

---

# Table of Contents

Installation - Get Started	1.1
Add React To A Web Site	1.1.1
Create A New React App	1.1.2
CDN Links	1.1.3
Tutorial	1.2
Main Concepts - Hello World	1.3
Introducing JSX	1.3.1
Rendering Elements	1.3.2
Components And Props	1.3.3
State And Lifecycle	1.3.4
Handling Events	1.3.5
Conditional Rendering	1.3.6
List And Keys	1.3.7
Forms	1.3.8
Lifting State Up	1.3.9
Composition vs Inheritance	1.3.10
Thinking in React	1.3.11
Advanced Guides - Accessibility	1.4
Code Splitting	1.4.1
Context	1.4.2
Error Boundaries	1.4.3
Forwarding Refs	1.4.4
Fragments	1.4.5
Higher-Order Components	1.4.6
Integrating With Other Libraries	1.4.7
JSX in Depth	1.4.8
Optimizing Performance	1.4.9
Portals	1.4.10
React Without ES6	1.4.11
React Without JSX	1.4.12
Reconciliation	1.4.13
Refs and The DOM	1.4.14
Render Props	1.4.15
Static Type Checking	1.4.16
Strict Mode	1.4.17
TypeChecking With PropTypes	1.4.18
Uncontrolled Components	1.4.19

---

Web Components	1.4.20
API Reference -React	1.5
React.Component	1.5.1
ReactDOM	1.5.2
ReactDOMServer	1.5.3
DOM Elements	1.5.4
SyntheticEvent	1.5.5
Test Utilities	1.5.6
Shallow Renderer	1.5.7
Test Renderer	1.5.8
JS Environment Requirements	1.5.9
Glossary	1.5.10
FAQ - AJAX And APIS	1.6
Babel, JSX, and Build Steps	1.6.1
Passing Functions to Components	1.6.2
Component State	1.6.3
Styling and CSS	1.6.4
File Structure	1.6.5
Virtual DOM and Internals	1.6.6
Warnings - Don't Call PropTypes Warning	1.7
Invalid Aria Prop	1.7.1
Legacy Factories	1.7.2
Refs Must Have Owner	1.7.3
Special Props	1.7.4
Unknown Props	1.7.5
Blogs - React v16.4.2: Server-side vulnerability fix	1.8
You Probably Don't Need Derived State	1.8.1
React v16.4.0: Pointer Events	1.8.2
React v16.3.0: New lifecycles and context API	1.8.3
Update on Async Rendering	1.8.4
Sneak Peek: Beyond React 16	1.8.5
Behind the Scenes: Improving the Repository Infrastructure	1.8.6

This page is an overview of the React documentation and related resources.

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

---

- [Try React](#)
- [Learn React](#)
- [Staying Informed](#)
- [Versioned Documentation](#)
- [Something Missing?](#)

## Try React

React has been designed from the start for gradual adoption, and **you can use as little or as much React as you need**. Whether you want to get a taste of React, add some interactivity to a simple HTML page, or start a complex React-powered app, the links in this section will help you get started.

### Online Playgrounds

If you're interested in playing around with React, you can use an online code playground. Try a Hello World template on [CodePen](#) or [CodeSandbox](#).

If you prefer to use your own text editor, you can also [download this HTML file](#), edit it, and open it from the local filesystem in your browser. It does a slow runtime code transformation, so we'd only recommend using this for simple demos.

### Add React to a Website

You can [add React to an HTML page in one minute](#). You can then either gradually expand its presence, or keep it contained to a few dynamic widgets.

### Create a New React App

When starting a React project, [a simple HTML page with script tags](#) might still be the best option. It only takes a minute to set up!

As your application grows, you might want to consider a more integrated setup. There are [several JavaScript toolchains](#) we recommend for larger applications. Each of them can work with little to no configuration and lets you take full advantage of the rich React ecosystem.

## Learn React

People come to React from different backgrounds and with different learning styles. Whether you prefer a more theoretical or a practical approach, we hope you'll find this section helpful.

- If you prefer to **learn by doing**, start with our [practical tutorial](#).
- If you prefer to **learn concepts step by step**, start with our [guide to main concepts](#).

Like any unfamiliar technology, React does have a learning curve. With practice and some patience, you *will* get the hang of it.

### First Examples

The [React homepage](#) contains a few small React examples with a live editor. Even if you don't know anything about React yet, try changing their code and see how it affects the result.

---

## React for Designers

If you're coming from a design background, [these resources](#) are a great place to get started.

## JavaScript Resources

The React documentation assumes some familiarity with programming in the JavaScript language. You don't have to be an expert, but it's harder to learn both React and JavaScript at the same time.

We recommend going through [this JavaScript overview](#) to check your knowledge level. It will take you between 30 minutes and an hour but you will feel more confident learning React.

### Tip

Whenever you get confused by something in JavaScript, [MDN](#) and [javascript.info](#) are great websites to check. There are also [community support forums](#) where you can ask for help.

## Practical Tutorial

If you prefer to **learn by doing**, check out our [practical tutorial](#). In this tutorial, we build a tic-tac-toe game in React. You might be tempted to skip it because you're not building games -- but give it a chance. The techniques you'll learn in the tutorial are fundamental to building *any* React apps, and mastering it will give you a much deeper understanding.

## Step-by-Step Guide

If you prefer to **learn concepts step by step**, our [guide to main concepts](#) is the best place to start. Every next chapter in it builds on the knowledge introduced in the previous chapters so you won't miss anything as you go along.

## Thinking in React

Many React users credit reading [Thinking in React](#) as the moment React finally "clicked" for them. It's probably the oldest React walkthrough but it's still just as relevant.

## Recommended Courses

Sometimes people find third-party books and video courses more helpful than the official documentation. We maintain [a list of commonly recommended resources](#), some of which are free.

## Advanced Concepts

Once you're comfortable with the [main concepts](#) and played with React a little bit, you might be interested in more advanced topics. This section will introduce you to the powerful, but less commonly used React features like [context](#) and [refs](#).

## API Reference

This documentation section is useful when you want to learn more details about a particular React API. For example, `React.Component` [API reference](#) can provide you with details on how `setState()` works, and what different lifecycle hooks are useful for.

## Glossary and FAQ

The [glossary](#) contains an overview of the most common terms you'll see in the React documentation. There is also a FAQ section dedicated to short questions and answers about common topics, including [making AJAX requests](#), [component state](#), and [file structure](#).

## Staying Informed

The [React blog](#) is the official source for the updates from the React team. Anything important, including release notes or deprecation notices, will be posted there first.

You can also follow the [@reactjs account](#) on Twitter, but you won't miss anything essential if you only read the blog.

Not every React release deserves its own blog post, but you can find a detailed changelog for every release [in the `CHANGELOG.md` file in the React repository](#), as well as on the [Releases](#) page.

## Versioned Documentation

This documentation always reflects the latest stable version of React. Since React 16, you can find older versions of the documentation [on a separate page](#). Note that documentation for past versions is snapshotted at the time of the release, and isn't being continuously updated.

## Something Missing?

If something is missing in the documentation or if you found some part confusing, please [file an issue for the documentation repository](#) with your suggestions for improvement, or tweet at the [@reactjs account](#). We love hearing from you!

Use as little or as much React as you need.

React has been designed from the start for gradual adoption, and **you can use as little or as much React as you need**. Perhaps you only want to add some "sprinkles of interactivity" to an existing page. React components are a great way to do that.

The majority of websites aren't, and don't need to be, single-page apps. With **a few lines of code and no build tooling**, try React in a small part of your website. You can then either gradually expand its presence, or keep it contained to a few dynamic widgets.

- 
- [Add React in One Minute](#)
  - [Optional: Try React with JSX](#) (no bundler necessary!)

## Add React in One Minute

In this section, we will show how to add a React component to an existing HTML page. You can follow along with your own website, or create an empty HTML file to practice.

There will be no complicated tools or install requirements -- **to complete this section, you only need an internet connection, and a minute of your time.**

Optional: [Download the full example \(2KB zipped\)](#)

### Step 1: Add a DOM Container to the HTML

First, open the HTML page you want to edit. Add an empty `<div>` tag to mark the spot where you want to display something with React. For example:

```
<!-- ... existing HTML ... -->

<div id="like_button_container"></div>

<!-- ... existing HTML ... -->
```

We gave this `<div>` a unique `id` HTML attribute. This will allow us to find it from the JavaScript code later and display a React component inside of it.

#### Tip

You can place a "container" `<div>` like this **anywhere** inside the `<body>` tag. You may have as many independent DOM containers on one page as you need. They are usually empty -- React will replace any existing content inside DOM containers.

### Step 2: Add the Script Tags

Next, add three `<script>` tags to the HTML page right before the closing `</body>` tag:

```
<!-- ... other HTML ... -->

<!-- Load React. -->
<!-- Note: when deploying, replace "development.js" with "production.min.js". -->
<script src="https://unpkg.com/react@16/umd/react.development.js" crossorigin>
</script>
<script src="https://unpkg.com/react-dom@16/umd/react-dom.development.js"
```

```
crossorigin></script>

<!-- Load our React component. -->
<script src="like_button.js"></script>

</body>
```

The first two tags load React. The third one will load your component code.

### Step 3: Create a React Component

Create a file called `like_button.js` next to your HTML page.

Open [this starter code](#) and paste it into the file you created.

#### Tip

This code defines a React component called `LikeButton`. Don't worry if you don't understand it yet -- we'll cover the building blocks of React later in our [hands-on tutorial](#) and [main concepts guide](#). For now, let's just get it showing on the screen!

After [the starter code](#), add two lines to the bottom of `like_button.js` :

```
// ... the starter code you pasted ...

const domContainer = document.querySelector('#like_button_container');
ReactDOM.render(e(LikeButton), domContainer);
```

These two lines of code find the `<div>` we added to our HTML in the first step, and then display our "Like" button React component inside of it.

### That's It!

There is no step four. **You have just added the first React component to your website.**

Check out the next sections for more tips on integrating React.

[View the full example source code](#)

[Download the full example \(2KB zipped\)](#)

### Tip: Reuse a Component

Commonly, you might want to display React components in multiple places on the HTML page. Here is an example that displays the "Like" button three times and passes some data to it:

[View the full example source code](#)

[Download the full example \(2KB zipped\)](#)

#### Note

This strategy is mostly useful while React-powered parts of the page are isolated from each other. Inside React code, it's easier to use [component composition](#) instead.

### Tip: Minify JavaScript for Production

Before deploying your website to production, be mindful that unminified JavaScript can significantly slow down the page for your users.

If you already minify the application scripts, **your site will be production-ready** if you ensure that the deployed HTML loads the versions of React ending in `production.min.js` :

```
<script src="https://unpkg.com/react@16/umd/react.production.min.js" crossorigin>
</script>
<script src="https://unpkg.com/react-dom@16/umd/react-dom.production.min.js"
crossorigin></script>
```

If you don't have a minification step for your scripts, [here's one way to set it up](#).

## Optional: Try React with JSX

In the examples above, we only relied on features that are natively supported by the browsers. This is why we used a JavaScript function call to tell React what to display:

```
const e = React.createElement;

// Display a "Like" <button>
return e(
  'button',
  { onClick: () => this.setState({ liked: true }) },
  'Like'
);
```

However, React also offers an option to use [JSX](#) instead:

```
// Display a "Like" <button>
return (
  <button onClick={() => this.setState({ liked: true })}>
    Like
  </button>
);
```

These two code snippets are equivalent. While **JSX is completely optional**, many people find it helpful for writing UI code -- both with React and with other libraries.

You can play with JSX using [this online converter](#).

## Quickly Try JSX

The quickest way to try JSX in your project is to add this `<script>` tag to your page:

```
<script src="https://unpkg.com/babel-standalone@6/babel.min.js"></script>
```

Now you can use JSX in any `<script>` tag by adding `type="text/babel"` attribute to it. Here is [an example HTML file with JSX](#) that you can download and play with.



This approach is fine for learning and creating simple demos. However, it makes your website slow and **isn't suitable for production**. When you're ready to move forward, remove this new `<script>` tag and the `type="text/babel"` attributes you've added. Instead, in the next section you will set up a JSX preprocessor to convert all your `<script>` tags automatically.

## Add JSX to a Project

Adding JSX to a project doesn't require complicated tools like a bundler or a development server. Essentially, adding JSX **is a lot like adding a CSS preprocessor**. The only requirement is to have [Node.js](#) installed on your computer.

Go to your project folder in the terminal, and paste these two commands:

1. **Step 1:** Run `npm init -y` (if it fails, [here's a fix](#))
2. **Step 2:** Run `npm install babel-cli@6 babel-preset-react-app@3`

### Tip

We're **using npm here only to install the JSX preprocessor**; you won't need it for anything else. Both React and the application code can stay as `<script>` tags with no changes.

Congratulations! You just added a **production-ready JSX setup** to your project.

## Run JSX Preprocessor

Create a folder called `src` and run this terminal command:

```
npx babel --watch src --out-dir . --presets react-app/prod
```

### Note

`npx` is not a typo -- it's a [package runner tool that comes with npm 5.2+](#).

If you see an error message saying "You have mistakenly installed the `babel` package", you might have missed [the previous step](#). Perform it in the same folder, and then try again.

Don't wait for it to finish -- this command starts an automated watcher for JSX.

If you now create a file called `src/like_button.js` with this [JSX starter code](#), the watcher will create a preprocessed `like_button.js` with the plain JavaScript code suitable for the browser. When you edit the source file with JSX, the transform will re-run automatically.

As a bonus, this also lets you use modern JavaScript syntax features like classes without worrying about breaking older browsers. The tool we just used is called Babel, and you can learn more about it from [its documentation](#).

If you notice that you're getting comfortable with build tools and want them to do more for you, [the next section](#) describes some of the most popular and approachable toolchains. If not -- those script tags will do just fine!

Use an integrated toolchain for the best user and developer experience.

This page describes a few popular React toolchains which help with tasks like:

- Scaling to many files and components.
- Using third-party libraries from npm.
- Detecting common mistakes early.
- Live-editing CSS and JS in development.
- Optimizing the output for production.

The toolchains recommended on this page **don't require configuration to get started**.

## You Might Not Need a Toolchain

If you don't experience the problems described above or don't feel comfortable using JavaScript tools yet, consider [adding React as a plain `<script>` tag on an HTML page](#), optionally [with JSX](#).

This is also **the easiest way to integrate React into an existing website**. You can always add a larger toolchain if you find it helpful!

## Recommended Toolchains

The React team primarily recommends these solutions:

- If you're **learning React** or **creating a new single-page app**, use [Create React App](#).
- If you're building a **server-rendered website with Node.js**, try [Next.js](#).
- If you're building a **static content-oriented website**, try [Gatsby](#).
- If you're building a **component library** or **integrating with an existing codebase**, try [More Flexible Toolchains](#).

## Create React App

[Create React App](#) is a comfortable environment for **learning React**, and is the best way to start building a **new single-page application** in React.

It sets up your development environment so that you can use the latest JavaScript features, provides a nice developer experience, and optimizes your app for production. You'll need to have Node  $\geq 6$  and npm  $\geq 5.2$  on your machine. To create a project, run:

```
npx create-react-app my-app
cd my-app
npm start
```

### Note

`npx` on the first line is not a typo -- it's a [package runner tool that comes with npm 5.2+](#).

Create React App doesn't handle backend logic or databases; it just creates a frontend build pipeline, so you can use it with any backend you want. Under the hood, it uses [Babel](#) and [webpack](#), but you don't need to know anything about them.

When you're ready to deploy to production, running `npm run build` will create an optimized build of your app in the `build` folder. You can learn more about Create React App [from its README](#) and the [User Guide](#).

## Next.js

[Next.js](#) is a popular and lightweight framework for **static and server-rendered applications** built with React. It includes **styling and routing solutions** out of the box, and assumes that you're using [Node.js](#) as the server environment.

Learn Next.js from [its official guide](#).

## Gatsby

[Gatsby](#) is the best way to create **static websites** with React. It lets you use React components, but outputs pre-rendered HTML and CSS to guarantee the fastest load time.

Learn Gatsby from [its official guide](#) and a [gallery of starter kits](#).

## More Flexible Toolchains

The following toolchains offer more flexibility and choice. We recommend them to more experienced users:

- [Neutrino](#) combines the power of [webpack](#) with the simplicity of presets, and includes a preset for [React apps](#) and [React components](#).
- [nwb](#) is particularly great for [publishing React components for npm](#). It [can be used](#) for creating React apps, too.
- [Parcel](#) is a fast, zero configuration web application bundler that [works with React](#).
- [Razzle](#) is a server-rendering framework that doesn't require any configuration, but offers more flexibility than Next.js.

## Creating a Toolchain from Scratch

A JavaScript build toolchain typically consists of:

- A **package manager**, such as [Yarn](#) or [npm](#). It lets you take advantage of a vast ecosystem of third-party packages, and easily install or update them.
- A **bundler**, such as [webpack](#) or [Parcel](#). It lets you write modular code and bundle it together into small packages to optimize load time.
- A **compiler** such as [Babel](#). It lets you write modern JavaScript code that still works in older browsers.

If you prefer to set up your own JavaScript toolchain from scratch, [check out this guide](#) that re-creates some of the Create React App functionality.

Don't forget to ensure your custom toolchain [is correctly set up for production](#).

Both React and ReactDOM are available over a CDN.

```
<script crossorigin src="https://unpkg.com/react@16/umd/react.development.js">
</script>
<script crossorigin src="https://unpkg.com/react-dom@16/umd/react-dom.development.js"></script>
```

The versions above are only meant for development, and are not suitable for production. Minified and optimized production versions of React are available at:

```
<script crossorigin src="https://unpkg.com/react@16/umd/react.production.min.js">
</script>
<script crossorigin src="https://unpkg.com/react-dom@16/umd/react-dom.production.min.js"></script>
```

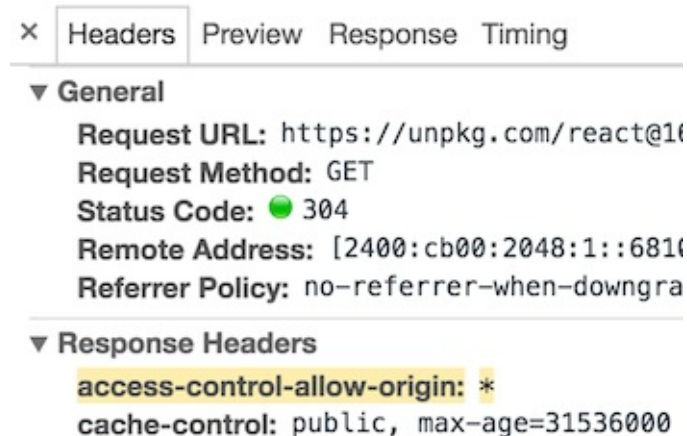
To load a specific version of `react` and `react-dom`, replace `16` with the version number.

## Why the `crossorigin` Attribute?

If you serve React from a CDN, we recommend to keep the `crossorigin` attribute set:

```
<script crossorigin src="..."></script>
```

We also recommend to verify that the CDN you are using sets the `Access-Control-Allow-Origin: *` HTTP header:



This enables a better [error handling experience](#) in React 16 and later.

This tutorial doesn't assume any existing React knowledge.

## Before We Start the Tutorial

We will build a small game during this tutorial. **You might be tempted to skip it because you're not building games -- but give it a chance.** The techniques you'll learn in the tutorial are fundamental to building any React apps, and mastering it will give you a deep understanding of React.

### Tip

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

The tutorial is divided into several sections:

- [Setup for the Tutorial](#) will give you **a starting point** to follow the tutorial.
- [Overview](#) will teach you **the fundamentals** of React: components, props, and state.
- [Completing the Game](#) will teach you **the most common techniques** in React development.
- [Adding Time Travel](#) will give you **a deeper insight** into the unique strengths of React.

You don't have to complete all of the sections at once to get the value out of this tutorial. Try to get as far as you can -- even if it's one or two sections.

It's fine to copy and paste code as you're following along the tutorial, but we recommend to type it by hand. This will help you develop a muscle memory and a stronger understanding.

## What Are We Building?

In this tutorial, we'll show how to build an interactive tic-tac-toe game with React.

You can see what we'll be building here: [Final Result](#). If the code doesn't make sense to you, or if you are unfamiliar with the code's syntax, don't worry! The goal of this tutorial is to help you understand React and its syntax.

We recommend that you check out the tic-tac-toe game before continuing with the tutorial. One of the features that you'll notice is that there is a numbered list to the right of the game's board. This list gives you a history of all of the moves that have occurred in the game, and is updated as the game progresses.

You can close the tic-tac-toe game once you're familiar with it. We'll be starting from a simpler template in this tutorial. Our next step is to set you up so that you can start building the game.

## Prerequisites

We'll assume that you have some familiarity with HTML and JavaScript, but you should be able to follow along even if you're coming from a different programming language. We'll also assume that you're familiar with programming concepts like functions, objects, arrays, and to a lesser extent, classes.

If you need to review JavaScript, we recommend reading [this guide](#). Note that we're also using some features from ES6 -- a recent version of JavaScript. In this tutorial, we're using [arrow functions](#), [classes](#), `let`, and `const` statements. You can use the [Babel REPL](#) to check what ES6 code compiles to.

## Setup for the Tutorial

There are two ways to complete this tutorial: you can either write the code in your browser, or you can set up a local development environment on your computer.

## Setup Option 1: Write Code in the Browser

This is the quickest way to get started!

First, open this [Starter Code](#) in a new tab. The new tab should display an empty tic-tac-toe game board and React code. We will be editing the React code in this tutorial.

You can now skip the second setup option, and go to the [Overview](#) section to get an overview of React.

## Setup Option 2: Local Development Environment

This is completely optional and not required for this tutorial!

