() is only called once
  const [rows, setRows] = useState(() => createRows(props.count));
  // ...
}
```

React will only call this function during the first render. See the [useState API reference](#).

You might also occasionally want to avoid re-creating the `useRef()` initial value. For example, maybe you want to ensure some imperative class instance only gets created once:

```
function Image(props) {
  // ⚠ IntersectionObserver is created on every render
  const ref = useRef(new IntersectionObserver(onIntersect));
  // ...
}
```

`useRef` **does not** accept a special function overload like `useState`. Instead, you can write your own function that creates and sets it lazily:

```
function Image(props) {
  const ref = useRef(null);

  // ✅ IntersectionObserver is created lazily once
  function getObserver() {
    if (ref.current === null) {
      ref.current = new IntersectionObserver(onIntersect);
    }
  }
}
```

```

    }
    return ref.current;
  }

  // When you need it, call getObserver()
  // ...
}

```

This avoids creating an expensive object until it's truly needed for the first time. If you use Flow or TypeScript, you can also give `getObserver()` a non-nullable type for convenience.

Are Hooks slow because of creating functions in render?

No. In modern browsers, the raw performance of closures compared to classes doesn't differ significantly except in extreme scenarios.

In addition, consider that the design of Hooks is more efficient in a couple ways:

- Hooks avoid a lot of the overhead that classes require, like the cost of creating class instances and binding event handlers in the constructor.
- **Idiomatic code using Hooks doesn't need the deep component tree nesting** that is prevalent in codebases that use higher-order components, render props, and context. With smaller component trees, React has less work to do.

Traditionally, performance concerns around inline functions in React have been related to how passing new callbacks on each render breaks `shouldComponentUpdate` optimizations in child components. Hooks approach this problem from three sides.

- The `useCallback` Hook lets you keep the same callback reference between re-renders so that `shouldComponentUpdate` continues to work:

```

// Will not change unless `a` or `b` changes
const memoizedCallback = useCallback(() => {
  doSomething(a, b);
}, [a, b]);

```

- The `useMemo` Hook makes it easier to control when individual children update, reducing the need for pure components.
- Finally, the `useReducer` Hook reduces the need to pass callbacks deeply, as explained below.

How to avoid passing callbacks down?

We've found that most people don't enjoy manually passing callbacks through every level of a component tree. Even though it is more explicit, it can feel like a lot of "plumbing".

In large component trees, an alternative we recommend is to pass down a `dispatch` function from `useReducer` via context:

```

const TodosDispatch = React.createContext(null);

function TodosApp() {
  // Note: `dispatch` won't change between re-renders
  const [todos, dispatch] = useReducer(todosReducer);
  return (
    <TodosDispatch.Provider value={dispatch}>
      <DeepTree todos={todos} />
    </TodosDispatch.Provider>
  );
}

```

Any child in the tree inside `TodosApp` can use the `dispatch` function to pass actions up to `TodosApp`:

```

function DeepChild(props) {
  // If we want to perform an action, we can get dispatch from context.
  const dispatch = useContext(TodosDispatch);
  function handleClick() {
    dispatch({ type: 'add', text: 'hello' });
  }

  return <button onClick={handleClick}>Add todo</button>;
}

```

This is both more convenient from the maintenance perspective (no need to keep forwarding callbacks), and avoids the callback problem altogether. Passing `dispatch` down like this is the recommended pattern for deep updates.

Note that you can still choose whether to pass the application state down as props (more explicit) or as context (more convenient for very deep updates). If you use context to pass down the state too, use two different context types — the `dispatch` context never changes, so components that read it don't need to rerender unless they also need the application state.

How to read an often-changing value from `useCallback` ?

Note

We recommend to pass `dispatch` down in context rather than individual callbacks in props. The approach below is only mentioned here for completeness and as an escape hatch.

In some rare cases you might need to memoize a callback with `useCallback` but the memoization doesn't work very well because the inner function has to be re-created too often. If the function you're memoizing is an event handler and isn't used during rendering, you can use ref as an instance variable, and save the last committed value into it manually:

```
function Form() {
  const [text, updateText] = useState('');
  const textRef = useRef();

  useEffect(() => {
    textRef.current = text; // Write it to the ref
  });

  const handleSubmit = useCallback(() => {
    const currentText = textRef.current; // Read it from the ref
    alert(currentText);
  }, [textRef]); // Don't recreate handleSubmit like [text] would do

  return (
    <>
      <input value={text} onChange={(e) => updateText(e.target.value)} />
      <ExpensiveTree onSubmit={handleSubmit} />
    </>
  );
}
```

This is a rather convoluted pattern but it shows that you can do this escape hatch optimization if you need it. It's more bearable if you extract it to a custom Hook:

```
function Form() {
  const [text, updateText] = useState('');
  // Will be memoized even if `text` changes:
  const handleSubmit = useEventCallback(() => {
    alert(text);
  }, [text]);

  return (
    <>
      <input value={text} onChange={(e) => updateText(e.target.value)} /> <ExpensiveTree onSubmit={handleSubmit} />
    </>
  );
}