► **Optional: Instructions for following along locally using your preferred text editor**

## Help, I'm Stuck!

If you get stuck, check out the [community support resources](#). In particular, [Reactiflux Chat](#) is a great way to get help quickly. If you don't receive an answer, or if you remain stuck, please file an issue, and we'll help you out.

## Overview

Now that you're set up, let's get an overview of React!

## What Is React?

React is a declarative, efficient, and flexible JavaScript library for building user interfaces. It lets you compose complex UIs from small and isolated pieces of code called "components".

React has a few different kinds of components, but we'll start with `React.Component` subclasses:

```
class ShoppingList extends React.Component {
  render() {
    return (
      <div className="shopping-list">
        <h1>Shopping List for {this.props.name}</h1>
        <ul>
          <li>Instagram</li>
          <li>WhatsApp</li>
          <li>Oculus</li>
        </ul>
      </div>
    );
  }
}

// Example usage: <ShoppingList name="Mark" />
```

We'll get to the funny XML-like tags soon. We use components to tell React what we want to see on the screen. When our data changes, React will efficiently update and re-render our components.

Here, `ShoppingList` is a **React component class**, or **React component type**. A component takes in parameters, called `props` (short for "properties"), and returns a hierarchy of views to display via the `render` method.

The `render` method returns a *description* of what you want to see on the screen. React takes the description and displays the result. In particular, `render` returns a **React element**, which is a lightweight description of what to render. Most React developers use a special syntax called "JSX" which makes these structures easier to write. The `<div />` syntax is transformed at build time to `React.createElement('div')`. The example above is equivalent to:

```
return React.createElement('div', {className: 'shopping-list'},
  React.createElement('h1', /* ... h1 children ... */),
  React.createElement('ul', /* ... ul children ... */)
);
```

[See full expanded version.](#)

If you're curious, `createElement()` is described in more detail in the [API reference](#), but we won't be using it in this tutorial. Instead, we will keep using JSX.

JSX comes with the full power of JavaScript. You can put *any* JavaScript expressions within braces inside JSX. Each React element is a JavaScript object that you can store in a variable or pass around in your program.

The `ShoppingList` component above only renders built-in DOM components like `<div />` and `<li />`. But you can compose and render custom React components too. For example, we can now refer to the whole shopping list by writing `<ShoppingList />`. Each React component is encapsulated and can operate independently; this allows you to build complex UIs from simple components.

## Inspecting the Starter Code

If you're going to work on the tutorial **in your browser**, open this code in a new tab: [Starter Code](#). If you're going to work on the tutorial **locally**, instead open `src/index.js` in your project folder (you have already touched this file during the [setup](#)).

This Starter Code is the base of what we're building. We've provided the CSS styling so that you only need focus on learning React and programming the tic-tac-toe game.

By inspecting the code, you'll notice that we have three React components:

- Square
- Board
- Game

The Square component renders a single `<button>` and the Board renders 9 squares. The Game component renders a board with placeholder values which we'll modify later. There are currently no interactive components.

## Passing Data Through Props

Just to get our feet wet, let's try passing some data from our Board component to our Square component.

In Board's `renderSquare` method, change the code to pass a prop called `value` to the Square:

```
class Board extends React.Component {
  renderSquare(i) {
    return <Square value={i} />;
  }
}
```

Change Square's `render` method to show that value by replacing `{/* TODO */}` with `{this.props.value}` :

```
class Square extends React.Component {
  render() {
    return (
      <button className="square">
        {this.props.value}
      </button>
    );
  }
}
```

Before:

Next player: X


After: You should see a number in each square in the rendered output.

Next player: X

<b>0</b>	<b>1</b>	<b>2</b>
<b>3</b>	<b>4</b>	<b>5</b>
<b>6</b>	<b>7</b>	<b>8</b>

[View the full code at this point](#)

Congratulations! You've just "passed a prop" from a parent Board component to a child Square component. Passing props is how information flows in React apps, from parents to children.



## Making an Interactive Component

Let's fill the Square component with an "X" when we click it. First, change the button tag that is returned from the Square component's `render()` function to this:

```
class Square extends React.Component {
  render() {
    return (
      <button className="square" onClick={function() { alert('click'); }}>
        {this.props.value}
      </button>
    );
  }
}
```

If we click on a Square now, we should get an alert in our browser.

### Note

To save typing and avoid the [confusing behavior of `this`](#), we will use the [arrow function syntax](#) for event handlers here and further below:

```
class Square extends React.Component {
  render() {
    return (
      <button className="square" onClick={() => alert('click')}>
        {this.props.value}
      </button>
    );
  }
}
```

Notice how with `onClick={() => alert('click')}`, we're passing a *function* as the `onClick` prop. It only fires after a click. Forgetting `() =>` and writing `onClick={alert('click')}` is a common mistake, and would fire the alert every time the component re-renders.

As a next step, we want the Square component to "remember" that it got clicked, and fill it with an "X" mark. To "remember" things, components use **state**.

React components can have state by setting `this.state` in their constructors. `this.state` should be considered as private to a React component that it's defined in. Let's store the current value of the Square in `this.state`, and change it when the Square is clicked.

First, we'll add a constructor to the class to initialize the state:

```
class Square extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      value: null,
    };
  }
}
```

```
render() {
  return (
    <button className="square" onClick={() => alert('click')}>
      {this.props.value}
    </button>
  );
}
```

#### Note

In [JavaScript classes](#), you need to always call `super` when defining the constructor of a subclass. All React component classes that have a `constructor` should start it with a `super(props)` call.

Now we'll change the Square's `render` method to display the current state's value when clicked:

- Replace `this.props.value` with `this.state.value` inside the `<button>` tag.
- Replace the `() => alert()` event handler with `() => this.setState({value: 'X'})`.
- Put the `className` and `onClick` props on separate lines for better readability.

After these changes, the `<button>` tag that is returned by the Square's `render` method looks like this:

```
class Square extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      value: null,
    };
  }

  render() {
    return (
      <button
        className="square"
        onClick={() => this.setState({value: 'X'})}
      >
        {this.state.value}
      </button>
    );
  }
}
```

By calling `this.setState` from an `onClick` handler in the Square's `render` method, we tell React to re-render that Square whenever its `<button>` is clicked. After the update, the Square's `this.state.value` will be `'X'`, so we'll see the `X` on the game board. If you click on any Square, an `X` should show up.

When you call `setState` in a component, React automatically updates the child components inside of it too.

[View the full code at this point](#)

## Developer Tools

The React Devtools extension for [Chrome](#) and [Firefox](#) lets you inspect a React component tree with your browser's developer tools.

```
▼ <Game>
  ▼ <div className="game">
    ▼ <div className="game-board">
      ▼ <Board>
        ▼ <div>
          <div className="status">Next player: X</div>
          ▼ <div className="board-row">
            ▶ <Square>...</Square>
            ▶ <Square>...</Square>
            ▶ <Square>...</Square>
          </div>
          ▼ <div className="board-row">
            ▶ <Square>...</Square>
            ▶ <Square>...</Square>
            ▶ <Square>...</Square>
          </div>
          ▼ <div className="board-row">
            ▶ <Square>...</Square>
            ▶ <Square>...</Square>
            ▶ <Square>...</Square>
          </div>
        </div>
      </Board>
    </div>
    ▼ <div className="game-info">
      <div/>
      <ol/>
    </div>
  </div>
</Game>
```

The React DevTools let you check the props and the state of your React components.

After installing React DevTools, you can right-click on any element on the page, click "Inspect" to open the developer tools, and the React tab will appear as the last tab to the right.

**However, note there are a few extra steps to get it working with CodePen:**

1. Log in or register and confirm your email (required to prevent spam).
2. Click the "Fork" button.
3. Click "Change View" and then choose "Debug mode".
4. In the new tab that opens, the devtools should now have a React tab.

## Completing the Game

We now have the basic building blocks for our tic-tac-toe game. To have a complete game, we now need to alternate placing "X"s and "O"s on the board, and we need a way to determine a winner.

### Lifting State Up

Currently, each Square component maintains the game's state. To check for a winner, we'll maintain the value of each of the 9 squares in one location.

We may think that Board should just ask each Square for the Square's state. Although this approach is possible in React, we discourage it because the code becomes difficult to understand, susceptible to bugs, and hard to refactor. Instead, the best approach is to store the game's state in the parent Board component instead of in each Square. The Board component can tell each

Square what to display by passing a prop, [just like we did when we passed a number to each Square](#).

**To collect data from multiple children, or to have two child components communicate with each other, you need to declare the shared state in their parent component instead. The parent component can pass the state back down to the children by using props; this keeps the child components in sync with each other and with the parent component.**

Lifting state into a parent component is common when React components are refactored -- let's take this opportunity to try it out. We'll add a constructor to the Board and set the Board's initial state to contain an array with 9 nulls. These 9 nulls correspond to the 9 squares:

```
class Board extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      squares: Array(9).fill(null),
    };
  }

  renderSquare(i) {
    return <Square value={i} />;
  }

  render() {
    const status = 'Next player: X';

    return (
      <div>
        <div className="status">{status}</div>
        <div className="board-row">
          {this.renderSquare(0)}
          {this.renderSquare(1)}
          {this.renderSquare(2)}
        </div>
        <div className="board-row">
          {this.renderSquare(3)}
          {this.renderSquare(4)}
          {this.renderSquare(5)}
        </div>
        <div className="board-row">
          {this.renderSquare(6)}
          {this.renderSquare(7)}
          {this.renderSquare(8)}
        </div>
      </div>
    );
  }
}
```

When we fill the board in later, the board will look something like this:

```
[
```

```
'O', null, 'X',
'X', 'X', 'O',
'O', null, null,
]
```

The Board's `renderSquare` method currently looks like this:

```
renderSquare(i) {
  return <Square value={i} />;
}
```

In the beginning, we passed the `value` prop down from the Board to show numbers from 0 to 8 in every Square. In a different previous step, we replaced the numbers with an "X" mark determined by Square's own state. This is why Square currently ignores the `value` prop passed to it by the Board.

We will now use the prop passing mechanism again. We will modify the Board to instruct each individual Square about its current value ( 'X' , 'O' , or null ). We have already defined the `squares` array in the Board's constructor, and we will modify the Board's `renderSquare` method to read from it:

```
renderSquare(i) {
  return <Square value={this.state.squares[i]} />;
}
```

[View the full code at this point](#)

Each Square will now receive a `value` prop that will either be 'X' , 'O' , or null for empty squares.

Next, we need to change what happens when a Square is clicked. The Board component now maintains which squares are filled. We need to create a way for the Square to update the Board's state. Since state is considered to be private to a component that defines it, we cannot update the Board's state directly from Square.

To maintain the Board's state's privacy, we'll pass down a function from the Board to the Square. This function will get called when a Square is clicked. We'll change the `renderSquare` method in Board to:

```
renderSquare(i) {
  return (
    <Square
      value={this.state.squares[i]}
      onClick={() => this.handleClick(i)}
    />
  );
}
```

#### Note

We split the returned element into multiple lines for readability, and added parentheses so that JavaScript doesn't insert a semicolon after `return` and break our code.

Now we're passing down two props from Board to Square: `value` and `onClick` . The `onClick` prop is a function that Square can call when clicked. We'll make the following changes to Square:

- Replace `this.state.value` with `this.props.value` in Square's `render` method
- Replace `this.setState()` with `this.props.onClick()` in Square's `render` method

- Delete the `constructor` from `Square` because `Square` no longer keeps track of the game's state

After these changes, the `Square` component looks like this:

```
class Square extends React.Component {
  render() {
    return (
      <button
        className="square"
        onClick={() => this.props.onClick()}
      >
        {this.props.value}
      </button>
    );
  }
}
```

When a `Square` is clicked, the `onClick` function provided by the `Board` is called. Here's a review of how this is achieved:

1. The `onClick` prop on the built-in DOM `<button>` component tells React to set up a click event listener.
2. When the button is clicked, React will call the `onClick` event handler that is defined in `Square`'s `render()` method.
3. This event handler calls `this.props.onClick()`. The `Square`'s `onClick` prop was specified by the `Board`.
4. Since the `Board` passed `onClick={() => this.handleClick(i)}` to `Square`, the `Square` calls `this.handleClick(i)` when clicked.
5. We have not defined the `handleClick()` method yet, so our code crashes.

#### Note

The DOM `<button>` element's `onClick` attribute has a special meaning to React because it is a built-in component. For custom components like `Square`, the naming is up to you. We could name the `Square`'s `onClick` prop or `Board`'s `handleClick` method differently. In React, however, it is a convention to use `on[Event]` names for props which represent events and `handle[Event]` for the methods which handle the events.

When we try to click a `Square`, we should get an error because we haven't defined `handleClick` yet. We'll now add `handleClick` to the `Board` class:

```
class Board extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      squares: Array(9).fill(null),
    };
  }

  handleClick(i) {
    const squares = this.state.squares.slice();
    squares[i] = 'X';
    this.setState({squares: squares});
  }

  renderSquare(i) {
    return (
      <Square
```

```

        value={this.state.squares[i]}
        onClick={() => this.handleClick(i)}
      />
    );
  }

  render() {
    const status = 'Next player: X';

    return (
      <div>
        <div className="status">{status}</div>
        <div className="board-row">
          {this.renderSquare(0)}
          {this.renderSquare(1)}
          {this.renderSquare(2)}
        </div>
        <div className="board-row">
          {this.renderSquare(3)}
          {this.renderSquare(4)}
          {this.renderSquare(5)}
        </div>
        <div className="board-row">
          {this.renderSquare(6)}
          {this.renderSquare(7)}
          {this.renderSquare(8)}
        </div>
      </div>
    );
  }
}

```

[View the full code at this point](#)

After these changes, we're again able to click on the Squares to fill them. However, now the state is stored in the Board component instead of the individual Square components. When the Board's state changes, the Square components re-render automatically. Keeping the state of all squares in the Board component will allow it to determine the winner in the future.

Since the Square components no longer maintain state, the Square components receive values from the Board component and inform the Board component when they're clicked. In React terms, the Square components are now **controlled components**. The Board has full control over them.

Note how in `handleClick`, we call `.slice()` to create a copy of the `squares` array to modify instead of modifying the existing array. We will explain why we create a copy of the `squares` array in the next section.

## Why Immutability Is Important

In the previous code example, we suggested that you use the `.slice()` operator to create a copy of the `squares` array to modify instead of modifying the existing array. We'll now discuss immutability and why immutability is important to learn.

There are generally two approaches to changing data. The first approach is to *mutate* the data by directly changing the data's values. The second approach is to replace the data with a new copy which has the desired changes.

## Data Change with Mutation

```
var player = {score: 1, name: 'Jeff'};
player.score = 2;
// Now player is {score: 2, name: 'Jeff'}
```

## Data Change without Mutation

```
var player = {score: 1, name: 'Jeff'};

var newPlayer = Object.assign({}, player, {score: 2});
// Now player is unchanged, but newPlayer is {score: 2, name: 'Jeff'}

// Or if you are using object spread syntax proposal, you can write:
// var newPlayer = {...player, score: 2};
```

The end result is the same but by not mutating (or changing the underlying data) directly, we gain several benefits described below.

## Complex Features Become Simple

Immutability makes complex features much easier to implement. Later in this tutorial, we will implement a "time travel" feature that allows us to review the tic-tac-toe game's history and "jump back" to previous moves. This functionality isn't specific to games -- an ability to undo and redo certain actions is a common requirement in applications. Avoiding direct data mutation lets us keep previous versions of the game's history intact, and reuse them later.

## Detecting Changes

Detecting changes in mutable objects is difficult because they are modified directly. This detection requires the mutable object to be compared to previous copies of itself and the entire object tree to be traversed.

Detecting changes in immutable objects is considerably easier. If the immutable object that is being referenced is different than the previous one, then the object has changed.

## Determining When to Re-render in React

The main benefit of immutability is that it helps you build *pure components* in React. Immutable data can easily determine if changes have been made which helps to determine when a component requires re-rendering.

You can learn more about `shouldComponentUpdate()` and how you can build *pure components* by reading [Optimizing Performance](#).

## Functional Components

We'll now change the Square to be a **functional component**.

In React, **functional components** are a simpler way to write components that only contain a `render` method and don't have their own state. Instead of defining a class which extends `React.Component`, we can write a function that takes `props` as input and returns what should be rendered. Functional components are less tedious to write than classes, and many components can be expressed this way.

Replace the Square class with this function:



```
function Square(props) {  
  return (  
    <button className="square" onClick={props.onClick}>  
      {props.value}  
    </button>  
  );  
}
```

We have changed `this.props` to `props` both times it appears.

[View the full code at this point](#)

#### Note

When we modified the Square to be a functional component, we also changed `onClick={() => this.props.onClick()}` to a shorter `onClick={props.onClick}` (note the lack of parentheses on *both* sides). In a class, we used an arrow function to access the correct `this` value, but in a functional component we don't need to worry about `this`.

## Taking Turns

We now need to fix an obvious defect in our tic-tac-toe game: the "O"s cannot be marked on the board.

We'll set the the first move to be "X" by default. We can set this default by modifying the initial state in our Board constructor:

```
class Board extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state = {  
      squares: Array(9).fill(null),  
      xIsNext: true,  
    };  
  }  
}
```

Each time a player moves, `xIsNext` (a boolean) will be flipped to determine which player goes next and the game's state will be saved. We'll update the Board's `handleClick` function to flip the value of `xIsNext`:

```
handleClick(i) {  
  const squares = this.state.squares.slice();  
  squares[i] = this.state.xIsNext ? 'X' : 'O';  
  this.setState({  
    squares: squares,  
    xIsNext: !this.state.xIsNext,  
  });  
}
```

With this change, "X"s and "O"s can take turns. Let's also change the "status" text in Board's `render` so that it displays which player has the next turn:

```
render() {  
  const status = 'Next player: ' + (this.state.xIsNext ? 'X' : 'O');
```

```
    return (  
      // the rest has not changed
```

After applying these changes, you should have this Board component:

```
class Board extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state = {  
      squares: Array(9).fill(null),  
      xIsNext: true,  
    };  
  }  
  
  handleClick(i) {  
    const squares = this.state.squares.slice();  
    squares[i] = this.state.xIsNext ? 'X' : 'O';  
    this.setState({  
      squares: squares,  
      xIsNext: !this.state.xIsNext,  
    });  
  }  
  
  renderSquare(i) {  
    return (  
      <Square  
        value={this.state.squares[i]}  
        onClick={() => this.handleClick(i)}  
      />  
    );  
  }  
  
  render() {  
    const status = 'Next player: ' + (this.state.xIsNext ? 'X' : 'O');  
  
    return (  
      <div>  
        <div className="status">{status}</div>  
        <div className="board-row">  
          {this.renderSquare(0)}  
          {this.renderSquare(1)}  
          {this.renderSquare(2)}  
        </div>  
        <div className="board-row">  
          {this.renderSquare(3)}  
          {this.renderSquare(4)}  
          {this.renderSquare(5)}  
        </div>  
        <div className="board-row">  
          {this.renderSquare(6)}
```

```

        {this.renderSquare(7)}
        {this.renderSquare(8)}
      </div>
    </div>
  );
}
}

```

[View the full code at this point](#)

## Declaring a Winner

Now that we show which player's turn is next, we should also show when the game is won and there are no more turns to make. We can determine a winner by adding this helper function to the end of the file:

```

function calculateWinner(squares) {
  const lines = [
    [0, 1, 2],
    [3, 4, 5],
    [6, 7, 8],
    [0, 3, 6],
    [1, 4, 7],
    [2, 5, 8],
    [0, 4, 8],
    [2, 4, 6],
  ];
  for (let i = 0; i < lines.length; i++) {
    const [a, b, c] = lines[i];
    if (squares[a] && squares[a] === squares[b] && squares[a] === squares[c]) {
      return squares[a];
    }
  }
  return null;
}

```

We will call `calculateWinner(squares)` in the Board's `render` function to check if a player has won. If a player has won, we can display text such as "Winner: X" or "Winner: O". We'll replace the `status` declaration in Board's `render` function with this code:

```

render() {
  const winner = calculateWinner(this.state.squares);
  let status;
  if (winner) {
    status = 'Winner: ' + winner;
  } else {
    status = 'Next player: ' + (this.state.xIsNext ? 'X' : 'O');
  }

  return (
    // the rest has not changed

```

We can now change the Board's `handleClick` function to return early by ignoring a click if someone has won the game or if a Square is already filled:

```
handleClick(i) {
  const squares = this.state.squares.slice();
  if (calculateWinner(squares) || squares[i]) {
    return;
  }
  squares[i] = this.state.xIsNext ? 'X' : 'O';
  this.setState({
    squares: squares,
    xIsNext: !this.state.xIsNext,
  });
}
```

[View the full code at this point](#)

Congratulations! You now have a working tic-tac-toe game. And you've just learned the basics of React too. So *you're* probably the real winner here.

## Adding Time Travel

As a final exercise, let's make it possible to "go back in time" to the previous moves in the game.

### Storing a History of Moves

If we mutated the `squares` array, implementing time travel would be very difficult.

However, we used `slice()` to create a new copy of the `squares` array after every move, and [treated it as immutable](#). This will allow us to store every past version of the `squares` array, and navigate between the turns that have already happened.

We'll store the past `squares` arrays in another array called `history`. The `history` array represents all board states, from the first to the last move, and has a shape like this:

```
history = [
  // Before first move
  {
    squares: [
      null, null, null,
      null, null, null,
      null, null, null,
    ]
  },
  // After first move
  {
    squares: [
      null, null, null,
      null, 'X', null,
      null, null, null,
    ]
  },
]
```

```
// After second move
{
  squares: [
    null, null, null,
    null, 'X', null,
    null, null, 'O',
  ]
},
// ...
]
```

Now we need to decide which component should own the `history` state.

## Lifting State Up, Again

We'll want the top-level Game component to display a list of past moves. It will need access to the `history` to do that, so we will place the `history` state in the top-level Game component.

Placing the `history` state into the Game component lets us remove the `squares` state from its child Board component. Just like we "lifted state up" from the Square component into the Board component, we are now lifting it up from the Board into the top-level Game component. This gives the Game component full control over the Board's data, and lets it instruct the Board to render previous turns from the `history`.

First, we'll set up the initial state for the Game component within its constructor:

```
class Game extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      history: [{
        squares: Array(9).fill(null),
      }],
      xIsNext: true,
    };
  }

  render() {
    return (
      <div className="game">
        <div className="game-board">
          <Board />
        </div>
        <div className="game-info">
          <div>{/* status */</div>
          <ol>{/* TODO */</ol>
        </div>
      </div>
    );
  }
}
```

Next, we'll have the Board component receive `squares` and `onClick` props from the Game component. Since we now have a single click handler in Board for many Squares, we'll need to pass the location of each Square into the `onClick` handler to indicate which Square was clicked. Here are the required steps to transform the Board component:

- Delete the `constructor` in Board.
- Replace `this.state.squares[i]` with `this.props.squares[i]` in Board's `renderSquare`.
- Replace `this.handleClick(i)` with `this.props.onClick(i)` in Board's `renderSquare`.

The Board component now looks like this:

```
class Board extends React.Component {
  handleClick(i) {
    const squares = this.state.squares.slice();
    if (calculateWinner(squares) || squares[i]) {
      return;
    }
    squares[i] = this.state.xIsNext ? 'X' : 'O';
    this.setState({
      squares: squares,
      xIsNext: !this.state.xIsNext,
    });
  }

  renderSquare(i) {
    return (
      <Square
        value={this.props.squares[i]}
        onClick={() => this.props.onClick(i)}
      />
    );
  }

  render() {
    const winner = calculateWinner(this.state.squares);
    let status;
    if (winner) {
      status = 'Winner: ' + winner;
    } else {
      status = 'Next player: ' + (this.state.xIsNext ? 'X' : 'O');
    }

    return (
      <div>
        <div className="status">{status}</div>
        <div className="board-row">
          {this.renderSquare(0)}
          {this.renderSquare(1)}
          {this.renderSquare(2)}
        </div>
        <div className="board-row">
          {this.renderSquare(3)}
          {this.renderSquare(4)}
        </div>
      </div>
    );
  }
}
```

```

        {this.renderSquare(5)}
      </div>
      <div className="board-row">
        {this.renderSquare(6)}
        {this.renderSquare(7)}
        {this.renderSquare(8)}
      </div>
    </div>
  </div>
);
}
}

```

We'll update the Game component's `render` function to use the most recent history entry to determine and display the game's status:

```

render() {
  const history = this.state.history;
  const current = history[history.length - 1];
  const winner = calculateWinner(current.squares);

  let status;
  if (winner) {
    status = 'Winner: ' + winner;
  } else {
    status = 'Next player: ' + (this.state.xIsNext ? 'X' : 'O');
  }

  return (
    <div className="game">
      <div className="game-board">
        <Board
          squares={current.squares}
          onClick={(i) => this.handleClick(i)}
        />
      </div>
      <div className="game-info">
        <div>{status}</div>
        <ol>{/* TODO */}</ol>
      </div>
    </div>
  );
}

```

Since the Game component is now rendering the game's status, we can remove the corresponding code from the Board's `render` method. After refactoring, the Board's `render` function looks like this:

```

render() {
  return (
    <div>
      <div className="board-row">

```

```

        {this.renderSquare(0)}
        {this.renderSquare(1)}
        {this.renderSquare(2)}
      </div>
      <div className="board-row">
        {this.renderSquare(3)}
        {this.renderSquare(4)}
        {this.renderSquare(5)}
      </div>
      <div className="board-row">
        {this.renderSquare(6)}
        {this.renderSquare(7)}
        {this.renderSquare(8)}
      </div>
    </div>
  );
}

```

Finally, we need to move the `handleClick` method from the Board component to the Game component. We also need to modify `handleClick` because the Game component's state is structured differently. Within the Game's `handleClick` method, we concatenate new history entries onto `history`.

```

handleClick(i) {
  const history = this.state.history;
  const current = history[history.length - 1];
  const squares = current.squares.slice();
  if (calculateWinner(squares) || squares[i]) {
    return;
  }
  squares[i] = this.state.xIsNext ? 'X' : 'O';
  this.setState({
    history: history.concat([
      {
        squares: squares,
      }
    ]),
    xIsNext: !this.state.xIsNext,
  });
}

```

#### Note

Unlike the array `push()` method you might be more familiar with, the `concat()` method doesn't mutate the original array, so we prefer it.

At this point, the Board component only needs the `renderSquare` and `render` methods. The game's state and the `handleClick` method should be in the Game component.

[View the full code at this point](#)

## Showing the Past Moves

Since we are recording the tic-tac-toe game's history, we can now display it to the player as a list of past moves.



We learned earlier that React elements are first-class JavaScript objects; we can pass them around in our applications. To render multiple items in React, we can use an array of React elements.

In JavaScript, arrays have a `map()` method that is commonly used for mapping data to other data, for example:

```
const numbers = [1, 2, 3];
const doubled = numbers.map(x => x * 2); // [2, 4, 6]
```

Using the `map` method, we can map our history of moves to React elements representing buttons on the screen, and display a list of buttons to "jump" to past moves.

Let's `map` over the `history` in the Game's `render` method:

```
render() {
  const history = this.state.history;
  const current = history[history.length - 1];
  const winner = calculateWinner(current.squares);

  const moves = history.map((step, move) => {
    const desc = move ?
      'Go to move #' + move :
      'Go to game start';
    return (
      <li>
        <button onClick={() => this.jumpTo(move)}>{desc}</button>
      </li>
    );
  });

  let status;
  if (winner) {
    status = 'Winner: ' + winner;
  } else {
    status = 'Next player: ' + (this.state.xIsNext ? 'X' : 'O');
  }

  return (
    <div className="game">
      <div className="game-board">
        <Board
          squares={current.squares}
          onClick={(i) => this.handleClick(i)}
        />
      </div>
      <div className="game-info">
        <div>{status}</div>
        <ol>{moves}</ol>
      </div>
    </div>
  );
}
```

### [View the full code at this point](#)

For each move in the tic-tac-toes's game's history, we create a list item `<li>` which contains a button `<button>`. The button has a `onClick` handler which calls a method called `this.jumpTo()`. We haven't implemented the `jumpTo()` method yet. For now, we should see a list of the moves that have occurred in the game and a warning that says:

Warning: Each child in an array or iterator should have a unique "key" prop. Check the render method of "Game".

Let's discuss what the above warning means.

## Picking a Key

When we render a list, React stores some information about each rendered list item. When we update a list, React needs to determine what has changed. We could have added, removed, re-arranged, or updated the list's items.

Imagine transitioning from

```
<li>Alexa: 7 tasks left</li>
<li>Ben: 5 tasks left</li>
```

to

```
<li>Ben: 9 tasks left</li>
<li>Claudia: 8 tasks left</li>
<li>Alexa: 5 tasks left</li>
```

From our perspective, our transition swapped Alexa and Ben's ordering and inserted Claudia between Alexa and Ben. However, React is a computer program and does not know what we intended. Because React cannot know our intentions, we need to specify a `key` property for each list item to differentiate each list item from its siblings. The strings `alexa`, `ben`, `claudia` may be used as keys. If we had access to a database, Alexa, Ben, and Claudia's database IDs could be used as keys.

```
<li key={user.id}>{user.name}: {user.taskCount} tasks left</li>
```

`key` is a special and reserved property in React (along with `ref`, a more advanced feature). When an element is created, React extracts the `key` property and stores the key directly on the returned element. Even though `key` may look like it belongs in `props`, `key` cannot be referenced using `this.props.key`. React automatically uses `key` to decide which components to update. A component cannot inquire about its `key`.

When a list is re-rendered, React takes each list item's key and searches the previous list's items for a matching key. If the current list has a key that does not exist in the previous list, React creates a component. If the current list is missing a key that exists in the previous list, React destroys a component. Keys tell React about the identity of each component which allows React to maintain state between re-renders. If a component's key changes, the component will be destroyed and re-created with a new state.

**It's strongly recommended that you assign proper keys whenever you build dynamic lists.** If you don't have an appropriate key, you may want to consider restructuring your data so that you do.

If no key is specified, React will present a warning and use the array index as a key by default. Using the array index as a key is problematic when trying to re-order a list's items or inserting/removing list items. Explicitly passing `key={i}` silences the warning but has the same problems as array indices and is not recommended in most cases.

Keys do not need to be globally unique. Keys only need to be unique between components and their siblings.

## Implementing Time Travel

In the tic-tac-toe game's history, each past move has a unique ID associated with it: it's the sequential number of the move. The moves are never re-ordered, deleted, or inserted in the middle, so it's safe to use the move index as a key.

In the Game component's `render` method, we can add the key as `<li key={move}>` and React's warning about keys should disappear:

```
const moves = history.map((step, move) => {
  const desc = move ?
    'Go to move #' + move :
    'Go to game start';
  return (
    <li key={move}>
      <button onClick={() => this.jumpTo(move)}>{desc}</button>
    </li>
  );
});
```

[View the full code at this point](#)

Clicking any of the list item's buttons throws an error because the `jumpTo` method is undefined. Before we implement `jumpTo`, we'll add `stepNumber` to the Game component's state to indicate which step we're currently viewing.

First, add `stepNumber: 0` to the initial state in Game's `constructor`:

```
class Game extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      history: [{
        squares: Array(9).fill(null),
      }],
      stepNumber: 0,
      xIsNext: true,
    };
  }
}
```

Next, we'll define the `jumpTo` method in Game to update that `stepNumber`. We also set `xIsNext` to true if the number that we're changing `stepNumber` to is even:

```
handleClick(i) {
  // this method has not changed
}

jumpTo(step) {
  this.setState({
    stepNumber: step,
    xIsNext: (step % 2) === 0,
  });
}

render() {
```

```
// this method has not changed  
}
```

We will now make a few changes to the Game's `handleClick` method which fires when you click on a square.

The `stepNumber` state we've added reflects the move displayed to the user now. After we make a new move, we need to update `stepNumber` by adding `stepNumber: history.length` as part of the `this.setState` argument. This ensures we don't get stuck showing the same move after a new one has been made.

We will also replace reading `this.state.history` with `this.state.history.slice(0, this.state.stepNumber + 1)`. This ensures that if we "go back in time" and then make a new move from that point, we throw away all the "future" history that would now become incorrect.

```
handleClick(i) {  
  const history = this.state.history.slice(0, this.state.stepNumber + 1);  
  const current = history[history.length - 1];  
  const squares = current.squares.slice();  
  if (calculateWinner(squares) || squares[i]) {  
    return;  
  }  
  squares[i] = this.state.xIsNext ? 'X' : 'O';  
  this.setState({  
    history: history.concat([  
      squares: squares  
    ]),  
    stepNumber: history.length,  
    xIsNext: !this.state.xIsNext,  
  });  
}
```

Finally, we will modify the Game component's `render` method from always rendering the last move to rendering the currently selected move according to `stepNumber`:

```
render() {  
  const history = this.state.history;  
  const current = history[this.state.stepNumber];  
  const winner = calculateWinner(current.squares);  
  
  // the rest has not changed
```

If we click on any step in the game's history, the tic-tac-toe board should immediately update to show what the board looked like after that step occurred.

[View the full code at this point](#)

## Wrapping Up

Congratulations! You've created a tic-tac-toe game that:

- Lets you play tic-tac-toe,
- Indicates when a player has won the game,
- Stores a game's history as a game progresses,

- Allows players to review a game's history and see previous versions of a game's board.

Nice work! We hope you now feel like you have a decent grasp on how React works.

Check out the final result here: [Final Result](#).

If you have extra time or want to practice your new React skills, here are some ideas for improvements that you could make to the tic-tac-toe game which are listed in order of increasing difficulty:

1. Display the location for each move in the format (col, row) in the move history list.
2. Bold the currently selected item in the move list.
3. Rewrite Board to use two loops to make the squares instead of hardcoding them.
4. Add a toggle button that lets you sort the moves in either ascending or descending order.
5. When someone wins, highlight the three squares that caused the win.
6. When no one wins, display a message about the result being a draw.

Throughout this tutorial, we touched on React concepts including elements, components, props, and state. For a more detailed explanation of each of these topics, check out [the rest of the documentation](#). To learn more about defining components, check out the `React.Component` [API reference](#).

The smallest React example looks like this:

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

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 of the 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.

## Let's Get Started!

Keep scrolling down, and you'll find the link to the [next chapter of this guide](#) right before the website footer.

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>;

ReactDOM.render(
  element,
  document.getElementById('root')
);
```

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>
);

ReactDOM.render(
  element,
  document.getElementById('root')
);
```

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 = <div tabIndex="0"></div>;
```

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. This website uses the [Oceanic Next](#) color scheme which is compatible with it.

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 into a root DOM node, pass both to `ReactDOM.render()` :

```
embed:rendering-elements/render-an-element.js
```

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 `ReactDOM.render()` .

Consider this ticking clock example:

```
embed:rendering-elements/update-rendered-element.js
```

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

**Note:**

In practice, most React apps only call `ReactDOM.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.

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:

# Hello, world!

## It is 12:26:46 PM.



Even though we create an element describing the whole UI tree on every tick, only the text node whose contents has 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.

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.

## Functional 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 "functional" 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.

Classes have some additional features that we will discuss in the [next sections](#). Until then, we will use functional components for their conciseness.

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

```
ReactDOM.render(  
  element,  
  document.getElementById( 'root' )  
);
```

Let's recap what happens in this example:

1. We call `ReactDOM.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.

You can read more about the reasoning behind this convention [here](#).

## 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>  
  );  
}  
  
ReactDOM.render(  
  <App />,  
  document.getElementById( 'root' )  
);
```

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 a reusable 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.

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 `ReactDOM.render()` to change the rendered output:

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

setInterval(tick, 1000);
```

#### [Try it on CodePen](#)

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:

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

function tick() {
  ReactDOM.render(
    <Clock date={new Date()} />,
    document.getElementById('root')
  );
}

setInterval(tick, 1000);
```

#### [Try it on CodePen](#)

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:

---

```
ReactDOM.render(  
  <Clock />,  
  document.getElementById('root')  
);
```

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.

We [mentioned before](#) that components defined as classes have some additional features. Local state is exactly that: a feature available only to classes.

## Converting a Function to a Class

You can convert a functional 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>  
    );  
  }  
}
```

[Try it on CodePen](#)

`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 hooks.

## 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>
  );
}
}

```

2) 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` .

3) Remove the `date` prop from the `<Clock />` element:

```

ReactDOM.render(
  <Clock />,
  document.getElementById('root')
);

```

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>
      );
    }
  }

ReactDOM.render(
  <Clock />,
  document.getElementById('root')
);
```

[Try it on CodePen](#)

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 hooks".

The `componentDidMount()` hook 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`.

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 hook:

```
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>
  );
}

ReactDOM.render(
  <Clock />,
  document.getElementById('root')
);
```

[Try it on CodePen](#)

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 `ReactDOM.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 hook. 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 hook 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((prevState, props) => ({
  counter: prevState.counter + props.increment
}));
```

We used an [arrow function](#) above, but it also works with regular functions:

```
// Correct
this.setState(function(prevState, props) {
  return {
    counter: prevState.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:

```
<h2>It is {this.state.date.toLocaleTimeString()}.</h2>
```

This also works for user-defined 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>;
}
```

[Try it on CodePen](#)

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>`s:

```
function App() {  
  return (  
    <div>  
      <Clock />  
      <Clock />  
      <Clock />  
    </div>  
  );  
}  
  
ReactDOM.render(  
  <App />,  
  document.getElementById( 'root' )  
);
```

[Try it on CodePen](#)

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.

Handling events with React elements is very similar to handling events on DOM elements. There are some syntactic 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 link behavior of opening a new page, you can write:

```
<a href="#" onclick="console.log('The link was clicked.');" return false">
  Click me
</a>
```

In React, this could instead be:

```
function ActionLink() {
  function handleClick(e) {
    e.preventDefault();
    console.log('The link was clicked.');
```

```
  }

  return (
    <a href="#" onClick={handleClick}>
      Click me
    </a>
  );
}
```

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. See the [SyntheticEvent](#) reference guide to learn more.

When using React you should generally not 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>
    );
  }
}

ReactDOM.render(
  <Toggle />,
  document.getElementById('root')
);

```

### Try it on CodePen

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. If you are using the experimental [public class fields syntax](#), you can use class fields to correctly bind callbacks:

```

class LoggingButton extends React.Component {
  // This syntax ensures `this` is bound within handleClick.
  // Warning: this is *experimental* syntax.
  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={(e) => this.handleClick(e)}>
        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.

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 />;
}

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

[Try it on CodePen](#)

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>
    );
  }
}

ReactDOM.render(
  <LoginControl />,
  document.getElementById('root')
);

```

### Try it on CodePen

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

### Try it on CodePen

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.

## 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);
  }

  handleClick() {
    this.setState(prevState => ({
      showWarning: !prevState.showWarning
    }));
  }
}
```

```
    }));  
  }  
  
  render() {  
    return (  
      <div>  
        <WarningBanner warn={this.state.showWarning} />  
        <button onClick={this.handleToggleClick}>  
          {this.state.showWarning ? 'Hide' : 'Show'}  
        </button>  
      </div>  
    );  
  }  
}  
  
ReactDOM.render(  
  <Page />,  
  document.getElementById('root')  
);
```

### [Try it on CodePen](#)

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.

HTML form elements work a little 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>
  );
}
```

### [Try it on CodePen](#)

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, every state mutation will have an associated handler function. This makes it straightforward to modify or validate user input. For example, if we wanted to enforce that names are written with all uppercase letters, we could write `handleChange` as:

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

## 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>
  <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.handleInputChange = this.handleInputChange.bind(this);
}

handleInputChange(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.handleInputChange} />
      </label>
      <br />
      <label>
        Number of guests:
        <input
          name="numberOfGuests"
          type="number"
          value={this.state.numberOfGuests}
          onChange={this.handleInputChange} />
      </label>
    </form>
  );
}
```

[Try it on CodePen](#)

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.render(<input value="hi" />, mountNode);  
  
setTimeout(function() {  
  ReactDOM.render(<input value={null} />, mountNode);  
}, 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.



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>
    );
  }
}
```

[Try it on CodePen](#)

## 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>
    );
  }
}
```

[Try it on CodePen](#)

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>
    );
  }
}

```

### Try it on CodePen

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` has 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.

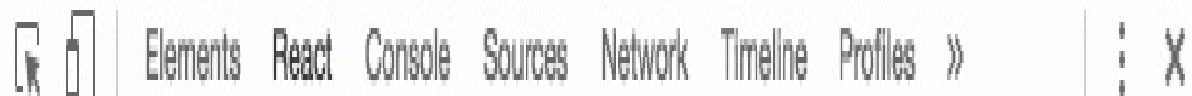
If something can be derived from either props or state, it probably shouldn't be in the state. For example, instead of storing both `celsiusValue` and `fahrenheitValue`, we store just the last edited `temperature` and its `scale`. The value of the other input can always be calculated from them in the `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:

Enter temperature in Celsius: \_\_\_\_\_

Enter temperature in Fahrenheit: \_\_\_\_\_

The water would not boil.



☐ Trace React Updates ☐ Highlight Search ☐ Use Regular Expressions

▼ **<Calculator>**

▼ **<div>**

▶ **<TemperatureInput** scale="c" temperature="" onTemperatureChan

▶ **<TemperatureInput** scale="f" temperature="" onTemperatureChan

▶ **<BoilingVerdict** celsius=null>...**</BoilingVerdict>**

**</div>**

**</Calculator>**

**<Calculator>**

(\$r in the console)

**Props**

Empty object

**State**

scale: "c"

temperature: ""

Calculator





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>
  );
}
```

[Try it on CodePen](#)

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 />
      } />
  );
}
```

[Try it on CodePen](#)

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!" />
  );
}
```

[Try it on CodePen](#)

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}!`);
  }
}
```

[Try it on CodePen](#)

## 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 a class, without extending it.

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:

Name	Price
<b>Sporting Goods</b>	
Football	\$49.99
Baseball	\$9.99
Basketball	\$29.99
<b>Electronics</b>	
iPod Touch	\$99.99
iPhone 5	\$399.99
Nexus 7	\$199.99

Our JSON API returns some data that looks like this:

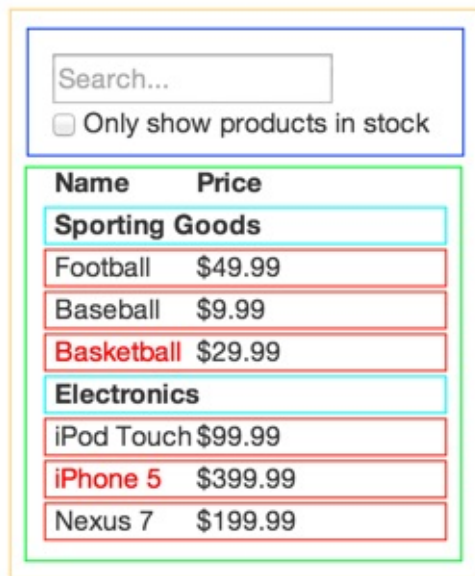
```
[
  {category: "Sporting Goods", price: "$49.99", stocked: true, name: "Football"},
  {category: "Sporting Goods", price: "$9.99", stocked: true, name: "Baseball"},
  {category: "Sporting Goods", price: "$29.99", stocked: false, name:
"Basketball"},
  {category: "Electronics", price: "$99.99", stocked: 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? Just 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*, which means the work of separating your UI into components is often trivial. Just break it up into components that represent exactly one piece of your data model.



You'll see here that we have five components in our simple app. We've italicized the data each component represents.