function useEventCallback(fn, dependencies) {
  const ref = useRef(() => {
    throw new Error('Cannot call an event handler while rendering.');
```

In either case, we **don't recommend this pattern** and only show it here for completeness. Instead, it is preferable to avoid passing callbacks deep down.

Under the Hood

How does React associate Hook calls with components?

React keeps track of the currently rendering component. Thanks to the Rules of Hooks, we know that Hooks are only called from React components (or custom Hooks — which are also only called from React components).

There is an internal list of “memory cells” associated with each component. They're just JavaScript objects where we can put some data. When you call a Hook like `useState()`, it reads the current cell (or initializes it during the first render), and then moves the pointer to the next one. This is how multiple `useState()` calls each get independent local state.

What is the prior art for Hooks?

Hooks synthesize ideas from several different sources:

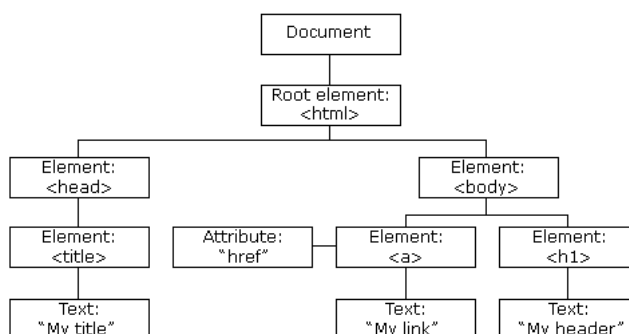
- Our old experiments with functional APIs in the react-future repository.
- React community's experiments with render prop APIs, including Ryan Florence's Reactions Component.
- Dominic Gannaway's adopt keyword proposal as a sugar syntax for render props.
- State variables and state cells in DisplayScript.
- Reducer components in ReasonReact.
- Subscriptions in Rx.
- Algebraic effects in Multicore OCaml.

Sebastian Markbåge came up with the original design for Hooks, later refined by Andrew Clark, Sophie Alpert, Dominic Gannaway, and other members of the React team.

Virtual DOM trong ReactJs?

Khi làm việc với ReactJs, sớm hay muộn chúng ta cũng sẽ nghe đến Virtual DOM. Nghe DOM thì có vẻ quen quen, vậy thêm Virtual vào thì khác gì? Hoặc bạn được nói là Virtual DOM ngon lắm, nhanh lắm thì có thực sự đúng không?

Trước tiên, DOM là gì?



DOM là tên gọi tắt của **Document Object Model** (Mô hình Đối tượng Tài liệu), là một chuẩn được định nghĩa bởi W3C dùng để truy xuất và thao tác trên code HTML hay XML bằng các ngôn ngữ lập trình thông dịch (scripting language) như Javascript.

DOM giúp thao tác với dữ liệu theo mô hình hướng đối tượng do các phần tử trong **DOM** có cấu trúc được định nghĩa thành các *đối tượng*, *phương thức*, *thuộc tính* để có thể truy xuất dễ dàng. Chúng được coi như các *node* và được biểu diễn dưới dạng **DOM Tree**.

Trong khi HTML là 1 đoạn code, **DOM** là một thể hiện trừu tượng của đoạn code đó trong bộ nhớ.

<code><html></code>	-> document node
<code><head></code>	-> element node - head
<code><title></code>	
HTML DOM	-> text node
<code></title></code>	-> element node - title
<code></head></code>	
<code><body></code>	-> element node - body
<code></body></code>	
<code></html></code>	

HTML DOM cung cấp API để duyệt và chỉnh sửa các *node*. Nó chứa các phương thức như `getElementById` hay `removeChild`.

```
var content = document.getElementById("myContent");
content.parentNode.removeChild(item);
```

Vì vậy với lập trình viên web, khi nắm rõ kiến thức về **DOM** và khả năng thao tác với **DOM** cũng có nghĩa là có sức mạnh thay đổi mọi thứ của trang web :v

DOM ngon vậy thì sao lại có Virtual DOM?

Chúng ta có thể thấy hình ở trên,

HTML DOM

được cấu trúc dạng cây. Thực sự rất ngon vì có thể duyệt cây rất dễ dàng. Thật là đen, ở đây không phải cứ dễ là tốc độ nhanh.

List of Items

List Item 1
List Item 2
List Item 3
List Item 4
List Item 5
List Item 6

Re-rendered List

List Item 1
List Item 2
List Item 3
List Item 4
List Item 5
List Item 6

Tuy nhiên đừng hiểu nhầm việc đọc và ghi vào **DOM** của trình duyệt là chậm. Điều này không đúng. **DOM** nhanh. Việc cập nhật các *node* trong **DOM** không mất nhiều thời gian hơn việc thiết lập một thuộc tính trên một đối tượng JavaScript. Đó là một hoạt động đơn giản.

Điều chậm ở đây là layout mà các trình duyệt phải làm bất cứ khi nào DOM thay đổi. Mỗi khi DOM thay đổi, trình duyệt cần phải tính toán lại CSS, thực hiện dựng lại trang web. Đây là việc cần có thời gian.

Và hiện nay **DOM Tree** thực sự rất lớn. Việc những trang web SPA ngày càng được phát triển mạnh và đa dạng, vì vậy việc sửa đổi **DOM Tree** là liên tục không ngừng và sửa đổi rất nhiều.

Xem xét một **DOM** được tạo bởi hàng nghìn `div`, Có quá nhiều phương thức để xử lý các event `click, submit, ...` Điển hình của việc xử lý event trong *jQuery* sẽ là:

- Tìm tất cả các *node* liên quan đến event

- Cập nhật nó nếu thấy cần thiết

Chúng ta gặp 2 vấn đề:

- Thực sự rất khó để quản lý. Tưởng tượng xem nếu phải chỉnh sửa một đoạn xử lý event mà không nắm được context thì bạn sẽ phải bơi thật sâu trong code để có thể xem nó đang làm gì. Tốn thời gian và rủi ro cao.
- Nó không hiệu quả. Có nhất thiết cứ phải đi tìm tất cả những gì liên quan không? Hay có thể thông minh hơn bằng cách chỉ tìm `node` nào cần cập nhật.

ReactJs đến và cho chúng ta giải pháp, thay vì xử lý **DOM Tree** thủ công, chúng ta định nghĩa các `component` trông giống giống thể còn ReactJs sẽ thực hiện công việc ở tầng thấp hơn. Thực chất, công việc xử lý sẽ được **HTML DOM API** đảm nhiệm ở các tầng đó. Đây chính xác là cách mà **Virtual DOM** hoạt động.

Virtual DOM

Virtual DOM không được tạo ra bởi React tuy nhiên nó được React sử dụng và cung cấp miễn phí.

Một cách tổng quát thì nó là một định dạng dữ liệu JavaScript nhẹ được dùng để thể hiện nội dung của DOM tại một thời điểm nhất định nào đó. Nó có tất cả các thuộc tính giống như **DOM** nhưng không có khả năng tương tác lên màn hình như **DOM**.

Bạn có thể tưởng tượng, ở **DOM** có thể `div` và các thẻ `p` ở trong, ReactJs sử dụng **Virtual DOM** bằng cách tạo ra các `object` `React.div` và `React.p` và khi tương tác, ta sẽ tương tác qua các object đó một cách nhanh chóng mà không phải đụng tới **DOM** hay **DOM API** của nó.

Đây là lí do tại sao JSX của code ReactJs có thể trông như code HTML thuần túy ==))