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 (i.e. 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. This is easy. 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 `ReactDOM.render()` again, the UI will be updated. It's easy to see how your UI is updated and where to make changes since there's nothing complicated going on. React's **one-way data flow** (also called **one-way binding**) keeps everything modular and fast. Simply refer to the [React docs](/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](/docs/interactivity-and-dynamic-uis.html) if you aren't sure what the difference is. **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 makes this easy 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\**](https://en.wikipedia.org/wiki/Don%27t\_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, just 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, simply take the length of the TODO items array. Think of all of 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. Simply 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 simply 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 make it easy to 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 current version of the example, 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.

Though this sounds complex, it's really just a few lines of code. And it's really explicit how your data is flowing throughout the app.

## 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 than it's written, and it's extremely easy 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. :)

## 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 ( `<ol>` , `<ul>` 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 => (
        // 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 also 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.

```
function CustomTextInput(props) {
  return (
    <div>
      <input ref={props.inputRef} />
    </div>
  );
}

class Parent extends React.Component {
  constructor(props) {
    super(props);
    this.inputElement = React.createRef();
  }
  render() {
    return (
      <CustomTextInput inputRef={this.inputElement} />
    );
  }
}
```

```
// Now you can set focus when required.  
this.inputElement.current.focus();
```

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.

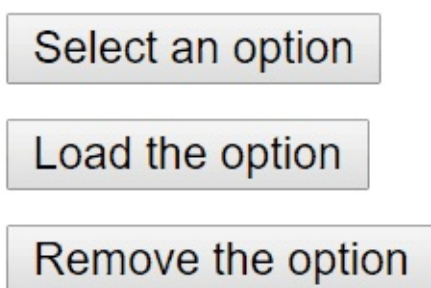
Note:

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.



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>
      ) : null}
    </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 an 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);
}



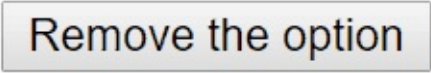
render() {
  // React assists us by bubbling the blur and
  // focus events to the parent.
  return (
    <div onBlur={this.onBlurHandler}
      onFocus={this.onFocusHandler}>
      <button onClick={this.onClickHandler}
        aria-haspopup="true"
        aria-expanded={this.state.isOpen}>
        Select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

```



```
    ) : null}
  </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.

A rectangular button with a light gray background and a thin gray border. The text "Select an option" is centered in a dark gray, sans-serif font.A rectangular button with a light gray background and a thin gray border. The text "Load the option" is centered in a dark gray, sans-serif font.A rectangular button with a light gray background and a thin gray border. The text "Remove the option" is centered in a dark gray, sans-serif font.

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:

- [WAI-ARIA Authoring Practices - Design Patterns and Widgets](#)
- [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. Plugging out 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 [react-axe](#) 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:

- [Activate 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 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](#)

## Bundling

Most React apps will have their files "bundled" using tools like [Webpack](#) 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 setup 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 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-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));
});
```

#### Note:

The dynamic `import()` syntax is a ECMAScript (JavaScript) [proposal](#) not currently part of the language standard. It is expected to be accepted in the near future.

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](#).

## Libraries

### React Loadable

[React Loadable](#) wraps dynamic imports in a nice, React-friendly API for introducing code splitting into your app at a given component.

### Before:

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

const MyComponent = () => (
  <OtherComponent/>
);
```

### After:

```
import Loadable from 'react-loadable';
```

```
const LoadableOtherComponent = Loadable({
  loader: () => import('./OtherComponent'),
  loading: () => <div>Loading...</div>,
});

const MyComponent = () => (
  <LoadableOtherComponent/>
);
```

React Loadable helps you create [loading states](#), [error states](#), [timeouts](#), [preloading](#), and more. It can even help you [server-side render](#) an app with lots of code-splitting.

## 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](#) and [React Loadable](#).

```
import { BrowserRouter as Router, Route, Switch } from 'react-router-dom';
import Loadable from 'react-loadable';

const Loading = () => <div>Loading...</div>;

const Home = Loadable({
  loader: () => import('./routes/Home'),
  loading: Loading,
});

const About = Loadable({
  loader: () => import('./routes/About'),
  loading: Loading,
});

const App = () => (
  <Router>
    <Switch>
      <Route exact path="/" component={Home}/>
      <Route path="/about" component={About}/>
    </Switch>
  </Router>
);
```

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 this 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](#)
  - [Provider](#)
  - [Consumer](#)
- [Examples](#)
  - [Dynamic Context](#)
  - [Updating Context from a Nested Component](#)
  - [Consuming Multiple Contexts](#)
  - [Accessing Context in Lifecycle Methods](#)
  - [Consuming Context with a HOC](#)
  - [Forwarding Refs to Context Consumers](#)
- [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:

```
embed:context/motivation-problem.js
```

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

```
embed:context/motivation-solution.js
```

## 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>
```



```
</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` prop:

```
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} />
// ... 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. However, this 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 {Provider, Consumer} = React.createContext(defaultValue);
```

Creates a `{ Provider, Consumer }` pair. When React renders a context `Consumer`, it will read the current context value from the closest matching `Provider` above it in the tree.

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

### Provider

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

A React component that allows `Consumers` to subscribe to context changes.

Accepts a `value` prop to be passed to `Consumers` 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.

### Consumer

```

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

```

A React component that subscribes to context changes.

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](#).

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 is not subject to the `shouldComponentUpdate` method, so the Consumer is updated even when an ancestor component bails out of the 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](#).

## Examples

### Dynamic Context

A more complex example with dynamic values for the theme:

**theme-context.js** `embed:context/theme-detailed-theme-context.js`

**themed-button.js** `embed:context/theme-detailed-themed-button.js`

**app.js** `embed:context/theme-detailed-app.js`

### 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** `embed:context/updating-nested-context-context.js`

**theme-toggler-button.js** `embed:context/updating-nested-context-theme-toggler-button.js`

**app.js** `embed:context/updating-nested-context-app.js`

### Consuming Multiple Contexts

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

`embed:context/multiple-contexts.js`

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

### Accessing Context in Lifecycle Methods

Accessing values from context in lifecycle methods is a relatively common use case. Instead of adding context to every lifecycle method, you just need to pass it as a prop, and then work with it just like you'd normally work with a prop.

`embed:context/lifecycles.js`

### Consuming Context with a HOC

Some types of contexts are consumed by many components (e.g. theme or localization). It can be tedious to explicitly wrap each dependency with a `<Context.Consumer>` element. A [higher-order component](#) can help with this.

For example, a button component might consume a theme context like so:

`embed:context/higher-order-component-before.js`

That's alright for a few components, but what if we wanted to use the theme context in a lot of places?

We could create a higher-order component called `withTheme` :

```
embed:context/higher-order-component.js
```

Now any component that depends on the theme context can easily subscribe to it using the `withTheme` function we've created:

```
embed:context/higher-order-component-usage.js
```

## Forwarding Refs to Context Consumers

One issue with the render prop API is that refs don't automatically get passed to wrapped elements. To get around this, use

```
React.forwardRef :
```

**fancy-button.js** `embed:context/forwarding-refs-fancy-button.js`

**app.js** `embed:context/forwarding-refs-app.js`

## 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` :

```
embed:context/reference-caveats-problem.js
```

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

```
embed:context/reference-caveats-solution.js
```

## 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](#).

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 a new lifecycle method called `componentDidCatch(error, info) :`

```
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  componentDidCatch(error, info) {
    // Display fallback UI
    this.setState({ hasError: true });
    // You can also log the error to an error reporting service
    logErrorToMyService(error, info);
  }

  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>
```

The `componentDidCatch()` method works 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 `catch {}` block works in JavaScript.

## componentDidCatch Parameters

`error` is an error that has been thrown.

`info` is an object with `componentStack` key. The property has information about component stack during thrown error.

```
// ...
componentDidCatch(error, info) {

  /* Example stack information:
    in ComponentThatThrows (created by App)
    in ErrorBoundary (created by App)
    in div (created by App)
    in App
  */
  logComponentStackToMyService(info.componentStack);
}

// ...
```

## Live Demo

Check out [this example of declaring and using an error boundary](#) with [React 16](#).

## 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 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` hook 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 hooks, 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 event handler, use the regular JavaScript `try / catch` statement:

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

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

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

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.



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: `embed:forwarding-refs/fancy-button-simple.js`

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:

```
embed:forwarding-refs/fancy-button-simple-ref.js
```

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 functional 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: `embed:forwarding-refs/log-props-before.js`

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: `embed:forwarding-refs/fancy-button.js`

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: `embed:forwarding-refs/fancy-button-ref.js`

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: `embed:forwarding-refs/log-props-after.js`

## 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:

`embed:forwarding-refs/wrapped-component.js`

If you name the render function, DevTools will also include its name (e.g. "*ForwardRef(myFunction)*"): `embed:forwarding-refs/wrapped-component-with-function-name.js`

You can even set the function's `displayName` property to include the component you're wrapping:

`embed:forwarding-refs/customized-display-name.js`

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, but it isn't supported by all popular tools yet.

## 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>
```

So, we introduce `Fragment`s.

## 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.

Note that **many tools don't support it yet** so you might want to explicitly write `<React.Fragment>` until the tooling catches up.

## 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.

## Live Demo

You can try out the new JSX fragment syntax with this [CodePen](#).

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) => DataSource.getComments()
);

const BlogPostWithSubscription = withSubscription(
  BlogPost,
  (DataSource, props) => 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.componentWillReceiveProps = function(nextProps) {
    console.log('Current props: ', this.props);
    console.log('Next props: ', nextProps);
  };
  // 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 `componentWillReceiveProps`, the first HOC's functionality will be overridden! This HOC also won't work with functional 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 {
    componentWillReceiveProps(nextProps) {
      console.log('Current props: ', this.props);
      console.log('Next props: ', nextProps);
    }
  };
}

```

```
    }  
    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 functional 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.](#)



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 hooks](#). 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 hooks. 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');
}
```

### Try it on CodePen

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);
}
```

### Try it on CodePen

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 hook 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>
    );
  }
}
```

[Try it on CodePen](#)

## Integrating with Other View Libraries

React can be embedded into other applications thanks to the flexibility of `ReactDOM.render()`.

Although React is commonly used at startup to load a single root React component into the DOM, `ReactDOM.render()` 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>;
}

ReactDOM.render(
  <Button />,
  document.getElementById('container'),
  function() {
    $('#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} />;
}

ReactDOM.render(
  <HelloButton />,
  document.getElementById('container')
);
```

[Try it on CodePen](#)

You can have as many such isolated components as you like, and use `ReactDOM.render()` 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.render()` 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.render()`:

```
function Paragraph(props) {
  return <p>{props.text}</p>;
}

const ParagraphView = Backbone.View.extend({
  render() {
    const text = this.model.get('text');
    ReactDOM.render(<Paragraph text={text} />, this.el);
    return this;
  },
  remove() {
    ReactDOM.unmountComponentAtNode(this.el);
    Backbone.View.prototype.remove.call(this);
  }
});
```

[Try it on CodePen](#)

It is important that we also call `ReactDOM.unmountComponentAtNode()` 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>
    );
  }
}
```

```

    );
  }
}

```

[Try it on CodePen](#)

## 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' });
ReactDOM.render(
  <Example model={model} />,
  document.getElementById('root')
);

```

[Try it on CodePen](#)

This technique is not limited to Backbone. You can use React with any model library by subscribing to its changes in the lifecycle hooks and, optionally, copying the data into the local React state.

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'},
  null
)
```

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 `<span>` 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-unescaped. 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 using this 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" operator 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 operator.

```
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 & 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>
  Hello
  World
</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 only renders a `<Header />` 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>
```



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.

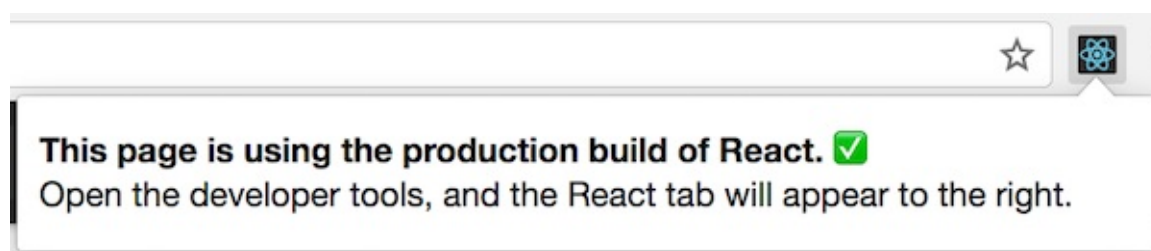
## Use the Production Build

If you're benchmarking or experiencing performance problems in your React apps, make sure you're testing with the minified production build.

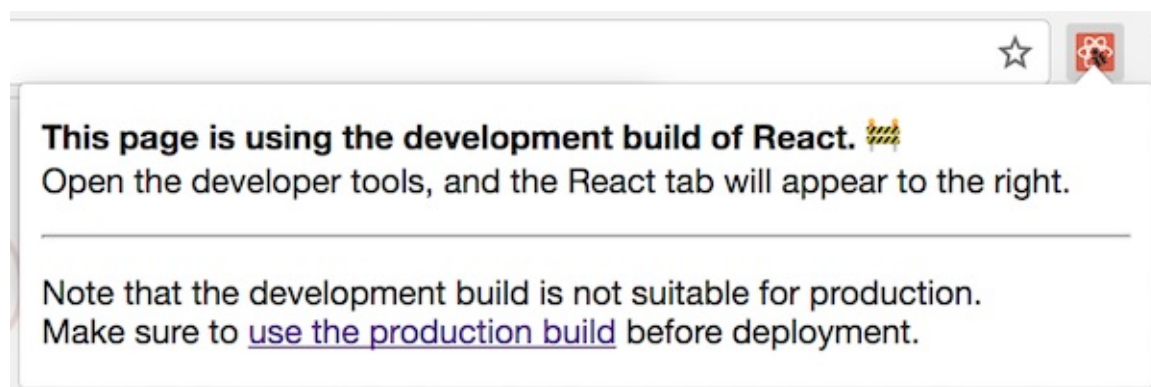
By default, React includes many helpful warnings. These warnings are very useful in development. However, they make React larger and slower so you should make sure to use the production version when you deploy the app.

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@16/umd/react.production.min.js"></script>
<script src="https://unpkg.com/react-dom@16/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 `uglify-js-brunch` plugin:

```
# If you use npm
npm install --save-dev uglify-js-brunch

# If you use Yarn
yarn add --dev uglify-js-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 uglify-js uglifyify

# If you use Yarn
yarn add --dev envify uglify-js 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 `uglify-js` for mangling ([read why](#)).

For example:

```
browserify ./index.js \
  -g [ envify --NODE_ENV production ] \
  -g uglifyify \
  | uglifyjs --compress --mangle > ./bundle.js
```

### Note:

The package name is `uglify-js`, but the binary it provides is called `uglifyjs`. This is not a typo.

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-uglify

# If you use Yarn
yarn add --dev rollup-plugin-commonjs rollup-plugin-replace rollup-plugin-uglify
```

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 `uglify` 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-uglify')(),
  // ...
]
```

For a complete setup example [see this gist](#).

Remember that you only need to do this for production builds. You shouldn't apply the `uglify` 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.

For the most efficient webpack production build, make sure to include these plugins in your production configuration:

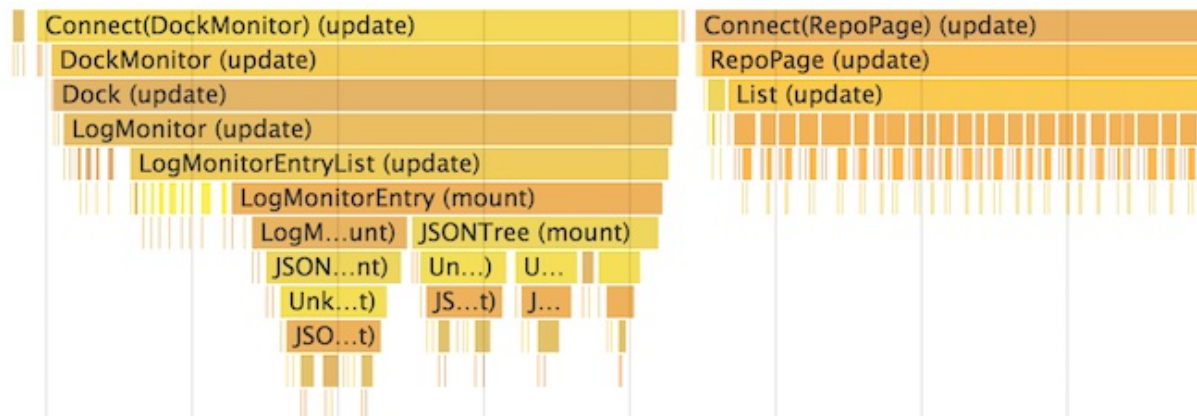
```
new webpack.DefinePlugin({
  'process.env.NODE_ENV': JSON.stringify('production')
}),
new webpack.optimize.UglifyJsPlugin()
```

You can learn more about this in [webpack documentation](#).

Remember that you only need to do this for production builds. You shouldn't apply `UglifyJsPlugin` or `DefinePlugin` with `'production'` value in development because they will hide useful React warnings, and make the builds much slower.

## Profiling Components with the Chrome Performance Tab

In the **development** mode, you can visualize how components mount, update, and unmount, using the performance tools in supported browsers. For example:



To do this in Chrome:

1. Temporarily **disable all Chrome extensions, especially React DevTools**. They can significantly skew the results!
2. Make sure you're running the application in the development mode.
3. Open the Chrome DevTools **Performance** tab and press **Record**.
4. Perform the actions you want to profile. Don't record more than 20 seconds or Chrome might hang.
5. Stop recording.
6. React events will be grouped under the **User Timing** label.

For a more detailed walkthrough, check out [this article by Ben Schwarz](#).

Note that **the numbers are relative so components will render faster in production**. Still, this should help you realize when unrelated UI gets updated by mistake, and how deep and how often your UI updates occur.

Currently Chrome, Edge, and IE are the only browsers supporting this feature, but we use the standard [User Timing API](#) so we expect more browsers to add support for it.

## Virtualize Long Lists

If your application renders long lists of data (hundreds or thousands of rows), we recommended 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

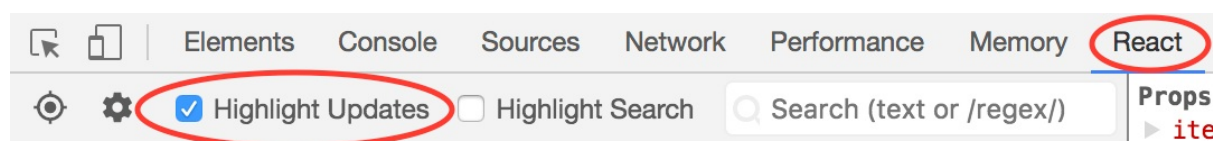
React builds and maintains an internal representation of the rendered UI. It includes the React elements you return from your components. This representation lets React avoid creating DOM nodes and accessing existing ones beyond necessity, as that can be slower than operations on JavaScript objects. 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.

You can now visualize these re-renders of the virtual DOM with React DevTools:

- [Chrome Browser Extension](#)
- [Firefox Browser Extension](#)
- [Standalone Node Package](#)

In the developer console select the **Highlight Updates** option in the **React** tab:



Interact with your page and you should see colored borders momentarily appear around any components that have re-rendered. This lets you spot re-renders that were not necessary. You can learn more about this React DevTools feature from this [blog post](#) from [Ben Edelstein](#).

Consider this example:



Note that when we're entering a second todo, the first todo also flashes on the screen on every keystroke. This means it is being re-rendered by React together with the input. This is sometimes called a "wasted" render. We know it is unnecessary because the first todo content has not changed, but React doesn't know this.

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) {
```

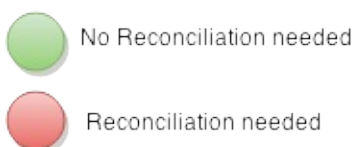
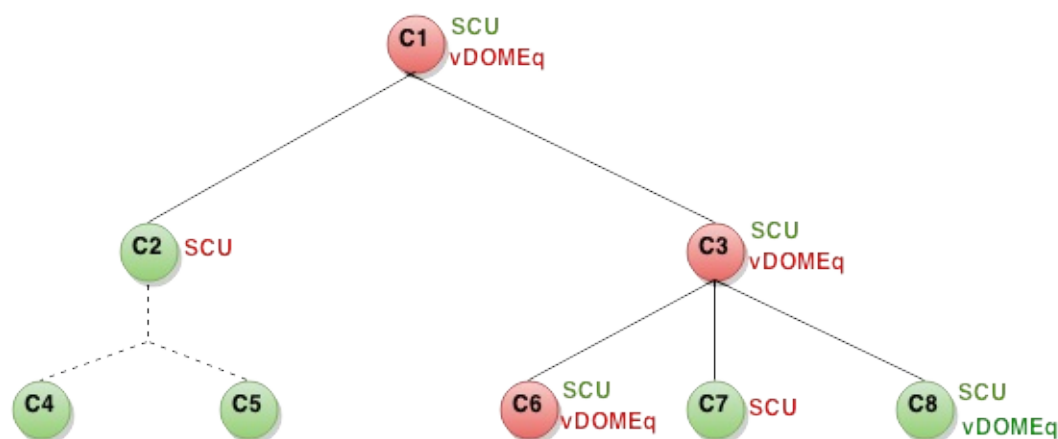
```
return true;
}
```

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.



**SCU** `shouldComponentUpdate?`  
**SCU**  
**vDOMEq** `are virtual DOMs equivalent?`  
**vDOMEq**

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(prevState => ({
    words: prevState.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(prevState => ({
    words: [...prevState.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.

There is a JavaScript proposal to add [object spread properties](#) to make it easier to update objects without mutation as well:

```
function updateColorMap(colormap) {
  return {...colormap, right: 'blue'};
}
```

If you're using Create React App, both `Object.assign` and the object spread syntax are available by default.

## Using Immutable Data Structures

[Immutable.js](#) is another way to solve this problem. It provides immutable, persistent collections that work via structural sharing:

- *Immutable*: once created, a collection cannot be altered at another point in time.
- *Persistent*: new collections can be created from a previous collection and a mutation such as `set`. The original collection is still valid after the new collection is created.
- *Structural Sharing*: new collections are created using as much of the same structure as the original collection as possible,

reducing copying to a minimum to improve performance.

Immutability makes tracking changes cheap. A change will always result in a new object so we only need to check if the reference to the object has changed. For example, in this regular JavaScript code:

```
const x = { foo: 'bar' };
const y = x;
y.foo = 'baz';
x === y; // true
```

Although `y` was edited, since it's a reference to the same object as `x`, this comparison returns `true`. You can write similar code with `immutable.js`:

```
const SomeRecord = Immutable.Record({ foo: null });
const x = new SomeRecord({ foo: 'bar' });
const y = x.set('foo', 'baz');
const z = x.set('foo', 'bar');
x === y; // false
x === z; // true
```

In this case, since a new reference is returned when mutating `x`, we can use a reference equality check (`x === y`) to verify that the new value stored in `y` is different than the original value stored in `x`.

Two other libraries that can help use immutable data are [seamless-immutable](#) and [immutability-helper](#).

Immutable data structures provide you with a cheap way to track changes on objects, which is all we need to implement `shouldComponentUpdate`. This can often provide you with a nice performance boost.

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>
  );
}
```

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](#).

[Try it on CodePen](#)

## 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(prevState => ({
      clicks: prevState.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>
  );
}

ReactDOM.render(<Parent />, appRoot);

```

### [Try it on CodePen](#)

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.

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 enable the **experimental** [Class Properties](#) syntax proposal with Babel:

```
class SayHello extends React.Component {
  constructor(props) {
    super(props);
    this.state = {message: 'Hello!'};
  }
  // WARNING: this syntax is experimental!
  // Using an arrow here binds the method:
  handleClick = () => {
    alert(this.state.message);
  }

  render() {
    return (
      <button onClick={this.handleClick}>
        Say hello
      </button>
    );
  }
}
```

Please note that the syntax above is **experimental** and the syntax may change, or the proposal might not make it into the language.

If you'd rather play it safe, you have a few options:

- Bind methods in the constructor.
- Use arrow functions, e.g. `onClick={(e) => this.handleClick(e)}` .
- Keep using `createClass` .

## 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>
    );
  }
});

ReactDOM.render(
  <TickTock />,
  document.getElementById('example')
```

```
);
```

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.

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>;
  }
}

ReactDOM.render(
  <Hello toWhat="World" />,
  document.getElementById('root')
);
```

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}`);
  }
}

ReactDOM.render(
  React.createElement(Hello, {toWhat: 'World'}, null),
  document.getElementById('root')
);
```

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, or as a subclass of `React.Component`, or a plain function for stateless components.

If you get tired of typing `React.createElement` so much, one common pattern is to assign a shorthand:

```
const e = React.createElement;

ReactDOM.render(
  e('div', null, 'Hello World'),
  document.getElementById('root')
);
```

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.

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 `<img>`, 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 `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.

### 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 `componentWillReceiveProps()` and `componentWillUpdate()` on the underlying instance.

Next, the `render()` method is called and the diff algorithm recurses on the previous result and the new result.

## 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 `<li>first</li>` trees, match the two `<li>second</li>` trees, and then insert the `<li>third</li>` 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 `<li>Duke</li>` and `<li>Villanova</li>` 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.

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 functional 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 hooks.

## 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 Functional Components

**You may not use the `ref` attribute on functional components** because they don't have instances:

```
function MyFunctionalComponent() {
  return <input />;
}

class Parent extends React.Component {
  constructor(props) {
    super(props);
    this.textInput = React.createRef();
  }
  render() {
    // This will *not* work!
    return (
      <MyFunctionalComponent ref={this.textInput} />
    );
  }
}
```

You should convert the component to a class if you need a ref to it, just like you do when you need lifecycle methods or state.

You can, however, **use the `ref` attribute inside a functional 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  
  let textInput = React.createRef();  
  
  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 functional 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.

## 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.

The term "[render prop](#)" refers to a simple 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](#) and [Downshift](#).

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: '100%' }} 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: '100%' }} 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 (
      <div>
        <h1>Move the mouse around!</h1>
        <Mouse />
      </div>
    );
  }
}
```

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: '100%' }} 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>
    );
  }
}

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.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: '100%' }} onMouseMove={this.handleMouseMove}>

        {/*
          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() {
```

```
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.

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": [
    "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.

### 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 run:

```
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 'querysting' {  
  export function stringify(val: object): string  
  export function parse(val: string): object  
}
```

## Using TypeScript with Create React App

`react-scripts-ts` automatically configures a `create-react-app` project to support TypeScript. You can use it like this:

```
create-react-app my-app --scripts-version=react-scripts-ts
```

Note that it is a **third party** project, and is not a part of Create React App.

You can also try [typescript-react-starter](#).

You are now ready to code! We recommend to check out the following resources to learn more about Typescript:

- [TypeScript Documentation: Basic Types](#)
- [TypeScript Documentation: Migrating from Javascript](#)
- [TypeScript Documentation: React and Webpack](#)

## Reason

[Reason](#) is not a new language; it's a new syntax and toolchain powered by the battle-tested language, [OCaml](#). Reason gives OCaml a familiar syntax geared toward JavaScript programmers, and caters to the existing NPM/Yarn workflow folks already know.

Reason is developed at Facebook, and is used in some of its products like Messenger. It is still somewhat experimental but it has [dedicated React bindings](#) maintained by Facebook and a [vibrant community](#).

## 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!

`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: `embed:strict-mode/enabling-strict-mode.js`

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](#)
- [Detecting unexpected side effects](#)
- [Detecting legacy context API](#)

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 async 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: `embed:16-3-release-blog-post/create-ref-example.js`

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.](#)

## Detecting unexpected side effects

## 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 async 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`
- `componentWillReceiveProps`
- `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 methods:

- Class component `constructor` method
- The `render` method
- `setState` updater functions (the first argument)
- The static `getDerivedStateFromProps` lifecycle

Note:

This only applies to development mode. *Lifecycles will not be double-invoked in production mode.*

For example, consider the following code: `embed:strict-mode/side-effects-in-constructor.js`

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.

## 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.

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
};
```

`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,
```

```
// 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
}),

// 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 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":
ReactDOM.render(
  <Greeting />,
  document.getElementById('example')
);
```

If you are using a Babel transform like [transform-class-properties](#), you can also declare `defaultProps` as static property within a React component class. This syntax has not yet been finalized though and will require a compilation step to work within a browser. For more information, see the [class fields proposal](#).

```
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`.

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>
    );
  }
}
```

[Try it on CodePen](#)

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`.

```
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:

```
embed:uncontrolled-components/input-type-file.js
```

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);
    ReactDOM.render(<a href={url}>{name}</a>, mountPoint);
  }
}
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.

`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.

### 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`

---

## 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".

---

## 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,  
  [props],  
  [...children]  
)
```

Clone and return a new React element using `element` as the starting point. 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.

`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.

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 a keyed fragment or array it will be traversed: the function will never be passed the container objects. If `children` is `null` or `undefined` , returns `null` or `undefined` rather than an array.

### `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. `embed:16-3-release-blog-post/create-ref-example.js`

## 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.

`embed:reference-react-forward-ref.js`

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](#).

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:

These methods are considered legacy and you should [avoid them](#) 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

This method is called when there is an error during rendering, in a lifecycle method, or in the constructor of any child component.

- `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`.** 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 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:

```
embed:react-component-reference/get-snapshot-before-update.js
```

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`).



## componentDidCatch()

```
componentDidCatch(error, info)
```

[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 this lifecycle method. Calling `setState()` in it 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.

---

## 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()
```

`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 hook called on server rendering.

### 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\_componentWillReceiveProps()

```
UNSAFE_componentWillReceiveProps(nextProps)
```

### Note:

Using this lifecycle method often leads to bugs and inconsistencies, and for that reason it is going to be deprecated in the future.

---

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.

For other use cases, [follow the recommendations in this blog post about derived state](#).

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.

In very rare cases, you might want to use the `getDerivedStateFromProps` lifecycle as a last resort.

`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()`.

#### 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.

---

## UNSAFE\_componentWillUpdate()

```
UNSAFE_componentWillUpdate(nextProps, nextState)
```

`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

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.

#### 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. React does not guarantee that the state changes are applied immediately.

`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:

```
(prevState, props) => stateChange
```

`prevState` is a reference to the previous state. It should not be directly mutated. Instead, changes should be represented by building a new object based on the input from `prevState` and `props`. For instance, suppose we wanted to increment a value in state by `props.step`:

```
this.setState((prevState, props) => {  
  return {counter: prevState.counter + props.step};  
});
```

Both `prevState` and `props` received by the updater function are guaranteed to be up-to-date. The output of the updater is shallowly merged with `prevState`.

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 previous state, we recommend using the updater function form, instead:

```
this.setState((prevState) => {
  return {quantity: prevState.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.

If you load React from a `<script>` tag, these top-level APIs are available on the `ReactDOM` global. If you use ES6 with npm, you can write `import ReactDOM from 'react-dom'`. If you use ES5 with npm, you can write `var ReactDOM = require('react-dom')`.

## Overview

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 of the React model if you need to. Most of your components should not need to use this module.

- `render()`
- `hydrate()`
- `unmountComponentAtNode()`
- `findDOMNode()`
- `createPortal()`

## Browser Support

React supports all popular browsers, including Internet Explorer 9 and above, although [some polyfills are required](#) for older browsers such as IE 9 and IE 10.

### Note

We don't support older browsers that don't support ES5 methods, but 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. You're on your own if you choose to take this path.

---

## Reference

### `render()`

```
ReactDOM.render(element, container[, callback])
```

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:

`ReactDOM.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.

`ReactDOM.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.

`ReactDOM.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 `ReactDOM.render()` to hydrate a server-rendered container is deprecated and will be removed in React 17. Use `hydrate()` instead.

## hydrate()

```
ReactDOM.hydrate(element, container[, callback])
```

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()

```
ReactDOM.unmountComponentAtNode(container)
```

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()

```
ReactDOM.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` 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.

`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 functional components.

---

## `createPortal()`

```
ReactDOM.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](#).



The `ReactDOMServer` object enables you to render components to static markup. Typically, it's used on a Node server:

```
// ES modules
import ReactDOMServer from 'react-dom/server';
// CommonJS
var ReactDOMServer = require('react-dom/server');
```

## Overview

The following methods can be used in both the server and browser environments:

- `renderToString()`
- `renderToStaticMarkup()`

These additional methods depend on a package ( `stream` ) that is **only available on the server**, and won't work in the browser.

- `renderToNodeStream()`
- `renderToStaticNodeStream()`

---

## Reference

### `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.hydrate()` 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.

---

### `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.hydrate()` on the client.

---

### `renderToNodeStream()`

```
ReactDOMServer.renderToNodeStream(element)
```

Render a React element to its initial HTML. Returns a [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.hydrate()` 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.hydrate()` 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.

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 &middot; 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

The `selected` attribute is supported by `<option>` components. You can use it to set whether the component is selected. This is useful for building controlled components.

## 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.backgroundImage`). 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.hydrate\(\) documentation](#).

## value

The `value` attribute is supported by `<input>` 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.

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. 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()
DOMEventTarget target
number timeStamp
string type
```

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.

## Event Pooling

The `SyntheticEvent` is pooled. This means that the `SyntheticEvent` object will be reused and all properties will be nullified after the event callback has been invoked. This is for performance reasons. As such, you cannot access the event in an asynchronous way.

```
function onClick(event) {
  console.log(event); // => nullified object.
  console.log(event.type); // => "click"
  const eventType = event.type; // => "click"

  setTimeout(function() {
    console.log(event.type); // => null
    console.log(eventType); // => "click"
  }, 0);

  // Won't work. this.state.clickEvent will only contain null values.
  this.setState({clickEvent: event});
}
```

```
// You can still export event properties.  
this.setState({eventType: event.type});  
}
```

Note:

If you want to access the event properties in an asynchronous way, you should call `event.persist()` on the event, which will remove the synthetic event from the pool and allow references to the event to be retained by user code.

## 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](#)
- [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
```

---

## Form Events

---

Event names:

```
onChange onInput onInvalid onSubmit
```

For more information about the `onChange` event, see [Forms](#).

---

## 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 tiltX
number tiltY
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
```

---

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
```

## 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:

Airbnb has released a testing utility called Enzyme, which makes it easy to assert, manipulate, and traverse your React Components' output. If you're deciding on a unit testing utility to use together with Jest, or any other test runner, it's worth checking out: <http://airbnb.io/enzyme/>

Alternatively, there is another testing utility called react-testing-library designed to enable and encourage writing tests that use your components as the end users use them. It also works with any test runner: <https://git.io/react-testing-library>

- `simulate`
- `renderIntoDocument()`
- `mockComponent()`
- `isElement()`
- `isElementOfType()`
- `isDOMComponent()`
- `isCompositeComponent()`
- `isCompositeComponentWithType()`
- `findAllInRenderedTree()`
- `scryRenderedDOMComponentsWithClass()`
- `findRenderedDOMComponentWithClass()`
- `scryRenderedDOMComponentsWithTag()`
- `findRenderedDOMComponentWithTag()`
- `scryRenderedComponentsWithType()`
- `findRenderedComponentWithType()`

## Reference

### Shallow Rendering

When writing unit tests for React, shallow rendering can be helpful. Shallow rendering lets you render a component "one level deep" and assert facts about what its render method returns, without worrying about the behavior of child components, which are not instantiated or rendered. This does not require a DOM.

Note:

The shallow renderer has moved to `react-test-renderer/shallow`.  
[Learn more about shallow rendering on its reference page.](#)

### 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.

---

## renderIntoDocument()

```
renderIntoDocument(element)
```

Render a React element into a detached DOM node in the document. **This function requires a DOM.**

#### 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.

---

## 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 [shallow rendering](#) or `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 `<span>`).

---

## **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.

---

## Importing

```
import ShallowRenderer from 'react-test-renderer/shallow'; // ES6
var ShallowRenderer = require('react-test-renderer/shallow'); // ES5 with npm
```

## Overview

When writing unit tests for React, shallow rendering can be helpful. Shallow rendering lets you render a component "one level deep" and assert facts about what its render method returns, without worrying about the behavior of child components, which are not instantiated or rendered. This does not require a DOM.

For example, if you have the following component:

```
function MyComponent() {
  return (
    <div>
      <span className="heading">Title</span>
      <Subcomponent foo="bar" />
    </div>
  );
}
```

Then you can assert:

```
import ShallowRenderer from 'react-test-renderer/shallow';

// in your test:
const renderer = new ShallowRenderer();
renderer.render(<MyComponent />);
const result = renderer.getRenderOutput();

expect(result.type).toBe('div');
expect(result.props.children).toEqual([
  <span className="heading">Title</span>,
  <Subcomponent foo="bar" />
]);
```

Shallow testing currently has some limitations, namely not supporting refs.

Note:

We also recommend checking out Enzyme's [Shallow Rendering API](#). It provides a nicer higher-level API over the same functionality.

## Reference

### `shallowRenderer.render()`

You can think of the `shallowRenderer` as a "place" to render the component you're testing, and from which you can extract the component's output.

`shallowRenderer.render()` is similar to `ReactDOM.render()` but it doesn't require DOM and only renders a single level deep. This means you can test components isolated from how their children are implemented.

### **`shallowRenderer.getRenderOutput()`**

After `shallowRenderer.render()` has been called, you can use `shallowRenderer.getRenderOutput()` to get the shallowly rendered output.

You can then begin to assert facts about the output.

## 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 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. The returned instance has the following methods and properties.

### **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. Unlike `toJSON()`, 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 functional 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 functional 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);
```

React 16 depends on the collection types [Map](#) and [Set](#). If you support older browsers and devices which may not yet provide these natively (e.g. IE < 11) or which have non-compliant implementations (e.g. IE 11), consider including a global polyfill in your bundled application, such as [core-js](#) or [babel-polyfill](#).

A polyfilled environment for React 16 using core-js to support older browsers might look like:

```
import 'core-js/es6/map';
import 'core-js/es6/set';

import React from 'react';
import ReactDOM from 'react-dom';

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

React also depends on `requestAnimationFrame` (even in test environments).

You can use the [raf](#) package to shim `requestAnimationFrame` :

```
import 'raf/polyfill';
```

## 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:

```
const name = 'Clementine';
ReactDOM.render(
  <h1 className="hello">My name is {name}!</h1>,
  document.getElementById('root')
);
```

## 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`. To do so, it must call `this.setState()`. Only components defined as classes can have 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".

## How can I make an AJAX call?

You can use any AJAX library you like with React. Some popular ones are [Axios](#), [jQuery AJAX](#), and the browser built-in [window.fetch](#).

## Where in the component lifecycle should I make an AJAX call?

You should populate data with AJAX calls in the `componentDidMount` lifecycle method. This is so you can use `setState` to update your component when the data is retrieved.

## Example: Using AJAX results to set local state

The component below demonstrates how to make an AJAX call in `componentDidMount` to populate local component state.

The example API returns a JSON object like this:

```
{
  items: [
    { id: 1, name: 'Apples', price: '$2' },
    { id: 2, name: 'Peaches', price: '$5' }
  ]
}
```

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      error: null,
      isLoading: false,
      items: []
    };
  }

  componentDidMount() {
    fetch("https://api.example.com/items")
      .then(res => res.json())
      .then(
        (result) => {
          this.setState({
            isLoading: true,
            items: result.items
          });
        },
        // Note: it's important to handle errors here
        // instead of a catch() block so that we don't swallow
        // exceptions from actual bugs in components.
        (error) => {
          this.setState({
            isLoading: true,
            error
          });
        }
      );
  }
}
```



```
        });
    }
)
}

render() {
    const { error, isLoading, items } = this.state;
    if (error) {
        return <div>Error: {error.message}</div>;
    } else if (!isLoading) {
        return <div>Loading...</div>;
    } else {
        return (
            <ul>
                {items.map(item => (
                    <li key={item.name}>
                        {item.name} {item.price}
                    </li>
                ))}
            </ul>
        );
    }
}
```

## Do I need to use JSX with React?

No! Check out ["React Without JSX"](#) to learn more.

## Do I need to use ES6 (+) with React?

No! Check out ["React Without ES6"](#) to learn more.

## How can I write comments in JSX?

```
<div>
  {/* Comment goes here */}
  Hello, {name}!
</div>
```

```
<div>
  {/* It also works
   for multi-line comments. */}
  Hello, {name}!
</div>
```

## How do I pass an event handler (like `onClick`) to a component?

Pass event handlers and other functions as props to child components:

```
<button onClick={this.handleClick}>
```

If you need to have access to the parent component in the handler, you also need to bind the function to the component instance (see below).

## How do I bind a function to a component instance?

There are several ways to make sure functions have access to component attributes like `this.props` and `this.state`, depending on which syntax and build steps you are using.

### Bind in Constructor (ES2015)

```
class Foo extends Component {
  constructor(props) {
    super(props);
    this.handleClick = this.handleClick.bind(this);
  }
  handleClick() {
    console.log('Click happened');
  }
  render() {
    return <button onClick={this.handleClick}>Click Me</button>;
  }
}
```

### Class Properties (Stage 3 Proposal)

```
class Foo extends Component {
  // Note: this syntax is experimental and not standardized yet.
  handleClick = () => {
    console.log('Click happened');
  }
  render() {
    return <button onClick={this.handleClick}>Click Me</button>;
  }
}
```

### Bind in Render

```
class Foo extends Component {
  handleClick() {
    console.log('Click happened');
  }
}
```

```
render() {  
  return <button onClick={this.handleClick.bind(this)}>Click Me</button>;  
}
```

**Note:**

Using `Function.prototype.bind` in render creates a new function each time the component renders, which may have performance implications (see below).

## Arrow Function in Render

```
class Foo extends Component {  
  handleClick() {  
    console.log('Click happened');  
  }  
  render() {  
    return <button onClick={() => this.handleClick()}>Click Me</button>;  
  }  
}
```

**Note:**

Using an arrow function in render creates a new function each time the component renders, which may have performance implications (see below).

## Is it OK to use arrow functions in render methods?

Generally speaking, yes, it is OK, and it is often the easiest way to pass parameters to callback functions.

If you do have performance issues, by all means, optimize!

## Why is binding necessary at all?

In JavaScript, these two code snippets are **not** equivalent:

```
obj.method();
```

```
var method = obj.method;  
method();
```

Binding methods helps ensure that the second snippet works the same way as the first one.

With React, typically you only need to bind the methods you *pass* to other components. For example, `<button onClick={this.handleClick}>` passes `this.handleClick` so you want to bind it. However, it is unnecessary to bind the `render` method or the lifecycle methods: we don't pass them to other components.

[This post by Yehuda Katz](#) explains what binding is, and how functions work in JavaScript, in detail.

## Why is my function being called every time the component renders?

Make sure you aren't *calling the function* when you pass it to the component:

```
render() {
  // Wrong: handleClick is called instead of passed as a reference!
  return <button onClick={this.handleClick()}>Click Me</button>
}
```

Instead, *pass the function itself* (without parens):

```
render() {
  // Correct: handleClick is passed as a reference!
  return <button onClick={this.handleClick}>Click Me</button>
}
```

## How do I pass a parameter to an event handler or callback?

You can use an arrow function to wrap around an event handler and pass parameters:

```
<button onClick={() => this.handleClick(id)} />
```

This is equivalent to calling `.bind` :

```
<button onClick={this.handleClick.bind(this, id)} />
```

## Example: Passing params using arrow functions

```
const A = 65 // ASCII character code

class Alphabet extends React.Component {
  constructor(props) {
    super(props);
    this.handleClick = this.handleClick.bind(this);
    this.state = {
      justClicked: null,
      letters: Array.from({length: 26}, (_, i) => String.fromCharCode(A + i))
    };
  }
  handleClick(letter) {
    this.setState({ justClicked: letter });
  }
  render() {
    return (
      <div>
        Just clicked: {this.state.justClicked}
        <ul>
          {this.state.letters.map(letter =>
            <li key={letter} onClick={() => this.handleClick(letter)}>
              {letter}
            </li>
          )}
        </ul>
      </div>
    );
  }
}
```

```

    })
  </ul>
</div>
)
}
}

```

## Example: Passing params using data-attributes

Alternately, you can use DOM APIs to store data needed for event handlers. Consider this approach if you need to optimize a large number of elements or have a render tree that relies on `React.PureComponent` equality checks.