```
var CommentBox = React.createClass({
  render: function() {
    return (
      <div className="text">
        Syntax like HTML
      </div>
    );
  }
});
```

Trong hầu hết các trường hợp, khi bạn có đoạn code HTML và muốn chuyển nó vào ReactJs, tất cả những gì bạn phải làm là:

- Gõ code HTML trong render
- Thay thế `class` thành `className`

Virtual DOM

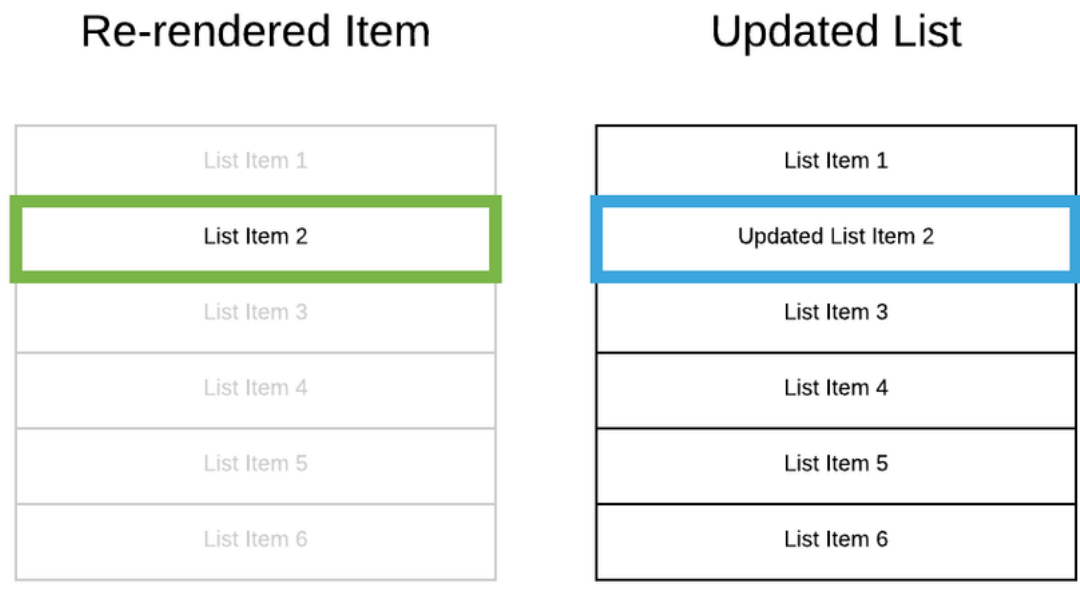
sử dụng **key**, **ref** mà ở **DOM** không có và **Virtual DOM** được tạo mới sau mỗi lần **render lại**.

Tuy nhiên, sự đặc biệt của **Virtual DOM** nằm ở *Snapshots & Diffing* Như giải thích ở trước đó, cách hoạt động của **Virtual DOM** trong React đó là:

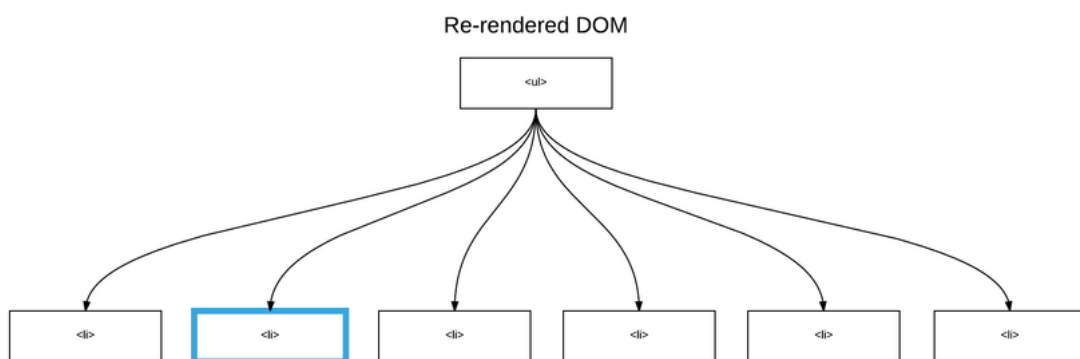
React lấy một *snapshot* của **Virtual DOM** (có thể hiểu là bản ghi trạng thái ngay lúc đó) ngay trước khi áp dụng bất kỳ bản cập nhật nào. Sau đó, nó sử dụng `snapshot` này để so sánh với một **Virtual DOM** được cập nhật trước khi thực hiện các thay đổi.



Khi cập nhật được cấp cho **Virtual DOM**, quá trình tiếp theo React sử dụng thuật toán `Diffing` để so sánh và đối chiếu để biết được sự cập nhật được diễn ra ở đâu sau đó cập nhật nó mà bỏ qua những elements không liên quan.



Chỉ những đối tượng này được cập nhật trên **DOM** và các thay đổi trên **DOM** vừa rồi sẽ làm cho màn hình thay đổi.



Lợi ích của Virtual Dom

Việc tách biệt logic liên quan đến việc rendering ra khỏi DOM cho phép React chạy được trên nhiều môi trường khác nhau thay vì chỉ một môi trường duy nhất là trình duyệt (React Native hoặc SSR - Server-Side Rendering cho cả React). Chúng ta biết rằng Virtual DOM trên thực tế chỉ là một JavaScript object do đó chúng ta chỉ cần một môi trường cho chép chạy JavaScript và việc sử dụng Virtual DOM là hoàn toàn khả thi. Một số ví dụ có thể nói đến ở đây là: SSR sử dụng NodeJS hoặc embedded JavaScript khi xây dựng các ứng dụng native cho nền tảng Web cũng như Mobile. Công việc của chúng ta là thay đổi generation algorithm cho từng môi trường cụ thể nhằm thu được các output mong muốn cho từng môi trường đó.

Sử dụng Virtual DOM cho phép chúng ta tính toán các thay đổi trên đó (just JavaScript) và áp dụng đồng thời các thay đổi đó lên Actual DOM khi cần thiết. Phương pháp này sẽ hiệu quả hơn khá nhiều so với việc access Actual DOM element (sử dụng jQuery chẳng hạn) và tính toán các thay đổi trên đó.