```

const A = 65 // ASCII character code

class Alphabet extends React.Component {
  constructor(props) {
    super(props);
    this.handleClick = this.handleClick.bind(this);
    this.state = {
      justClicked: null,
      letters: Array.from({length: 26}, (_, i) => String.fromCharCode(A + i))
    };
  }

  handleClick(e) {
    this.setState({
      justClicked: e.target.dataset.letter
    });
  }

  render() {
    return (
      <div>
        Just clicked: {this.state.justClicked}
        <ul>
          {this.state.letters.map(letter =>
            <li key={letter} data-letter={letter} onClick={this.handleClick}>
              {letter}
            </li>
          )}
        </ul>
      </div>
    )
  }
}

```

**How can I prevent a function from being called too quickly or too many times in a row?**

If you have an event handler such as `onClick` or `onScroll` and want to prevent the callback from being fired too quickly, then you can limit the rate at which callback is executed. This can be done by using:

- **throttling**: sample changes based on a time based frequency (eg `_.throttle` )
- **debouncing**: publish changes after a period of inactivity (eg `_.debounce` )
- `requestAnimationFrame` **throttling**: sample changes based on `requestAnimationFrame` (eg `raf-schd` )

See [this visualization](#) for a comparison of `throttle` and `debounce` functions.

Note:

`_.debounce` , `_.throttle` and `raf-schd` provide a `cancel` method to cancel delayed callbacks. You should either call this method from `componentWillUnmount` or check to ensure that the component is still mounted within the delayed function.

## Throttle

Throttling prevents a function from being called more than once in a given window of time. The example below throttles a "click" handler to prevent calling it more than once per second.

```
import throttle from 'lodash.throttle';

class LoadMoreButton extends React.Component {
  constructor(props) {
    super(props);
    this.handleClick = this.handleClick.bind(this);
    this.handleClickThrottled = throttle(this.handleClick, 1000);
  }

  componentWillUnmount() {
    this.handleClickThrottled.cancel();
  }

  render() {
    return <button onClick={this.handleClickThrottled}>Load More</button>;
  }

  handleClick() {
    this.props.loadMore();
  }
}
```

## Debounce

Debouncing ensures that a function will not be executed until after a certain amount of time has passed since it was last called. This can be useful when you have to perform some expensive calculation in response to an event that might dispatch rapidly (eg scroll or keyboard events). The example below debounces text input with a 250ms delay.

```
import debounce from 'lodash.debounce';

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

```

    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.emitChangeDebounce = debounce(this.emitChange, 250);
  }

  componentWillUnmount() {
    this.emitChangeDebounce.cancel();
  }

  render() {
    return (
      <input
        type="text"
        onChange={this.handleChange}
        placeholder="Search..."
        defaultValue={this.props.value}
      />
    );
  }

  handleChange(e) {
    // React pools events, so we read the value before debounce.
    // Alternately we could call `event.persist()` and pass the entire event.
    // For more info see reactjs.org/docs/events.html#event-pooling
    this.emitChangeDebounce(e.target.value);
  }

  emitChange(value) {
    this.props.onChange(value);
  }
}

```

## requestAnimationFrame throttling

`requestAnimationFrame` is a way of queuing a function to be executed in the browser at the optimal time for rendering performance. A function that is queued with `requestAnimationFrame` will fire in the next frame. The browser will work hard to ensure that there are 60 frames per second (60 fps). However, if the browser is unable to it will naturally *limit* the amount of frames in a second. For example, a device might only be able to handle 30 fps and so you will only get 30 frames in that second. Using `requestAnimationFrame` for throttling is a useful technique in that it prevents you from doing more than 60 updates in a second. If you are doing 100 updates in a second this creates additional work for the browser that the user will not see anyway.

### Note:

Using this technique will only capture the last published value in a frame. You can see an example of how this optimization works on [MDN](#)

```

import rafSchedule from 'raf-schd';

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

```



```
    this.handleScroll = this.handleScroll.bind(this);

    // Create a new function to schedule updates.
    this.scheduleUpdate = rafSchedule(
      point => this.props.onScroll(point)
    );
  }

  handleScroll(e) {
    // When we receive a scroll event, schedule an update.
    // If we receive many updates within a frame, we'll only publish the latest
    value.
    this.scheduleUpdate({ x: e.clientX, y: e.clientY });
  }

  componentWillUnmount() {
    // Cancel any pending updates since we're unmounting.
    this.scheduleUpdate.cancel();
  }

  render() {
    return (
      <div
        style={{ overflow: 'scroll' }}
        onScroll={this.handleScroll}
      >
        
      </div>
    );
  }
}
```

## Testing your rate limiting

When testing your rate limiting code works correctly it is helpful to have the ability to fast forward time. If you are using `jest` then you can use `mock timers` to fast forward time. If you are using `requestAnimationFrame` throttling then you may find `raf-stub` to be a useful tool to control the ticking of animation frames.

## What does `setState` do?

`setState()` schedules an update to a component's `state` object. When state changes, the component responds by re-rendering.

## What is the difference between `state` and `props` ?

`props` (short for "properties") and `state` are both plain JavaScript objects. While both hold information that influences the output of render, they are different in one important way: `props` get passed *to* the component (similar to function parameters) whereas `state` is managed *within* the component (similar to variables declared within a function).

Here are some good resources for further reading on when to use `props` vs `state` :

- [Props vs State](#)
- [ReactJS: Props vs. State](#)

## Why is `setState` giving me the wrong value?

In React, both `this.props` and `this.state` represent the *rendered* values, i.e. what's currently on the screen.

Calls to `setState` are asynchronous - don't rely on `this.state` to reflect the new value immediately after calling `setState` . Pass an updater function instead of an object if you need to compute values based on the current state (see below for details).

Example of code that will *not* behave as expected:

```
incrementCount() {
  // Note: this will *not* work as intended.
  this.setState({count: this.state.count + 1});
}

handleSomething() {
  // Let's say `this.state.count` starts at 0.
  this.incrementCount();
  this.incrementCount();
  this.incrementCount();
  // When React re-renders the component, `this.state.count` will be 1, but you
  // expected 3.

  // This is because `incrementCount()` function above reads from
  // `this.state.count`,
  // but React doesn't update `this.state.count` until the component is re-
  // rendered.
  // So `incrementCount()` ends up reading `this.state.count` as 0 every time, and
  // sets it to 1.

  // The fix is described below!
}
```

See below for how to fix this problem.

## How do I update state with values that depend on the current state?

Pass a function instead of an object to `setState` to ensure the call always uses the most updated version of state (see below).

## What is the difference between passing an object or a function in `setState` ?

Passing an update function allows you to access the current state value inside the updater. Since `setState` calls are batched, this lets you chain updates and ensure they build on top of each other instead of conflicting:

```
incrementCount() {  
  this.setState((prevState) => {  
    // Important: read `prevState` instead of `this.state` when updating.  
    return {count: prevState.count + 1}  
  });  
}  
  
handleSomething() {  
  // Let's say `this.state.count` starts at 0.  
  this.incrementCount();  
  this.incrementCount();  
  this.incrementCount();  
  
  // If you read `this.state.count` now, it would still be 0.  
  // But when React re-renders the component, it will be 3.  
}
```

[Learn more about setState](#)

## When is `setState` asynchronous?

Currently, `setState` is asynchronous inside event handlers.

This ensures, for example, that if both `Parent` and `Child` call `setState` during a click event, `Child` isn't re-rendered twice. Instead, React "flushes" the state updates at the end of the browser event. This results in significant performance improvements in larger apps.

This is an implementation detail so avoid relying on it directly. In the future versions, React will batch updates by default in more cases.

## Why doesn't React update `this.state` synchronously?

As explained in the previous section, React intentionally "waits" until all components call `setState()` in their event handlers before starting to re-render. This boosts performance by avoiding unnecessary re-renders.

However, you might still be wondering why React doesn't just update `this.state` immediately without re-rendering.

There are two main reasons:

- This would break the consistency between `props` and `state`, causing issues that are very hard to debug.
- This would make some of the new features we're working on impossible to implement.

This [GitHub comment](#) dives deep into the specific examples.

## Should I use a state management library like Redux or MobX?

[Maybe.](#)

It's a good idea to get to know React first, before adding in additional libraries. You can build quite complex applications using only React.

## How do I add CSS classes to components?

Pass a string as the `className` prop:

```
render() {  
  return <span className="menu navigation-menu">Menu</span>  
}
```

It is common for CSS classes to depend on the component props or state:

```
render() {  
  let className = 'menu';  
  if (this.props.isActive) {  
    className += ' menu-active';  
  }  
  return <span className={className}>Menu</span>  
}
```

If you often find yourself writing code like this, [classnames](#) package can simplify it.

## Can I use inline styles?

Yes, see the docs on styling [here](#).

## Are inline styles bad?

CSS classes are generally better for performance than inline styles.

## What is CSS-in-JS?

"CSS-in-JS" refers to a pattern where CSS is composed using JavaScript instead of defined in external files. Read a comparison of CSS-in-JS libraries [here](#).

*Note that this functionality is not a part of React, but provided by third-party libraries.* React does not have an opinion about how styles are defined; if in doubt, a good starting point is to define your styles in a separate `*.css` file as usual and refer to them using `className`.

## Can I do animations in React?

React can be used to power animations. See [React Transition Group](#) and [React Motion](#), for example.

## Is there a recommended way to structure React projects?

React doesn't have opinions on how you put files into folders. That said there are a few common approaches popular in the ecosystem you may want to consider.

### Grouping by features or routes

One common way to structure projects is locate CSS, JS, and tests together inside folders grouped by feature or route.

```
common/  
  Avatar.js  
  Avatar.css  
  APIUtils.js  
  APIUtils.test.js  
feed/  
  index.js  
  Feed.js  
  Feed.css  
  FeedStory.js  
  FeedStory.test.js  
  FeedAPI.js  
profile/  
  index.js  
  Profile.js  
  ProfileHeader.js  
  ProfileHeader.css  
  ProfileAPI.js
```

The definition of a "feature" is not universal, and it is up to you to choose the granularity. If you can't come up with a list of top-level folders, you can ask the users of your product what major parts it consists of, and use their mental model as a blueprint.

### Grouping by file type

Another popular way to structure projects is to group similar files together, for example:

```
api/  
  APIUtils.js  
  APIUtils.test.js  
  ProfileAPI.js  
  UserAPI.js  
components/  
  Avatar.js  
  Avatar.css  
  Feed.js  
  Feed.css  
  FeedStory.js  
  FeedStory.test.js  
  Profile.js  
  ProfileHeader.js  
  ProfileHeader.css
```

---

Some people also prefer to go further, and separate components into different folders depending on their role in the application. For example, [Atomic Design](#) is a design methodology built on this principle. Remember that it's often more productive to treat such methodologies as helpful examples rather than strict rules to follow.

## Avoid too much nesting

There are many pain points associated with deep directory nesting in JavaScript projects. It becomes harder to write relative imports between them, or to update those imports when the files are moved. Unless you have a very compelling reason to use a deep folder structure, consider limiting yourself to a maximum of three or four nested folders within a single project. Of course, this is only a recommendation, and it may not be relevant to your project.

## Don't overthink it

If you're just starting a project, [don't spend more than five minutes](#) on choosing a file structure. Pick any of the above approaches (or come up with your own) and start writing code! You'll likely want to rethink it anyway after you've written some real code.

If you feel completely stuck, start by keeping all files in a single folder. Eventually it will grow large enough that you will want to separate some files from the rest. By that time you'll have enough knowledge to tell which files you edit together most often. In general, it is a good idea to keep files that often change together close to each other. This principle is called "colocation".

As projects grow larger, they often use a mix of both of the above approaches in practice. So choosing the "right" one in the beginning isn't very important.

## What is the Virtual DOM?

The virtual DOM (VDOM) is a programming concept where an ideal, or "virtual", representation of a UI is kept in memory and synced with the "real" DOM by a library such as ReactDOM. This process is called [reconciliation](#).

This approach enables the declarative API of React: You tell React what state you want the UI to be in, and it makes sure the DOM matches that state. This abstracts out the attribute manipulation, event handling, and manual DOM updating that you would otherwise have to use to build your app.

Since "virtual DOM" is more of a pattern than a specific technology, people sometimes say it to mean different things. In React world, the term "virtual DOM" is usually associated with [React elements](#) since they are the objects representing the user interface. React, however, also uses internal objects called "fibers" to hold additional information about the component tree. They may also be considered a part of "virtual DOM" implementation in React.

## Is the Shadow DOM the same as the Virtual DOM?

No, they are different. The Shadow DOM is a browser technology designed primarily for scoping variables and CSS in web components. The virtual DOM is a concept implemented by libraries in JavaScript on top of browser APIs.

## What is "React Fiber"?

Fiber is the new reconciliation engine in React 16. Its main goal is to enable incremental rendering of the virtual DOM. [Read more](#).



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.

In a future major release of React, the code that implements PropTypes validation functions will be stripped in production. Once this happens, any code that calls these functions manually (that isn't stripped in production) will throw an error.

## Declaring PropTypes is still fine

The normal usage of PropTypes is still supported:

```
Button.propTypes = {
  highlighted: PropTypes.bool
};
```

Nothing changes here.

## Don't call PropTypes directly

Using PropTypes in any other way than annotating React components with them is no longer supported:

```
var apiShape = PropTypes.shape({
  body: PropTypes.object,
  statusCode: PropTypes.number.isRequired
}).isRequired;

// Not supported!
var error = apiShape(json, 'response');
```

If you depend on using PropTypes like this, we encourage you to use or create a fork of PropTypes (such as [these two](#) packages).

If you don't fix the warning, this code will crash in production with React 16.

## If you don't call PropTypes directly but still get the warning

Inspect the stack trace produced by the warning. You will find the component definition responsible for the PropTypes direct call. Most likely, the issue is due to third-party PropTypes that wrap React's PropTypes, for example:

```
Button.propTypes = {
  highlighted: ThirdPartyPropTypes.deprecated(
    PropTypes.bool,
    'Use `active` prop instead'
  )
}
```

In this case, `ThirdPartyPropTypes.deprecated` is a wrapper calling `PropTypes.bool`. This pattern by itself is fine, but triggers a false positive because React thinks you are calling PropTypes directly. The next section explains how to fix this problem for a library implementing something like `ThirdPartyPropTypes`. If it's not a library you wrote, you can file an issue against it.

## Fixing the false positive in third party PropTypes

If you are an author of a third party PropTypes library and you let consumers wrap existing React PropTypes, they might start seeing this warning coming from your library. This happens because React doesn't see a "secret" last argument that [it passes](#) to detect manual PropTypes calls.

Here is how to fix it. We will use `deprecated` from [react-bootstrap/react-prop-types](#) as an example. The current implementation only passes down the `props`, `propName`, and `componentName` arguments:

```
export default function deprecated(propType, explanation) {
  return function validate(props, propName, componentName) {
    if (props[propName] != null) {
      const message = `${propName} property of "${componentName}" has been
deprecated.\n${explanation}`;
      if (!warned[message]) {
        warning(false, message);
        warned[message] = true;
      }
    }

    return propType(props, propName, componentName);
  };
}
```

In order to fix the false positive, make sure you pass **all** arguments down to the wrapped PropTypes. This is easy to do with the ES6 `...rest` notation:

```
export default function deprecated(propType, explanation) {
  return function validate(props, propName, componentName, ...rest) { // Note
...rest here
    if (props[propName] != null) {
      const message = `${propName} property of "${componentName}" has been
deprecated.\n${explanation}`;
      if (!warned[message]) {
        warning(false, message);
        warned[message] = true;
      }
    }

    return propType(props, propName, componentName, ...rest); // and here
  };
}
```

This will silence the warning.

The `invalid-aria-prop` warning will fire if you attempt to render a DOM element with an `aria-*` prop that does not exist in the Web Accessibility Initiative (WAI) Accessible Rich Internet Application (ARIA) [specification](#).

1. If you feel that you are using a valid prop, check the spelling carefully. `aria-labelledby` and `aria-activedescendant` are often misspelled.
2. React does not yet recognize the attribute you specified. This will likely be fixed in a future version of React. However, React currently strips all unknown attributes, so specifying them in your React app will not cause them to be rendered

You probably came here because your code is calling your component as a plain function call. This is now deprecated:

```
var MyComponent = require('MyComponent');

function render() {
  return MyComponent({ foo: 'bar' }); // WARNING
}
```

## JSX

React components can no longer be called directly like this. Instead [you can use JSX](#).

```
var React = require('react');
var MyComponent = require('MyComponent');

function render() {
  return <MyComponent foo="bar" />;
}
```

## Without JSX

If you don't want to, or can't use JSX, then you'll need to wrap your component in a factory before calling it:

```
var React = require('react');
var MyComponent = React.createFactory(require('MyComponent'));

function render() {
  return MyComponent({ foo: 'bar' });
}
```

This is an easy upgrade path if you have a lot of existing function calls.

## Dynamic components without JSX

If you get a component class from a dynamic source, then it might be unnecessary to create a factory that you immediately invoke. Instead you can just create your element inline:

```
var React = require('react');

function render(MyComponent) {
  return React.createElement(MyComponent, { foo: 'bar' });
}
```

## In Depth

[Read more about WHY we're making this change.](#)



You are probably here because you got one of the following error messages:

*React 16.0.0+*

Warning:

Element ref was specified as a string (myRefName) but no owner was set. You may have multiple copies of React loaded. (details: <https://fb.me/react-refs-must-have-owner>).

*earlier versions of React*

Warning:

addComponentAsRefTo(...): Only a ReactOwner can have refs. You might be adding a ref to a component that was not created inside a component's `render` method, or you have multiple copies of React loaded.

This usually means one of three things:

- You are trying to add a `ref` to a functional component.
- You are trying to add a `ref` to an element that is being created outside of a component's `render()` function.
- You have multiple (conflicting) copies of React loaded (eg. due to a misconfigured npm dependency)

## Refs on Functional Components

If `<Foo>` is a functional component, you can't add a ref to it:

```
// Doesn't work if Foo is a function!
<Foo ref={foo} />
```

If you need to add a ref to a component, convert it to a class first, or consider not using refs as they are [rarely necessary](#).

## Strings Refs Outside the Render Method

This usually means that you're trying to add a ref to a component that doesn't have an owner (that is, was not created inside of another component's `render` method). For example, this won't work:

```
// Doesn't work!
ReactDOM.render(<App ref="app" />, el);
```

Try rendering this component inside of a new top-level component which will hold the ref. Alternatively, you may use a callback ref:

```
let app;
ReactDOM.render(
  <App ref={inst => {
    app = inst;
  }} />,
  el
);
```

Consider if you [really need a ref](#) before using this approach.

## Multiple copies of React

Bower does a good job of deduplicating dependencies, but npm does not. If you aren't doing anything (fancy) with refs, there is a good chance that the problem is not with your refs, but rather an issue with having multiple copies of React loaded into your project. Sometimes, when you pull in a third-party module via npm, you will get a duplicate copy of the dependency library, and this can create problems.

If you are using npm... `npm ls` or `npm ls react` might help illuminate.

Most props on a JSX element are passed on to the component, however, there are two special props ( `ref` and `key` ) which are used by React, and are thus not forwarded to the component.

For instance, attempting to access `this.props.key` from a component (i.e., the render function or `propTypes`) is not defined. If you need to access the same value within the child component, you should pass it as a different prop (ex: `<ListItemWrapper key={result.id} id={result.id} />` ). While this may seem redundant, it's important to separate app logic from reconciling hints.



The unknown-prop warning will fire if you attempt to render a DOM element with a prop that is not recognized by React as a legal DOM attribute/property. You should ensure that your DOM elements do not have spurious props floating around.

There are a couple of likely reasons this warning could be appearing:

1. Are you using `{...this.props}` or `cloneElement(element, this.props)` ? Your component is transferring its own props directly to a child element (eg. [transferring props](#)). When transferring props to a child component, you should ensure that you are not accidentally forwarding props that were intended to be interpreted by the parent component.
2. You are using a non-standard DOM attribute on a native DOM node, perhaps to represent custom data. If you are trying to attach custom data to a standard DOM element, consider using a custom data attribute as described [on MDN](#).
3. React does not yet recognize the attribute you specified. This will likely be fixed in a future version of React. However, React currently strips all unknown attributes, so specifying them in your React app will not cause them to be rendered.
4. You are using a React component without an upper case. React interprets it as a DOM tag because [React JSX transform uses the upper vs. lower case convention to distinguish between user-defined components and DOM tags](#).

To fix this, composite components should "consume" any prop that is intended for the composite component and not intended for the child component. Example:

**Bad:** Unexpected `layout` prop is forwarded to the `div` tag.

```
function MyDiv(props) {
  if (props.layout === 'horizontal') {
    // BAD! Because you know for sure "layout" is not a prop that <div>
    understands.
    return <div {...props} style={getHorizontalStyle()} />
  } else {
    // BAD! Because you know for sure "layout" is not a prop that <div>
    understands.
    return <div {...props} style={getVerticalStyle()} />
  }
}
```

**Good:** The spread operator can be used to pull variables off props, and put the remaining props into a variable.

```
function MyDiv(props) {
  const { layout, ...rest } = props
  if (layout === 'horizontal') {
    return <div {...rest} style={getHorizontalStyle()} />
  } else {
    return <div {...rest} style={getVerticalStyle()} />
  }
}
```

**Good:** You can also assign the props to a new object and delete the keys that you're using from the new object. Be sure not to delete the props from the original `this.props` object, since that object should be considered immutable.

```
function MyDiv(props) {

  const divProps = Object.assign({}, props);
```

```
delete divProps.layout;

if (props.layout === 'horizontal') {
  return <div {...divProps} style={getHorizontalStyle()} />
} else {
  return <div {...divProps} style={getVerticalStyle()} />
}
}
```

We discovered a minor vulnerability that might affect some apps using ReactDOMServer. We are releasing a patch version for every affected React minor release so that you can upgrade with no friction. Read on for more details.

## Short Description

Today, we are releasing a fix for a vulnerability we discovered in the `react-dom/server` implementation. It was introduced with the version 16.0.0 and has existed in all subsequent releases until today.

This vulnerability **can only affect some server-rendered React apps**. Purely client-rendered apps are **not** affected. Additionally, we expect that most server-rendered apps don't contain the vulnerable pattern described below. Nevertheless, we recommend to follow the mitigation instructions at the earliest opportunity.

While we were investigating this vulnerability, we found similar vulnerabilities in a few other popular front-end libraries. We have coordinated this release together with [Vue](#) and [Preact](#) releases fixing the same issue. The tracking number for this vulnerability is `CVE-2018-6341`.

## Mitigation

**We have prepared a patch release with a fix for every affected minor version.**

### 16.0.x

If you're using `react-dom/server` with this version:

- `react-dom@16.0.0`

Update to this version instead:

- `react-dom@16.0.1` **(contains the mitigation)**

### 16.1.x

If you're using `react-dom/server` with one of these versions:

- `react-dom@16.1.0`
- `react-dom@16.1.1`

Update to this version instead:

- `react-dom@16.1.2` **(contains the mitigation)**

### 16.2.x

If you're using `react-dom/server` with this version:

- `react-dom@16.2.0`

Update to this version instead:

- `react-dom@16.2.1` **(contains the mitigation)**

### 16.3.x

If you're using `react-dom/server` with one of these versions:

- `react-dom@16.3.0`

- `react-dom@16.3.1`
- `react-dom@16.3.2`

Update to this version instead:

- `react-dom@16.3.3` **(contains the mitigation)**

## 16.4.x

If you're using `react-dom/server` with one of these versions:

- `react-dom@16.4.0`
- `react-dom@16.4.1`

Update to this version instead:

- `react-dom@16.4.2` **(contains the mitigation)**

If you're using a newer version of `react-dom`, no action is required.

Note that only the `react-dom` package needs to be updated.

## Detailed Description

Your app might be affected by this vulnerability only if both of these two conditions are true:

- Your app is **being rendered to HTML using [ReactDOMServer API](#)**, and
- Your app **includes a user-supplied attribute name in an HTML tag**.

Specifically, the vulnerable pattern looks like this:

```
let props = {};  
props[userProvidedData] = "hello";  
let element = <div {...props} />;  
let html = ReactDOMServer.renderToString(element);
```

In order to exploit it, the attacker would need to craft a special attribute name that triggers an [XSS](#) vulnerability. For example:

```
let userProvidedData = '></div><script>alert("hi")</script>';
```

In the vulnerable versions of `react-dom/server`, the output would let the attacker inject arbitrary markup:

```
<div ></div><script>alert("hi")</script>
```

In the versions after the vulnerability was [fixed](#) (and before it was introduced), attributes with invalid names are skipped:

```
<div></div>
```

You would also see a warning about an invalid attribute name.

Note that **we expect attribute names based on user input to be very rare in practice**. It doesn't serve any common practical use case, and has other potential security implications that React can't guard against.

## Installation

React v16.4.2 is available on the npm registry.

To install React 16 with Yarn, run:

```
yarn add react@^16.4.2 react-dom@^16.4.2
```

To install React 16 with npm, run:

```
npm install --save react@^16.4.2 react-dom@^16.4.2
```

We also provide UMD builds of React via a CDN:

```
<script crossorigin src="https://unpkg.com/react@16/umd/react.production.min.js">
</script>
<script crossorigin src="https://unpkg.com/react-dom@16/umd/react-dom.production.min.js"></script>
```

Refer to the documentation for [detailed installation instructions](#).

## Changelog

### React DOM Server

- Fix a potential XSS vulnerability when the attacker controls an attribute name ( CVE-2018-6341 ). This fix is available in the latest `react-dom@16.4.2` , as well as in previous affected minor versions: `react-dom@16.0.1` , `react-dom@16.1.2` , `react-dom@16.2.1` , and `react-dom@16.3.3` . (@gaelarion in #13302)
- Fix a crash in the server renderer when an attribute is called `hasOwnProperty` . This fix is only available in `react-dom@16.4.2` . (@gaelarion in #13303)

React 16.4 included a [bugfix for `getDerivedStateFromProps`](#) which caused some existing bugs in React components to reproduce more consistently. If this release exposed a case where your application was using an anti-pattern and didn't work properly after the fix, we're sorry for the churn. In this post, we will explain some common anti-patterns with derived state and our preferred alternatives.

For a long time, the lifecycle `componentWillReceiveProps` was the only way to update state in response to a change in props without an additional render. In version 16.3, [we introduced a replacement lifecycle](#), `getDerivedStateFromProps` to solve the same use cases in a safer way. At the same time, we've realized that people have many misconceptions about how to use both methods, and we've found anti-patterns that result in subtle and confusing bugs. The `getDerivedStateFromProps` bugfix in 16.4 [makes derived state more predictable](#), so the results of misusing it are easier to notice.

#### Note

All of the anti-patterns described in this post apply to both the older `componentWillReceiveProps` and the newer `getDerivedStateFromProps`.

This blog post will cover the following topics:

- [When to use derived state](#)
- [Common bugs when using derived state](#)
  - [Anti-pattern: Unconditionally copying props to state](#)
  - [Anti-pattern: Erasing state when props change](#)
- [Preferred solutions](#)
- [What about memoization?](#)

## When to Use Derived State

`getDerivedStateFromProps` exists for only one purpose. It enables a component to update its internal state as the result of **changes in props**. Our previous blog post provided some examples, like [recording the current scroll direction based on a changing offset prop](#) or [loading external data specified by a source prop](#).

We did not provide many examples, because as a general rule, **derived state should be used sparingly**. All problems with derived state that we have seen can be ultimately reduced to either (1) unconditionally updating state from props or (2) updating state whenever props and state don't match. (We'll go over both in more detail below.)

- If you're using derived state to memoize some computation based only on the current props, you don't need derived state. See [What about memoization?](#) below.
- If you're updating derived state unconditionally or updating it whenever props and state don't match, your component likely resets its state too frequently. Read on for more details.

## Common Bugs When Using Derived State

The terms "[controlled](#)" and "[uncontrolled](#)" usually refer to form inputs, but they can also describe where any component's data lives. Data passed in as props can be thought of as **controlled** (because the parent component *controls* that data). Data that exists only in internal state can be thought of as **uncontrolled** (because the parent can't directly change it).

The most common mistake with derived state is mixing these two; when a derived state value is also updated by `setState` calls, there isn't a single source of truth for the data. The [external data loading example](#) mentioned above may sound similar, but it's different in a few important ways. In the loading example, there is a clear source of truth for both the "source" prop and the "loading" state. When the source prop changes, the loading state should **always** be overridden. Conversely, the state is overridden only when the prop **changes** and is otherwise managed by the component.

Problems arise when any of these constraints are changed. This typically comes in two forms. Let's take a look at both.

## Anti-pattern: Unconditionally copying props to state

A common misconception is that `getDerivedStateFromProps` and `componentWillReceiveProps` are only called when props "change". These lifecycles are called any time a parent component rerenders, regardless of whether the props are "different" from before. Because of this, it has always been unsafe to *unconditionally* override state using either of these lifecycles. **Doing so will cause state updates to be lost.**

Let's consider an example to demonstrate the problem. Here is a `EmailInput` component that "mirrors" an email prop in state:

```
class EmailInput extends Component {
  state = { email: this.props.email };

  render() {
    return <input onChange={this.handleChange} value={this.state.email} />;
  }

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

  componentWillReceiveProps(nextProps) {
    // This will erase any local state updates!
    // Do not do this.
    this.setState({ email: nextProps.email });
  }
}
```

At first, this component might look okay. State is initialized to the value specified by props and updated when we type into the `<input>`. But if our component's parent rerenders, anything we've typed into the `<input>` will be lost! (See [this demo for an example](#).) This holds true even if we were to compare `nextProps.email !== this.state.email` before resetting.

In this simple example, adding `shouldComponentUpdate` to rerender only when the email prop has changed could fix this. However in practice, components usually accept multiple props; another prop changing would still cause a rerender and improper reset. Function and object props are also often created inline, making it hard to implement a `shouldComponentUpdate` that reliably returns true only when a material change has happened. [Here is a demo that shows that happening](#). As a result, `shouldComponentUpdate` is best used as a performance optimization, not to ensure correctness of derived state.

Hopefully it's clear by now why **it is a bad idea to unconditionally copy props to state**. Before reviewing possible solutions, let's look at a related problematic pattern: what if we were to only update the state when the email prop changes?

## Anti-pattern: Erasing state when props change

Continuing the example above, we could avoid accidentally erasing state by only updating it when `props.email` changes:

```
class EmailInput extends Component {
  state = {
    email: this.props.email
  };

  componentWillReceiveProps(nextProps) {
    // Any time props.email changes, update state.
    if (nextProps.email !== this.props.email) {
```

```

    this.setState({
      email: nextProps.email
    });
  }
}

// ...
}

```

**Note**

Even though the example above shows `componentWillReceiveProps`, the same anti-pattern applies to `getDerivedStateFromProps`.

We've just made a big improvement. Now our component will erase what we've typed only when the props actually change.

There is still a subtle problem. Imagine a password manager app using the above input component. When navigating between details for two accounts with the same email, the input would fail to reset. This is because the prop value passed to the component would be the same for both accounts! This would be a surprise to the user, as an unsaved change to one account would appear to affect other accounts that happened to share the same email. ([See demo here.](#))

This design is fundamentally flawed, but it's also an easy mistake to make. ([I've made it myself!](#)) Fortunately there are two alternatives that work better. The key to both is that **for any piece of data, you need to pick a single component that owns it as the source of truth, and avoid duplicating it in other components.** Let's take a look at each of the alternatives.

## Preferred Solutions

### Recommendation: Fully controlled component

One way to avoid the problems mentioned above is to remove state from our component entirely. If the email address only exists as a prop, then we don't have to worry about conflicts with state. We could even convert `EmailInput` to a lighter-weight functional component:

```

function EmailInput(props) {
  return <input onChange={props.onChange} value={props.email} />;
}

```

This approach simplifies the implementation of our component, but if we still want to store a draft value, the parent form component will now need to do that manually. ([Click here to see a demo of this pattern.](#))

### Recommendation: Fully uncontrolled component with a `key`

Another alternative would be for our component to fully own the "draft" email state. In that case, our component could still accept a prop for the *initial* value, but it would ignore subsequent changes to that prop:

```

class EmailInput extends Component {
  state = { email: this.props.defaultEmail };

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

```



```
render() {
  return <input onChange={this.handleChange} value={this.state.email} />;
}
}
```

In order to reset the value when moving to a different item (as in our password manager scenario), we can use the special React attribute called `key`. When a `key` changes, React will *create a new component instance rather than update the current one*. Keys are usually used for dynamic lists but are also useful here. In our case, we could use the user ID to recreate the email input any time a new user is selected:

```
<EmailInput
  defaultEmail={this.props.user.email}
  key={this.props.user.id}
/>
```

Each time the ID changes, the `EmailInput` will be recreated and its state will be reset to the latest `defaultEmail` value. ([Click here to see a demo of this pattern.](#)) With this approach, you don't have to add `key` to every input. It might make more sense to put a `key` on the whole form instead. Every time the key changes, all components within the form will be recreated with a freshly initialized state.

In most cases, this is the best way to handle state that needs to be reset.

#### Note

While this may sound slow, the performance difference is usually insignificant. Using a key can even be faster if the components have heavy logic that runs on updates since diffing gets bypassed for that subtree.

## Alternative 1: Reset uncontrolled component with an ID prop

If `key` doesn't work for some reason (perhaps the component is very expensive to initialize), a workable but cumbersome solution would be to watch for changes to "userID" in `getDerivedStateFromProps`:

```
class EmailInput extends Component {
  state = {
    email: this.props.defaultEmail,
    prevPropsUserID: this.props.userID
  };

  static getDerivedStateFromProps(props, state) {
    // Any time the current user changes,
    // Reset any parts of state that are tied to that user.
    // In this simple example, that's just the email.
    if (props.userID !== state.prevPropsUserID) {
      return {
        prevPropsUserID: props.userID,
        email: props.defaultEmail
      };
    }
    return null;
  }
}
```

```
// ...  
}
```

This also provides the flexibility to only reset parts of our component's internal state if we so choose. ([Click here to see a demo of this pattern.](#))

#### Note

Even though the example above shows `getDerivedStateFromProps`, the same technique can be used with `componentWillReceiveProps`.

## Alternative 2: Reset uncontrolled component with an instance method

More rarely, you may need to reset state even if there's no appropriate ID to use as `key`. One solution is to reset the key to a random value or autoincrementing number each time you want to reset. One other viable alternative is to expose an instance method to imperatively reset the internal state:

```
class EmailInput extends Component {  
  state = {  
    email: this.props.defaultEmail  
  };  
  
  resetEmailForNewUser(newEmail) {  
    this.setState({ email: newEmail });  
  }  
  
  // ...  
}
```

The parent form component could then use a `ref` to call this method. ([Click here to see a demo of this pattern.](#))

Refs can be useful in certain cases like this one, but generally we recommend you use them sparingly. Even in the demo, this imperative method is nonideal because two renders will occur instead of one.

---

## Recap

To recap, when designing a component, it is important to decide whether its data will be controlled or uncontrolled.

Instead of trying to "**mirror**" a **prop value in state**, make the component **controlled**, and consolidate the two diverging values in the state of some parent component. For example, rather than a child accepting a "committed" `props.value` and tracking a "draft" `state.value`, have the parent manage both `state.draftValue` and `state.committedValue` and control the child's value directly. This makes the data flow more explicit and predictable.

For **uncontrolled** components, if you're trying to reset state when a particular prop (usually an ID) changes, you have a few options:

- **Recommendation: To reset *all internal state*, use the `key` attribute.**
- Alternative 1: To reset *only certain state fields*, watch for changes in a special property (e.g. `props.userID`).
- Alternative 2: You can also consider fall back to an imperative instance method using refs.

## What about memoization?

We've also seen derived state used to ensure an expensive value used in `render` is recomputed only when the inputs change. This technique is known as [memoization](#).

Using derived state for memoization isn't necessarily bad, but it's usually not the best solution. There is inherent complexity in managing derived state, and this complexity increases with each additional property. For example, if we add a second derived field to our component state then our implementation would need to separately track changes to both.

Let's look at an example of one component that takes one prop—a list of items—and renders the items that match a search query entered by the user. We could use derived state to store the filtered list:

```
class Example extends Component {
  state = {
    filterText: "",
  };

  // *****
  // NOTE: this example is NOT the recommended approach.
  // See the examples below for our recommendations instead.
  // *****

  static getDerivedStateFromProps(props, state) {
    // Re-run the filter whenever the list array or filter text change.
    // Note we need to store prevPropsList and prevFilterText to detect changes.
    if (
      props.list !== state.prevPropsList ||
      state.prevFilterText !== state.filterText
    ) {
      return {
        prevPropsList: props.list,
        prevFilterText: state.filterText,
        filteredList: props.list.filter(item =>
item.text.includes(state.filterText))
      };
    }
    return null;
  }

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

  render() {
    return (
      <Fragment>
        <input onChange={this.handleChange} value={this.state.filterText} />
        <ul>{this.state.filteredList.map(item => <li key={item.id}>{item.text}
</li>)}</ul>
      </Fragment>
    );
  }
}
```

This implementation avoids recalculating `filteredList` more often than necessary. But it is more complicated than it needs to be, because it has to separately track and detect changes in both props and state in order to properly update the filtered list. In this example, we could simplify things by using `PureComponent` and moving the filter operation into the render method:

```
// PureComponent only rerender if at least one state or prop value changes.
// Change is determined by doing a shallow comparison of state and prop keys.
class Example extends PureComponent {
  // State only needs to hold the current filter text value:
  state = {
    filterText: ""
  };

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

  render() {
    // The render method on this PureComponent is called only if
    // props.list or state.filterText has changed.
    const filteredList = this.props.list.filter(
      item => item.text.includes(this.state.filterText)
    )

    return (
      <Fragment>
        <input onChange={this.handleChange} value={this.state.filterText} />
        <ul>{filteredList.map(item => <li key={item.id}>{item.text}</li>)}</ul>
      </Fragment>
    );
  }
}
```

The above approach is much cleaner and simpler than the derived state version. Occasionally, this won't be good enough—filtering may be slow for large lists, and `PureComponent` won't prevent rerenders if another prop were to change. To address both of these concerns, we could add a memoization helper to avoid unnecessarily re-filtering our list:

```
import memoize from "memoize-one";

class Example extends Component {
  // State only needs to hold the current filter text value:
  state = { filterText: "" };

  // Re-run the filter whenever the list array or filter text changes:
  filter = memoize(
    (list, filterText) => list.filter(item => item.text.includes(filterText))
  );

  handleChange = event => {
    this.setState({ filterText: event.target.value });
  };
}
```

```
render() {
  // Calculate the latest filtered list. If these arguments haven't changed
  // since the last render, `memoize-one` will reuse the last return value.
  const filteredList = this.filter(this.props.list, this.state.filterText);

  return (
    <Fragment>
      <input onChange={this.handleChange} value={this.state.filterText} />
      <ul>{filteredList.map(item => <li key={item.id}>{item.text}</li>)}</ul>
    </Fragment>
  );
}
```

This is much simpler and performs just as well as the derived state version!

When using memoization, remember a couple of constraints:

1. In most cases, you'll want to **attach the memoized function to a component instance**. This prevents multiple instances of a component from resetting each other's memoized keys.
2. Typically you'll want to use a memoization helper with a **limited cache size** in order to prevent memory leaks over time. (In the example above, we used `memoize-one` because it only caches the most recent arguments and result.)
3. None of the implementations shown in this section will work if `props.list` is recreated each time the parent component renders. But in most cases, this setup is appropriate.

## In closing

In real world applications, components often contain a mix of controlled and uncontrolled behaviors. This is okay! If each value has a clear source of truth, you can avoid the anti-patterns mentioned above.

It is also worth re-iterating that `getDerivedStateFromProps` (and derived state in general) is an advanced feature and should be used sparingly because of this complexity. If your use case falls outside of these patterns, please share it with us on [GitHub](#) or [Twitter](#)!

The latest minor release adds support for an oft-requested feature: pointer events!

It also includes a bugfix for `getDerivedStateFromProps`. Check out the full [changelog](#) below.

## Pointer Events

The following event types are now available in React DOM:

- `onPointerDown`
- `onPointerMove`
- `onPointerUp`
- `onPointerCancel`
- `onGotPointerCapture`
- `onLostPointerCapture`
- `onPointerEnter`
- `onPointerLeave`
- `onPointerOver`
- `onPointerOut`

Please note that these events will only work in browsers that support the [Pointer Events](#) specification. (At the time of this writing, this includes the latest versions of Chrome, Firefox, Edge, and Internet Explorer.) If your application depends on pointer events, we recommend using a third-party pointer events polyfill. We have opted not to include such a polyfill in React DOM, to avoid an increase in bundle size.

[Check out this example on CodeSandbox.](#)

Huge thanks to [Philipp Spiess](#) for contributing this change!

## Bugfix for `getDerivedStateFromProps`

`getDerivedStateFromProps` is now called every time a component is rendered, regardless of the cause of the update. Previously, it was only called if the component was re-rendered by its parent, and would not fire as the result of a local `setState`. This was an oversight in the initial implementation that has now been corrected. The previous behavior was more similar to `componentWillReceiveProps`, but the improved behavior ensures compatibility with React's upcoming asynchronous rendering mode.

**This bug fix will not affect most apps**, but it may cause issues with a small fraction of components. The rare cases where it does matter fall into one of two categories:

### 1. Avoid Side Effects in `getDerivedStateFromProps`

Like the render method, `getDerivedStateFromProps` should be a pure function of props and state. Side effects in `getDerivedStateFromProps` were never supported, but since it now fires more often than it used to, the recent change may expose previously undiscovered bugs.

Side effectful code should be moved to other methods: for example, Flux dispatches typically belong inside the originating event handler, and manual DOM mutations belong inside `componentDidMount` or `componentDidUpdate`. You can read more about this in our recent post about [preparing for asynchronous rendering](#).

### 2. Compare Incoming Props to Previous Props When Computing Controlled Values

The following code assumes `getDerivedStateFromProps` only fires on prop changes:

```
static getDerivedStateFromProps(props, state) {
  if (props.value !== state.controlledValue) {
    return {
      // Since this method fires on both props and state changes, local updates
      // to the controlled value will be ignored, because the props version
      // always overrides it. Oops!
      controlledValue: props.value,
    };
  }
  return null;
}
```

One possible way to fix this is to compare the incoming value to the previous value by storing the previous props in state:

```
static getDerivedStateFromProps(props, state) {
  const prevProps = state.prevProps;
  // Compare the incoming prop to previous prop
  const controlledValue =
    prevProps.value !== props.value
      ? props.value
      : state.controlledValue;
  return {
    // Store the previous props in state
    prevProps: props,
    controlledValue,
  };
}
```

However, **code that "mirrors" props in state usually contains bugs**, whether you use the newer `getDerivedStateFromProps` or the legacy `componentWillReceiveProps`. We published a follow-up blog post that explains these problems in more detail, and suggests [simpler solutions that don't involve `getDerivedStateFromProps\(\)`](#).