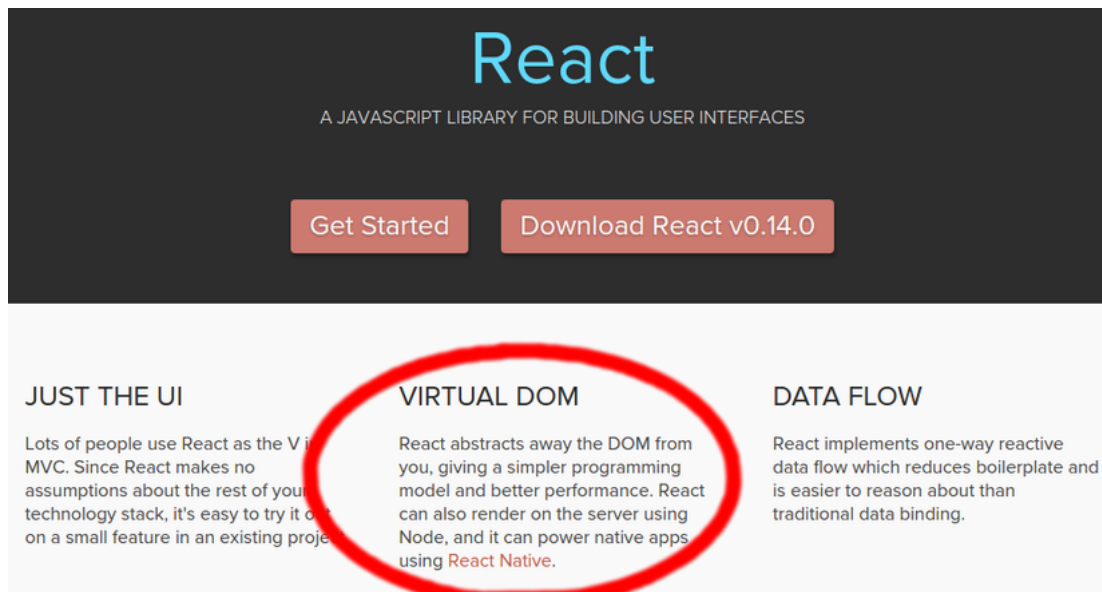
Một số vấn đề

Thực sự Virtual DOM trong ReactJs nhanh hơn nhiều so với DOM?

Bản thân **Virtual DOM** thực sự rất nhanh, nó có thể tìm kiếm, cập nhật và xóa bỏ các **elements** từ **Virtual DOM Tree** một cách nhanh chóng. Tuy nhiên, ở **DOM** thì những công việc đó cũng rất nhanh, chỉ là việc bố cục và thể hiện các elements của **DOM** mới là điều chậm. Nhưng React **Virtual DOM** không nhanh hơn.

Một lợi ích gắn liền khi sử dụng React là chúng ta có thể kiểm soát việc *re-render* của các **component** bằng cách sử dụng phương thức `shouldComponentUpdate` và `setState`.

Vậy sử dụng Virtual DOM có làm tăng performance cho ứng dụng không?



Có thời điểm React ghi rằng sử dụng Virtual DOM sẽ làm tăng performance. Tuy nhiên giờ đã không còn nữa mà chỉ đề cao sự tiện lợi cho nhà phát triển

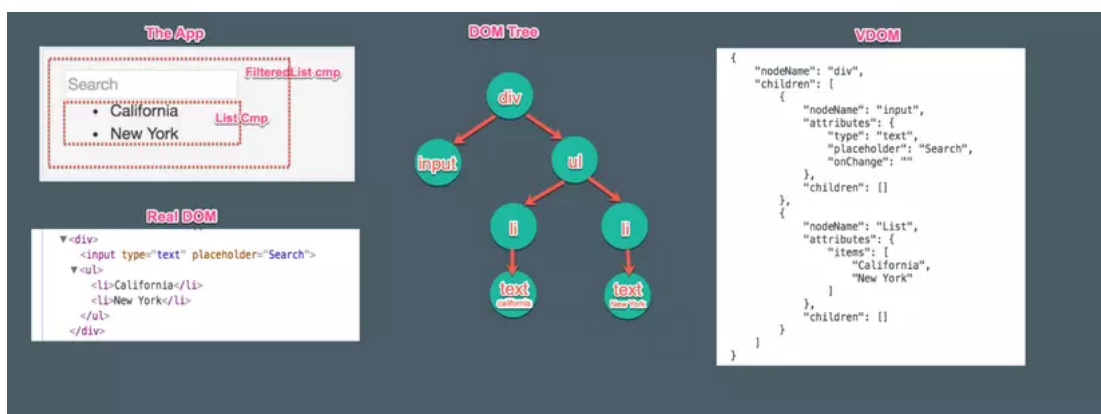
Trong thời kì đầu của **Virtual DOM** (đặc biệt là React), một số người nổi tiếng có nói rằng **Virtual DOM** giúp cập nhật **DOM** nhanh chóng. Như chúng ta có thể thấy được, điều này không khả thi về mặt kỹ thuật. Cập nhật **DOM** phải được tối ưu hóa trong mã gốc của trình duyệt chứ không có phép màu nào có thể giải quyết vấn đề này cả.

Reconciliation in ReactJS

1. Reconciliation trong React

a. Virtual DOM Tree

Nếu đã từng làm việc với **ReactJS** hoặc **VueJS** thì chắc hẳn bạn đã từng nghe đến khái niệm **Virtual DOM**. Thực tế thì **Virtual DOM** là một kiểu Object trong Javascript mà React sử dụng để biểu diễn DOM thật. Cụ thể thì bạn có thể nhìn hình dưới đây:



Như bạn thấy ở từng phần:

- **The App**: là thứ mà chúng ta sẽ nhìn thấy ở trên trình duyệt WEB.
- **Real DOM**: là phần DOM thật mà chúng ta có thể nhìn thấy bằng cách inspect trong dev tool trên trình duyệt hoặc view-source của trang web.
- **DOM Tree**: Là biểu diễn lại DOM của chúng ta dưới dạng "cây".
- **VDOM**: hay chính là phần **Virtual DOM** của chúng ta được biểu diễn dưới dạng là **Javascript Object**.

Phần **Javascript Object** mà bạn thấy trên hình trên nó còn có thể hiểu là **Virtual DOM Tree** tương ứng với việc **DOM Tree** của **Real DOM**. Trong **ReactJS** thì **Virtual DOM Tree** được cấu thành bởi các **Element** - là đơn vị nhỏ nhất trong **React**. Các **Element** này thực tế cũng đơn giản là một **Javascript Object**. Ví dụ:

```
// DOM thật
<h1 class="title">This is h1 tag</h1>

// React
{
  type: 'h1',
  props: {
    className: 'title',
    children: 'This is h1 tag'
  }
}
```

Như các bạn thấy trong ví dụ trên ta có một **Real DOM** là thẻ **<h1>** với class là **title** và nội dung bên trong là **This is h1 tag**. Thì ứng với nó chính là **React Element** có cấu trúc như bạn thấy bao gồm:

- type: 1 - ứng với thẻ h1
- props: danh sách các thuộc tính bao gồm **className** ứng với **class** bên DOM thật và phần **children** ứng với phần nội dung nằm trong thẻ đó.

Tương tự thì đối với các **Real DOM** phức tạp hơn như dạng lồng nhau thì **ReactJS** cũng biểu diễn lần lượt thành các **Element** như sau:

```
// DOM thật
<div class='container'>
  <h1 class='title'>This is title</h1>
  <p class='sub-title'>This is subtitle</p>
</div>

// React
{
  type: 'div',
  props: {
    className: 'container',
    children: [
      {
        type: 'h1',
        props: {
          className: 'title',
          children: 'This is title'
        }
      },
      {
        type: 'p',
        props: {
          className: 'sub-title',
          children: 'This is subtitle'
        }
      }
    ]
  }
}
```

Các **Javascript Object** mà mình cho các bạn thấy trong 2 ví dụ nói trên thực chất nó nằm kết quả mà chúng ta thu được khi **React** thực hiện việc **render** đã được loại bỏ đi bớt một vài thuộc tính..

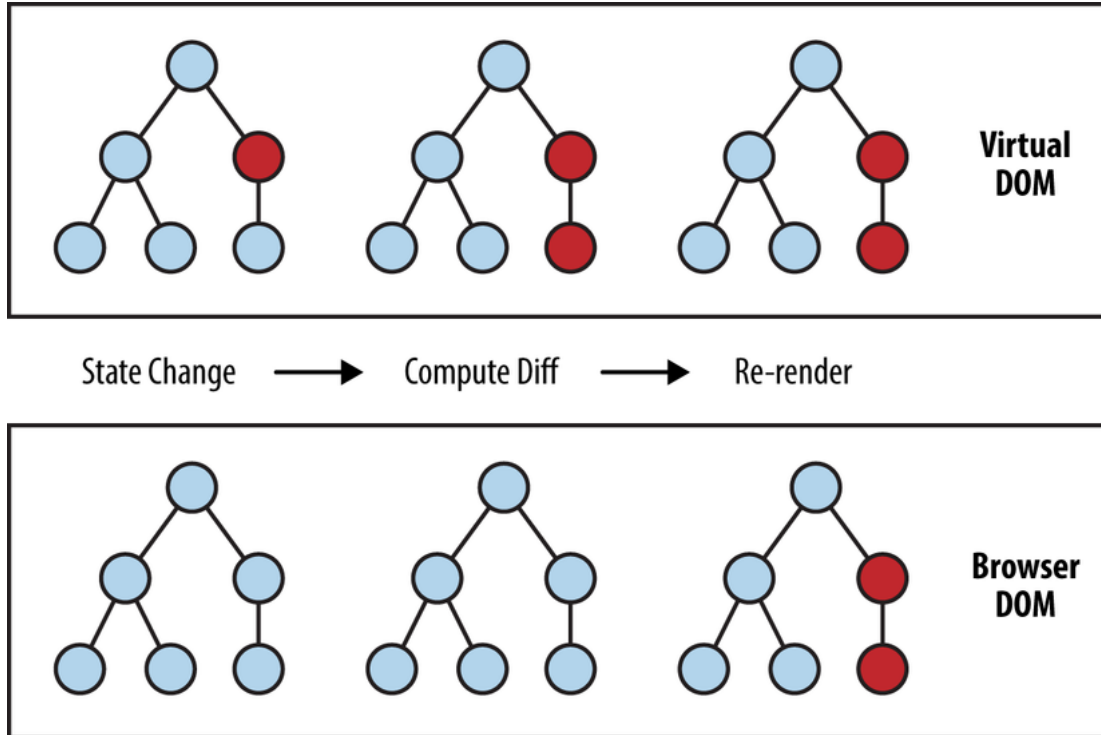
```
Object {type: "h1", key: null, ref: null, props: Object, _owner: null...}
  type: "h1"
  key: null
  ref: null
  props: Object
  className: "title"
```

```
children: "This is title"
_owner: null
_store: Object
```

Bạn có thể xem phần `console` của 2 ví dụ trên để thấy được kết quả của việc `render` như mình nói ở trên [ví dụ 1](#), [ví dụ 2](#).

b. Cập nhật DOM

Trong `React` mỗi khi component của chúng ta bị thay đổi về `state` hoặc `props` sẽ dẫn tới việc nó thực hiện việc `re-render` lại chính component nó và các component con của nó. Việc `re-render` này thực chất là chạy lại hàm `render` để thu được một `JavaScript Object` mới có dạng như mình đã đề cập đến ở phần trên. Khi thực hiện việc cập nhật cho DOM thật thì `React` sẽ đi qua các bước cơ bản như sau:



- Đầu tiên `React` sẽ thực hiện việc tạo ra một `Virtual DOM Tree` mới
- Thực hiện việc so sánh `Virtual DOM Tree` mới được tạo ra và `Virtual DOM Tree` ngay trước đó để xác định các vị trí cần thay đổi
- Tiến hành cập nhật `Real DOM`.

c. Diff Algorithm (Reconciliation)

Đây chính là phần nội dung chính mà mình muốn đề cập với các bạn và cụ thể đó là về cách mà `React` thực hiện việc so sánh 2 virtual DOM tree mới và cũ. Ở đây mình sẽ không nói về chi tiết từng bước từng bước mà `React` thực hiện việc so sánh này trên thực tế mà sẽ chỉ trình bày cho các bạn hiểu cơ bản về cách hoạt động của nó.

Khi thực hiện việc so sánh 2 Virtual DOM tree mới và cũ để update thì thuật toán của `React` sẽ cần đưa ra quyết định là khi nào tái sử dụng lại `Element` đang có và khi nào cần tạo mới `Element` đó. Nếu bỏ qua việc tái sử dụng lại `Element` đang có thì sẽ dẫn đến vấn đề về hiệu năng vì `React` sẽ tiến hành tạo mới lại toàn bộ các `Element` mà trong khi nó có thể tái sử dụng lại và chỉ cần cập nhật thuộc tính hoặc phần children của `Element` đó. Ta sẽ xét ví dụ như sau ta có 1 cái button và class của cái button đó bị thay đổi:

```
// Old
<button className="blue" />
{
  type: 'button',
  props: { className: 'blue' }
}

// New
```

```
<button className="red" />
{
  type: 'button',
  props: { className: 'red' }
}
```

Như các bạn thấy ở trên thì button của chúng ta chỉ bị thay đổi về phần `className` vì thế trong trường hợp này nếu React tạo lại `<button>` này thì nó sẽ dẫn đến việc lãng phí tài nguyên thay vào đó nó chỉ cần update đúng phần `className`. Việc quyết định tạo mới hay update lại sẽ được đưa ra bằng cách so sánh type giữa 2 `Element` nằm ở vị trí tương ứng với nhau trong `Virtual DOM Tree`. Bạn có thể hiểu cơ bản giả sử ta có 2 mảng lần lượt là:

```
$arr1 = [1, 2, 3, 4, 5];
$arr2 = [1, 3, 4, 5, 6];
```

Thì việc so sánh của theo vị trí tương ứng với nhau sẽ theo kiểu so sánh `$arr1[0]` với `$arr2[0]`, `$arr1[1]` với `$arr2[1]`, ... cho đến hết. Cách mà `React` thực hiện so sánh lần lượt cũng tương tự như vậy, so sánh 2 `Element` có cùng vị trí ở `Virtual DOM Tree` cũ và mới với nhau. Trong trường hợp nếu bên `Virtual DOM Tree` cũ hoặc mới ít `Element` hơn thì các vị trí đó sẽ nhận được giá trị là null. Quay trở lại với ví dụ nói trên thì như bạn thấy ở đây:

```
// Old
{
  type: 'button',
  props: { className: 'blue' }
}
// New
{
  type: 'button',
  props: { className: 'red' }
}
```

Cả 2 đều có cùng kiểu type là `button` chính vì thế `React` sẽ đưa được ra ngay quyết định đó là tái sử dụng `Element` này thay vì phải tạo mới nó. Ví thể quy tắc ở đây rất đơn giản:

- `type` khác nhau -> tạo mới
- `type` giống nhau -> tái sử dụng

Xét ví dụ tiếp theo:

```
// Old
{
  type: 'div',
  props: { className: 'blue' }
}
// New
{
  type: 'span',
  props: { className: 'blue' }
}
```

Nếu áp dụng quy tắc mà mình nói trên thì như bạn có thể thấy ngay khi bắt đầu việc so sánh thì ta có thể thấy ngay type của `Element` đã bị thay đổi từ `div` sang `span`. Chính vì vậy mà ở đây `React` sẽ tiến hành tạo mới `Element` này và đi kèm với việc đó sẽ gọi đến hàm `componentWillUnmount()` (nếu là class component) sau đó sẽ hủy bỏ toàn bộ trạng thái của `Element` này đi kèm với nó là cả phần `children` của nó nữa. Có nghĩa là nếu ta có phần `children` hay phần nội dung nằm bên trong nó như sau:

```
// Old
<div><SomeComponent /></div>

// New
<span><SomeComponent /></span>
```

Thì toàn bộ các component hay các thẻ tag html khác nằm bên trong nó sẽ bị xóa đi và tại lại mới hoàn toàn. Tương tự thì việc mất trạng thái cũ hay gọi hàm `componentWillUnmount()` cũng sẽ xảy ra với các component con này. Đối với các `Element` phức tạp hơn thì toàn bộ việc so sánh type để quyết định cập nhật hay tạo mới nó cũng diễn ra tương tự:

```

<div class='container'>
  <h1 class='title'>This is title</h1>
  <p class='sub-title'>This is subtitle</p>
</div>

{
  type: 'div',
  props: {
    className: 'container',
    children: [
      {
        type: 'h1',
        props: {
          className: 'title',
          children: 'This is title'
        }
      },
      {
        type: 'p',
        props: {
          className: 'sub-title',
          children: 'This is subtitle'
        }
      }
    ]
  }
}

```

React vẫn sẽ đầu tiên tiến hành so sánh `type` bọc ngoài cùng ở đây là `type: 'div'` nếu không bị thay đổi thì sẽ tiến hành đi tiếp vào bên trong nó là so sánh các `Element` con của ở vị trí tương ứng với nhau. Các bạn có thể hiểu với các `Element` có `children` là các `Element` khác thì việc so sánh nó vẫn diễn ra tuần tự với nhau theo kiểu `parent` với `parent` và `children` với `children`. Cụ thể thì nó giống như chúng ta so sánh mảng đa chiều như này:

```

$arr1 = [1, [2, 3, 6], 4];
$arr2 = [2, [3, 4, 7], 5];

```

Thì việc so sánh sẽ là `$arr1[0]` với `$arr2[0]`, `$arr1[1][0]` với `$arr2[1][0]` và lần lượt hết phần mảng con rồi mới quay ra so sánh tiếp `$arr1[2]` với `$arr2[2]`. Việc so sánh các `Element` có `children` là các `Element` khác (hay gọi là nested Element) cũng diễn ra như vậy. Còn trong trường hợp `type` bị thay đổi thì React sẽ bỏ qua toàn bộ quá trình so sánh phía trong mà lập tức tạo mới ngay. Như vậy bạn mới có thể thấy rằng trên thực tế việc ta gặp các `Element` dạng nested hay phức tạp hơn như này là rất thường xuyên. Chính vì thế như mình nói ở trên nếu không đưa ra quyết định là tạo mới hay cập nhật mà chỉ tạo mới sẽ dẫn đến ảnh hưởng về hiệu năng rất nhiều. Với toàn bộ những gì mình vừa nói thì ta sẽ xét ví dụ này:

```

// Old
<div>
  <h1 className="a">Title</h1>
  <p className="b">Hello</p>
</div>

// New
<div>
  <h2 className="a">Title</h2>
  <p className="b">Hello</p>
</div>

```

Việc so sánh sẽ lần lượt là:

- `type: 'div' -> 'div' -> cập nhật.`
- `type: 'h1' -> 'h2' -> tạo mới.`
- `type: 'p' -> 'p' -> không tạo mới.`

d. Diff Algorithm (Reconciliation) (continue)

Nhắc lại 1 chút kiến thức ở phần trước về việc so sánh 2 `Virtual DOM Tree` là React sẽ cần đưa ra quyết định là tái sử dụng một `Element` dựa vào `type` của nó như sau:

- `type` bị thay đổi -> tạo mới `Element`
 - Việc tạo mới `Element` sẽ dẫn đến toàn bộ trạng thái của `Element` đó cũng như trạng thái của các thành phần con trong `Element` đó bị loại bỏ và tạo lại mới hoàn toàn.
 - React sẽ bỏ qua việc so sánh các thành phần con này nếu `Element` cha bị tạo mới.
- `type` giữ nguyên -> tái sử dụng `Element`

- Lúc này `Element` sẽ không bị xóa bỏ mà thay vào đó sẽ được tiếp tục sử dụng.
- `React` sẽ tiếp tục so sánh các `Element` con ở bên trong theo quy tắc tương tự cho đến khi đi hết toàn bộ phần `children`.

Ngoài ra cách so sánh của `React` là sẽ là đem 2 phần tử có cùng vị trí (giống như index trong mảng) trong `Virtual DOM Tree` đem ra so sánh với nhau cho đến khi đi hết toàn bộ. Nếu `Element` đó có phần con thì sẽ đi vào trong phần con của `Element` này và so sánh giống như cách các phần tử trong mảng hai chiều. Trường hợp nếu `Virtual DOM Tree` cũ hoặc mới có ít `Element` hơn thì các chỗ thiếu đó sẽ được tính là `null`. Đó là toàn bộ những gì mà chúng ta đã nhắc đến trong bài viết trước đó. Để bắt đầu với nội dung của bài viết tiếp theo thì chúng ta sẽ đi đến với một ví dụ như sau:

```
// Old
<div><input /></div>

// New
<div><p>This is p tag</p><input /></div>
```

Kịch bản đặt ra là mỗi khi chúng ta nhập nội dung vào thẻ `<input />` thì thẻ `<p>` ở phía trên đó sẽ xuất hiện. Bây giờ chúng ta sẽ đem những kiến thức mà ta đã học được ở phần trước đó và áp dụng vào ví dụ này để xem cách `React` so sánh hai `Virtual DOM Tree` cũ và mới này như sau:

- Ở `Element` cha có `type` từ `div` -> `div` -> không tạo mới.
- Đi vào trong các `Element` con ta có:
 - Ở vị trí đầu tiên ta có `type` từ `input` -> `p` -> tạo mới thẻ `<p>`.
 - Ở vị trí thứ hai thì trong `Virtual DOM Tree` cũ ít hơn `Virtual DOM Tree` mới một phần tử nên như mình đã nói ở trên phần thiếu này sẽ được coi là `null` so sánh với `<input />` -> tạo mới thẻ `<input />`.

Như vậy là với những gì chúng ta tìm hiểu ở bài viết trước đó thì như vậy là toàn bộ quá trình **React**

so sánh hai **Virtual DOM Tree** cũ và mới. Tuy nhiên bạn có nhận ra vấn đề ở đây không?. Nếu bạn còn nhớ thì như mình đã nói, khi một **Element** được xóa bỏ và tạo mới thì nó sẽ xây ra việc **Element** đó sẽ bị mất đi toàn bộ trạng thái, ở đây sẽ bao gồm các trạng thái như việc khi ta **focus** vào ô `<input />` hay nội dung mà ta vừa nhập trong ô `<input />` sẽ bị biến mất và thay vào đó là 1 thẻ `<p>` được tạo ra cùng với một ô `<input /><input />` mới toanh. Tuy nhiên trên thực tế thì cách mà chúng ta viết code trong **React** với trường hợp nói trên sẽ có dạng như sau:

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

const Form = () => {
  const [value, setValue] = useState('');

  const handleChange = e => { setValue(e.target.value) }

  return (
    <div>
      {value !== '' && <p>This is p tag</p>}
      <input
        value={value}
        onChange={handleChange}
      />
    </div>
  )
}
```

Nhìn vào đoạn code trên và áp dụng vào ví dụ ngay trước đó bạn có thể thấy rằng khi ta chưa nhập nội dung và ngay sau khi ta bắt đầu nhập nội dung vào ô `<input />` thì `Virtual DOM Tree` cũ và mới sẽ được biểu diễn như sau:

```
// Trước khi nhập nội dung
<div>
  null
</div>

// Sau khi nhập nội dung
<div><p>This is p tag</p><input /></div>
```

Với trường hợp như trên thì việc so sánh `Virtual DOM Tree` cũ và mới sẽ được diễn ra tương tự như sau:

- Ở `Element` cha có `type` từ `div` -> `div` -> không tạo mới.

- Đi vào trong các `Element` con ta có:
 - Ở vị trí đầu tiên ta có `type` từ `null` -> `p` -> tạo mới thẻ `<p>`.
 - Ở vị trí thứ hai ta có `tyoe` từ `input` -> `input` -> không tạo mới.

Vậy là trên thực tế khi chúng ta code thì thẻ `<input />` của chúng ta ở đây không hề bị tạo mới vì thẻ `<p>` bị thiếu trước đó thực chất thì chúng ta thường đặt vào đó một giá trị `null` rồi nên nó sẽ hoàn toàn không ảnh hưởng gì đến việc so sánh bị sai sót trong việc quyết định tạo mới hay tái sử dụng. Đồng thời ở đây toàn bộ trạng thái trên thẻ `<input />` của chúng ta cũng sẽ được bảo toàn. Có một điều tiếp theo mình muốn chia sẻ cho các bạn là có thể đây là một trong những lý do mà `React` luôn yêu cầu chúng ta khi code phải có một `Element` cha bọc ngoài nếu có nhiều `Element` con cùng cấp như sau:

```
// Cách viết sai
const Demo = () => (
  <p>This is first p tag</p>
  <p>This is second p tag</p>
);

// Cách viết đúng
const Demo = () => (
  <div>
    <p>This is first p tag</p>
    <p>This is second p tag</p>
  </div>
);
```

Với trường hợp bạn code sai như trên thì sẽ nhận được một lỗi như sau:

```
SyntaxError
/src/index.js: Adjacent JSX elements must be wrapped in an enclosing tag.
Did you want a JSX fragment <...</>? (8:2)
```

Giả sử `React` cho chúng ta viết như cách sai đầu tiên thì áp vào ví dụ trước đó của chúng ta với thẻ `<input />` thì chúng ta sẽ có được đoạn code như sau:

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

const Form = () => {
  const [value, setValue] = useState('');

  const handleChange = e => { setValue(e.target.value) }

  return (
    <div>
      {
        value !== '' && (
          <p>This is first p tag</p>
          <p>This is second p tag</p>
        )
      }
      <input
        value={value}
        onChange={handleChange}
      />
    </div>
  )
}
```

Thì với đoạn code nói trên thì tương tự `Virtual DOM Tree` cũ và mới sẽ như sau:

```
<div>
  null
</div>

// Sau khi nhập nội dung
<div><p>This is first p tag</p><p>This is second p tag</p><input /></div>
```

Tuy nhiên bạn có thể dễ dàng nhận thấy quá trình so sánh 2 cây sẽ lại dẫn đến tạo mới thẻ `<input />` vì ở `Virtual DOM Tree` thứ nhất có 2 `Element` con và `Virtual DOM Tree` mới có 3 vì thế nó sẽ là:

- Ở vị trí đầu tiên thì ta có `type` là `null` -> `p` (`<p>This is first p tag</p>`) -> tạo mới

- Ở vị trí thứ 2 thì sẽ là `input` -> 'p' (`<p>This is second p tag</p>`) -> tạo mới
- Ở vị trí cuối cùng thì vì `Virtual DOM Tree` cũ có ít hơn `Virtual DOM Tree` mới một `Element` nên nó sẽ là `null` -> `input` -> tạo mới

Như vậy với trường hợp này thì `React` sẽ lại phải tạo lại thẻ `<input />` đồng nghĩa với việc ta mất hết toàn bộ trạng thái cũng như nội dung đã nhập. Mặc dù chưa tìm được tài liệu nào chính xác nói về vấn đề này nhưng theo ý kiến cá nhân của mình thì đây "**có thể**" là một trong những lý do mà `React` luôn yêu cầu chúng ta bọc một thẻ ở ngoài cùng. Tiếp theo đây chúng ta sẽ đi đến một ví dụ cuối cùng đó là việc `render` một danh sách như sau:

```
const List = ({ list }) => (
  <div>
    <h1>ITEM LIST</h1>
    {
      list.map(item => (
        <p>{item.name}</p>
      ));
    }
  </div>
)
```

Và đây là kết quả đầu ra của chúng ta:

```
const list = ['First item', 'Second item', 'Third tem']
// Kết quả thu được:
<div>
  <h1>ITEM LIST</h1>
  <p>First item</p>
  <p>Second item</p>
  <p>Third item</p>
</div>
```

Trong trường hợp danh sách của chúng ta thuộc dạng tĩnh và không bao giờ bị thay đổi về vị trí thì việc so sánh và cập nhật các `Element` sẽ diễn ra như bình thường không có vấn đề gì cả. Tuy nhiên đối với trường hợp dạng in ra danh sách như trên thì việc thứ tự bị thay đổi lại thường xuyên gặp phải. Nếu vẫn áp dụng phương pháp so sánh nói trên khi chúng ta thay đổi thứ tự trong danh sách thì như đã nói trước đó thì ở đây `type` của tất cả các `Element` vẫn là `p` nên tất nhiên ở đây sẽ không xảy ra việc tạo lại thẻ `<p>` mà thay vào đó `React` sẽ cập nhật lại toàn bộ phần nội dung nằm trong thẻ `p` vì khi được sắp xếp lại có thể dẫn đến toàn bộ nội dung đổi chỗ cho nhau như sau:

```
// Old
<div><h1>ITEM LIST</h1><p>First item</p><p>Second item</p><p>Third item</p></div>

// New
<div><h1>ITEM LIST</h1><p>Third item</p><p>First item</p><p>Second item</p></div>
```

Tuy nhiên `React` có thể làm tốt hơn thế bằng cách chúng ta thêm vào cho mỗi phần tử trong danh sách một thuộc tính là `key` như sau:

```
const List = ({ list }) => (
  <div>
    <h1>ITEM LIST</h1>
    {
      list.map(item => (
        <p key={item.id}>{item.name}</p>
      ));
    }
  </div>
);

const list = [
  {key: 1, name: 'First item'},
  {key: 2, name: 'Second item'},
  {key: 3, name: 'Third tem'}
];
// Kết quả thu được:
<div>
  <h1>ITEM LIST</h1>
  <p key="1">First item</p>
  <p key="2">Second item</p>
  <p key="3">Third item</p>
</div>
```


Trong trường hợp `Element` của chúng ta có `key` thì lúc này thay vì so sánh và update nội dung ngay thì `React` sẽ nhìn vào `Virtual DOM Tree` cũ vào tìm cái `key` đó để so tìm cái `key` với giá trị tương ứng rồi mới quyết định có cần update nội dung không hay chỉ cần sắp xếp lại thôi. Vì với 2 `Element` có giá trị `key` tương ứng sẽ được so sánh và quyết định là sắp xếp lại hay cập nhật nội dung. Việc sử dụng `key` như vậy còn hiệu quả hơn khi chúng ta thêm một phần tử mới vào đầu danh sách:

```
// Old
<div>
  <h1>ITEM LIST</h1>
  <p key="1">Fist item</p>
  <p key="2">Second item</p>
  <p key="3">Third item</p>
</div>

// New
<div>
  <h1>ITEM LIST</h1>
  <p key="4">Fourth item</p>
  <p key="1">Fist item</p>
  <p key="2">Second item</p>
  <p key="3">Third item</p>
</div>
```

Với trường hợp nói trên mà không có `key` thì sẽ dẫn đến việc cập nhật lại toàn bộ 3 phần tử đầu tiên và thêm mới phần tử cuối cùng. Còn khi bạn `Element` đã có `key` thì `React` sẽ chỉ cần sắp xếp lại 3 phần tử đầu tiên và đồng thời thêm 1 phần tử mới vào đầu danh sách.