## Installation

React v16.4.0 is available on the npm registry.

To install React 16 with Yarn, run:

```
yarn add react@^16.4.0 react-dom@^16.4.0
```

To install React 16 with npm, run:

```
npm install --save react@^16.4.0 react-dom@^16.4.0
```

We also provide UMD builds of React via a CDN:

```
<script crossorigin src="https://unpkg.com/react@16/umd/react.production.min.js">
</script>
<script crossorigin src="https://unpkg.com/react-dom@16/umd/react-
```

```
dom.production.min.js"></script>
```

Refer to the documentation for [detailed installation instructions](#).

## Changelog

### React

- Add a new [experimental](#) `React.unstable_Profiler` component for measuring performance. ([@bvaughn](#) in [#12745](#))

### React DOM

- Add support for the Pointer Events specification. ([@philipp-spiess](#) in [#12507](#))
- Properly call `getDerivedStateFromProps()` regardless of the reason for re-rendering. ([@acdlite](#) in [#12600](#) and [#12802](#))
- Fix a bug that prevented context propagation in some cases. ([@gaearon](#) in [#12708](#))
- Fix re-rendering of components using `forwardRef()` on a deeper `setState()`. ([@gaearon](#) in [#12690](#))
- Fix some attributes incorrectly getting removed from custom element nodes. ([@airamrguez](#) in [#12702](#))
- Fix context providers to not bail out on children if there's a legacy context provider above. ([@gaearon](#) in [#12586](#))
- Add the ability to specify `propTypes` on a context provider component. ([@nicolevy](#) in [#12658](#))
- Fix a false positive warning when using `react-lifecycles-compat` in `<StrictMode>`. ([@bvaughn](#) in [#12644](#))
- Warn when the `forwardRef()` render function has `propTypes` or `defaultProps`. ([@bvaughn](#) in [#12644](#))
- Improve how `forwardRef()` and context consumers are displayed in the component stack. ([@sophiebits](#) in [#12777](#))
- Change internal event names. This can break third-party packages that rely on React internals in unsupported ways. ([@philipp-spiess](#) in [#12629](#))

### React Test Renderer

- Fix the `getDerivedStateFromProps()` support to match the new React DOM behavior. ([@koba04](#) in [#12676](#))
- Fix a `testInstance.parent` crash when the parent is a fragment or another special node. ([@gaearon](#) in [#12813](#))
- `forwardRef()` components are now discoverable by the test renderer traversal methods. ([@gaearon](#) in [#12725](#))
- Shallow renderer now ignores `setState()` updaters that return `null` or `undefined`. ([@koba04](#) in [#12756](#))

### React ART

- Fix reading context provided from the tree managed by React DOM. ([@acdlite](#) in [#12779](#))

### React Call Return (Experimental)

- This experiment was deleted because it was affecting the bundle size and the API wasn't good enough. It's likely to come back in the future in some other form. ([@gaearon](#) in [#12820](#))

### React Reconciler (Experimental)

- The [new host config shape](#) is flat and doesn't use nested objects. ([@gaearon](#) in [#12792](#))



A few days ago, we [wrote a post about upcoming changes to our legacy lifecycle methods](#), including gradual migration strategies. In React 16.3.0, we are adding a few new lifecycle methods to assist with that migration. We are also introducing new APIs for long requested features: an official context API, a ref forwarding API, and an ergonomic ref API.

Read on to learn more about the release.

## Official Context API

For many years, React has offered an experimental API for context. Although it was a powerful tool, its use was discouraged because of inherent problems in the API, and because we always intended to replace the experimental API with a better one.

Version 16.3 introduces a new context API that is more efficient and supports both static type checking and deep updates.

### Note

The old context API will keep working for all React 16.x releases, so you will have time to migrate.

Here is an example illustrating how you might inject a "theme" using the new context API: [embed:16-3-release-blog-post/context-example.js](#)

[Learn more about the new context API here.](#)

## `createRef` API

Previously, React provided two ways of 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.

Version 16.3 adds a new option for managing refs that offers the convenience of a string ref without any of the downsides:

[embed:16-3-release-blog-post/create-ref-example.js](#)

### 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.](#)

## `forwardRef` API

Generally, React components are declarative, but sometimes imperative access to the component instances and the underlying DOM nodes is necessary. This is common for use cases like managing focus, selection, or animations. React provides [refs](#) as a way to solve this problem. However, component encapsulation poses some challenges with refs.

For example, if you replace a `<button>` with a custom `<FancyButton>` component, the `ref` attribute on it will start pointing at the wrapper component instead of the DOM node (and would be `null` for functional components). While this is desirable for "application-level" components like `FeedStory` or `Comment` that need to be encapsulated, it can be annoying for "leaf" components such as `FancyButton` or `MyTextInput` that are typically used like their DOM counterparts, and might need to expose their DOM nodes.

Ref forwarding is a new 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` forwards its ref to a DOM `button` that it renders:

[embed:16-3-release-blog-post/fancy-button-example.js](#)

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.

Ref forwarding is not limited to "leaf" components that render DOM nodes. If you write [higher order components](#), we recommend using ref forwarding to automatically pass the ref down to the wrapped class component instances.

[Learn more about the `forwardRef` API here.](#)

## Component Lifecycle Changes

React's class component API has been around for years with little change. However, as we add support for more advanced features (such as [error boundaries](#) and the upcoming [async rendering mode](#)) we stretch this model in ways that it was not originally intended.

For example, with the current API, it is too easy to block the initial render with non-essential logic. In part this is because there are too many ways to accomplish a given task, and it can be unclear which is best. We've observed that the interrupting behavior of error handling is often not taken into consideration and can result in memory leaks (something that will also impact the upcoming async rendering mode). The current class component API also complicates other efforts, like our work on [prototyping a React compiler](#).

Many of these issues are exacerbated by a subset of the component lifecycles ( `componentWillMount` , `componentWillReceiveProps` , and `componentWillUpdate` ). These also happen to be the lifecycles that cause the most confusion within the React community. For these reasons, we are going to deprecate those methods in favor of better alternatives.

We recognize that this change will impact many existing components. Because of this, the migration path will be as gradual as possible, and will provide escape hatches. (At Facebook, we maintain more than 50,000 React components. We depend on a gradual release cycle too!)

### Note:

Deprecation warnings will be enabled with a future 16.x release, **but the legacy lifecycles will continue to work until version 17.**

Even in version 17, it will still be possible to use them, but they will be aliased with an "UNSAFE\_" prefix to indicate that they might cause issues. We have also prepared an [automated script to rename them](#) in existing code.

In addition to deprecating unsafe lifecycles, we are also adding a couple of new lifecycles:

- `getDerivedStateFromProps` is being added as a safer alternative to the legacy `componentWillReceiveProps` . (Note that [in most cases you don't need either of them.](#))
- `getSnapshotBeforeUpdate` is being added to support safely reading properties from e.g. the DOM before updates are made.

[Learn more about these lifecycle changes here.](#)

## StrictMode Component

`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:

`StrictMode` checks are run in development mode only; *they do not impact the production build.*

Although it is not possible for strict mode to catch all problems (e.g. certain types of mutation), it can help with many. If you see warnings in strict mode, those things will likely cause bugs for async rendering.

In version 16.3, `StrictMode` helps with:

- Identifying components with unsafe lifecycles
- Warning about legacy string ref API usage
- Detecting unexpected side effects

Additional functionality will be added with future releases of React.

[Learn more about the `StrictMode` component here.](#)

For over a year, the React team has been working to implement asynchronous rendering. Last month during his talk at JSConf Iceland, [Dan unveiled some of the exciting new possibilities async rendering unlocks](#). Now we'd like to share with you some of the lessons we've learned while working on these features, and some recipes to help prepare your components for async rendering when it launches.

One of the biggest lessons we've learned is that some of our legacy component lifecycles tend to encourage unsafe coding practices. They are:

- `componentWillMount`
- `componentWillReceiveProps`
- `componentWillUpdate`

These lifecycle methods have often been misunderstood and subtly misused; furthermore, we anticipate that their potential misuse may be more problematic with async rendering. Because of this, we will be adding an "UNSAFE\_" prefix to these lifecycles in an upcoming release. (Here, "unsafe" refers not to security but instead conveys that code using these lifecycles will be more likely to have bugs in future versions of React, especially once async rendering is enabled.)

## Gradual Migration Path

React follows [semantic versioning](#), so this change will be gradual. Our current plan is:

- **16.3:** Introduce aliases for the unsafe lifecycles, `UNSAFE_componentWillMount`, `UNSAFE_componentWillReceiveProps`, and `UNSAFE_componentWillUpdate`. (Both the old lifecycle names and the new aliases will work in this release.)
- **A future 16.x release:** Enable deprecation warning for `componentWillMount`, `componentWillReceiveProps`, and `componentWillUpdate`. (Both the old lifecycle names and the new aliases will work in this release, but the old names will log a DEV-mode warning.)
- **17.0:** Remove `componentWillMount`, `componentWillReceiveProps`, and `componentWillUpdate`. (Only the new "UNSAFE\_" lifecycle names will work from this point forward.)

**Note that if you're a React application developer, you don't have to do anything about the legacy methods yet. The primary purpose of the upcoming version 16.3 release is to enable open source project maintainers to update their libraries in advance of any deprecation warnings. Those warnings will not be enabled until a future 16.x release.**

We maintain over 50,000 React components at Facebook, and we don't plan to rewrite them all immediately. We understand that migrations take time. We will take the gradual migration path along with everyone in the React community.

---

## Migrating from Legacy Lifecycles

If you'd like to start using the new component APIs introduced in React 16.3 (or if you're a maintainer looking to update your library in advance) here are a few examples that we hope will help you to start thinking about components a bit differently. Over time, we plan to add additional "recipes" to our documentation that show how to perform common tasks in a way that avoids the problematic lifecycles.

Before we begin, here's a quick overview of the lifecycle changes planned for version 16.3:

- We are **adding the following lifecycle aliases**: `UNSAFE_componentWillMount`, `UNSAFE_componentWillReceiveProps`, and `UNSAFE_componentWillUpdate`. (Both the old lifecycle names and the new aliases will be supported.)
- We are **introducing two new lifecycles**, static `getDerivedStateFromProps` and `getSnapshotBeforeUpdate`.

### New lifecycle: `getDerivedStateFromProps`

embed:update-on-async-rendering/definition-getderivedstatefromprops.js

The new static `getDerivedStateFromProps` lifecycle is invoked after a component is instantiated as well as before it is re-rendered. It can return an object to update `state`, or `null` to indicate that the new `props` do not require any `state` updates.

Together with `componentDidUpdate`, this new lifecycle should cover all use cases for the legacy `componentWillReceiveProps`.

Note:

Both the older `componentWillReceiveProps` and the new `getDerivedStateFromProps` methods add significant complexity to components. This often leads to [bugs](#). Consider [simpler alternatives to derived state](#) to make components predictable and maintainable.

## New lifecycle: `getSnapshotBeforeUpdate`

`embed:update-on-async-rendering/definition-getsnapshotbeforeupdate.js`

The new `getSnapshotBeforeUpdate` lifecycle is called right before mutations are made (e.g. before the DOM is updated). The return value for this lifecycle will be passed as the third parameter to `componentDidUpdate`. (This lifecycle isn't often needed, but can be useful in cases like manually preserving scroll position during rerenders.)

Together with `componentDidUpdate`, this new lifecycle should cover all use cases for the legacy `componentWillUpdate`.

You can find their type signatures [in this gist](#).

We'll look at examples of how both of these lifecycles can be used below.

## Examples

- [Initializing state](#)
- [Fetching external data](#)
- [Adding event listeners \(or subscriptions\)](#)
- [Updating `state` based on props](#)
- [Invoking external callbacks](#)
- [Side effects on props change](#)
- [Fetching external data when props change](#)
- [Reading DOM properties before an update](#)

Note

For brevity, the examples below are written using the experimental class properties transform, but the same migration strategies apply without it.

### Initializing state

This example shows a component with `setState` calls inside of `componentWillMount`: `embed:update-on-async-rendering/initializing-state-before.js`

The simplest refactor for this type of component is to move state initialization to the constructor or to a property initializer, like so: `embed:update-on-async-rendering/initializing-state-after.js`

### Fetching external data

Here is an example of a component that uses `componentWillMount` to fetch external data: `embed:update-on-async-rendering/fetching-external-data-before.js`

The above code is problematic for both server rendering (where the external data won't be used) and the upcoming async rendering mode (where the request might be initiated multiple times).

The recommended upgrade path for most use cases is to move data-fetching into `componentDidMount` : `embed:update-on-async-rendering/fetching-external-data-after.js`

There is a common misconception that fetching in `componentWillMount` lets you avoid the first empty rendering state. In practice this was never true because React has always executed `render` immediately after `componentWillMount` . If the data is not available by the time `componentWillMount` fires, the first `render` will still show a loading state regardless of where you initiate the fetch. This is why moving the fetch to `componentDidMount` has no perceptible effect in the vast majority of cases.

#### Note

Some advanced use-cases (e.g. libraries like Relay) may want to experiment with eagerly prefetching async data. An example of how this can be done is available [here](#).

In the longer term, the canonical way to fetch data in React components will likely be based on the “suspense” API [introduced at JSConf Iceland](#). Both simple data fetching solutions and libraries like Apollo and Relay will be able to use it under the hood. It is significantly less verbose than either of the above solutions, but will not be finalized in time for the 16.3 release.

When supporting server rendering, it's currently necessary to provide the data synchronously – `componentWillMount` was often used for this purpose but the constructor can be used as a replacement. The upcoming suspense APIs will make async data fetching cleanly possible for both client and server rendering.

## Adding event listeners (or subscriptions)

Here is an example of a component that subscribes to an external event dispatcher when mounting: `embed:update-on-async-rendering/adding-event-listeners-before.js`

Unfortunately, this can cause memory leaks for server rendering (where `componentWillUnmount` will never be called) and async rendering (where rendering might be interrupted before it completes, causing `componentWillUnmount` not to be called).

People often assume that `componentWillMount` and `componentWillUnmount` are always paired, but that is not guaranteed. Only once `componentDidMount` has been called does React guarantee that `componentWillUnmount` will later be called for clean up.

For this reason, the recommended way to add listeners/subscriptions is to use the `componentDidMount` lifecycle: `embed:update-on-async-rendering/adding-event-listeners-after.js`

Sometimes it is important to update subscriptions in response to property changes. If you're using a library like Redux or MobX, the library's container component should handle this for you. For application authors, we've created a small library, [create-subscription](#) , to help with this. We'll publish it along with React 16.3.

Rather than passing a subscribable `dataSource` prop as we did in the example above, we could use `create-subscription` to pass in the subscribed value:

`embed:update-on-async-rendering/adding-event-listeners-create-subscription.js`

#### Note

Libraries like Relay/Apollo should manage subscriptions manually with the same techniques as `create-subscription` uses under the hood (as referenced [here](#)) in a way that is most optimized for their library usage.

## Updating state based on props

#### Note:

Both the older `componentWillReceiveProps` and the new `getDerivedStateFromProps` methods add significant complexity to components. This often leads to [bugs](#). Consider [simpler alternatives to derived state](#) to make components predictable and maintainable.

Here is an example of a component that uses the legacy `componentWillReceiveProps` lifecycle to update `state` based on new `props` values: `embed:update-on-async-rendering/updating-state-from-props-before.js`

Although the above code is not problematic in itself, the `componentWillReceiveProps` lifecycle is often mis-used in ways that *do* present problems. Because of this, the method will be deprecated.

As of version 16.3, the recommended way to update `state` in response to `props` changes is with the new `static getDerivedStateFromProps` lifecycle. (That lifecycle is called when a component is created and each time it receives new props): `embed:update-on-async-rendering/updating-state-from-props-after.js`

You may notice in the example above that `props.currentRow` is mirrored in state (as `state.lastRow`). This enables `getDerivedStateFromProps` to access the previous props value in the same way as is done in `componentWillReceiveProps`.

You may wonder why we don't just pass previous props as a parameter to `getDerivedStateFromProps`. We considered this option when designing the API, but ultimately decided against it for two reasons:

- A `prevProps` parameter would be null the first time `getDerivedStateFromProps` was called (after instantiation), requiring an if-not-null check to be added any time `prevProps` was accessed.
- Not passing the previous props to this function is a step toward freeing up memory in future versions of React. (If React does not need to pass previous props to lifecycles, then it does not need to keep the previous `props` object in memory.)

#### Note

If you're writing a shared component, the `react-lifecycles-compat` polyfill enables the new `getDerivedStateFromProps` lifecycle to be used with older versions of React as well. [Learn more about how to use it below.](#)

## Invoking external callbacks

Here is an example of a component that calls an external function when its internal state changes: `embed:update-on-async-rendering/invoking-external-callbacks-before.js`

Sometimes people use `componentWillUpdate` out of a misplaced fear that by the time `componentDidUpdate` fires, it is "too late" to update the state of other components. This is not the case. React ensures that any `setState` calls that happen during `componentDidMount` and `componentDidUpdate` are flushed before the user sees the updated UI. In general, it is better to avoid cascading updates like this, but in some cases they are necessary (for example, if you need to position a tooltip after measuring the rendered DOM element).

Either way, it is unsafe to use `componentWillUpdate` for this purpose in async mode, because the external callback might get called multiple times for a single update. Instead, the `componentDidUpdate` lifecycle should be used since it is guaranteed to be invoked only once per update: `embed:update-on-async-rendering/invoking-external-callbacks-after.js`

## Side effects on props change

Similar to the [example above](#), sometimes components have side effects when `props` change.

`embed:update-on-async-rendering/side-effects-when-props-change-before.js`

Like `componentWillUpdate`, `componentWillReceiveProps` might get called multiple times for a single update. For this reason it is important to avoid putting side effects in this method. Instead, `componentDidUpdate` should be used since it is guaranteed to be invoked only once per update:

`embed:update-on-async-rendering/side-effects-when-props-change-after.js`

## Fetching external data when `props` change

Here is an example of a component that fetches external data based on `props` values: `embed:update-on-async-rendering/updating-external-data-when-props-change-before.js`

The recommended upgrade path for this component is to move data updates into `componentDidUpdate`. You can also use the new `getDerivedStateFromProps` lifecycle to clear stale data before rendering the new props: `embed:update-on-async-rendering/updating-external-data-when-props-change-after.js`

#### Note

If you're using an HTTP library that supports cancellation, like [axios](#), then it's simple to cancel an in-progress request when unmounting. For native Promises, you can use an approach like [the one shown here](#).

## Reading DOM properties before an update

Here is an example of a component that reads a property from the DOM before an update in order to maintain scroll position within a list: `embed:update-on-async-rendering/react-dom-properties-before-update-before.js`

In the above example, `componentWillUpdate` is used to read the DOM property. However with async rendering, there may be delays between "render" phase lifecycles (like `componentWillUpdate` and `render`) and "commit" phase lifecycles (like `componentDidUpdate`). If the user does something like resize the window during this time, the `scrollHeight` value read from `componentWillUpdate` will be stale.

The solution to this problem is to use the new "commit" phase lifecycle, `getSnapshotBeforeUpdate`. This method gets called *immediately before* mutations are made (e.g. before the DOM is updated). It can return a value for React to pass as a parameter to `componentDidUpdate`, which gets called *immediately after* mutations.

The two lifecycles can be used together like this:

```
embed:update-on-async-rendering/react-dom-properties-before-update-after.js
```

#### Note

If you're writing a shared component, the `react-lifecycles-compat` polyfill enables the new `getSnapshotBeforeUpdate` lifecycle to be used with older versions of React as well. [Learn more about how to use it below](#).

## Other scenarios

While we tried to cover the most common use cases in this post, we recognize that we might have missed some of them. If you are using `componentWillMount`, `componentWillUpdate`, or `componentWillReceiveProps` in ways that aren't covered by this blog post, and aren't sure how to migrate off these legacy lifecycles, please [file a new issue against our documentation](#) with your code examples and as much background information as you can provide. We will update this document with new alternative patterns as they come up.

## Open source project maintainers

Open source maintainers might be wondering what these changes mean for shared components. If you implement the above suggestions, what happens with components that depend on the new static `getDerivedStateFromProps` lifecycle? Do you also have to release a new major version and drop compatibility for React 16.2 and older?

Fortunately, you do not!

When React 16.3 is published, we'll also publish a new npm package, `react-lifecycles-compat`. This package polyfills components so that the new `getDerivedStateFromProps` and `getSnapshotBeforeUpdate` lifecycles will also work with older versions of React (0.14.9+).

To use this polyfill, first add it as a dependency to your library:

```
# Yarn
```



```
yarn add react-lifecycles-compat

# NPM
npm install react-lifecycles-compat --save
```

Next, update your components to use the new lifecycles (as described above).

Lastly, use the polyfill to make your component backwards compatible with older versions of React: `embed:update-on-async-rendering/using-react-lifecycles-compat.js`

[Dan Abramov](#) from our team just spoke at [JSConf Iceland 2018](#) with a preview of some new features we've been working on in React. The talk opens with a question: "With vast differences in computing power and network speed, how do we deliver the best user experience for everyone?"

Here's the video courtesy of JSConf Iceland:



I think you'll enjoy the talk more if you stop reading here and just watch the video. If you don't have time to watch, a (very) brief summary follows.

## About the Two Demos

On the first demo, Dan says: "We've built a generic way to ensure that high-priority updates don't get blocked by a low-priority update, called **time slicing**. If my device is fast enough, it feels almost like it's synchronous; if my device is slow, the app still feels responsive. It adapts to the device thanks to the [requestIdleCallback](#) API. Notice that only the final state was displayed; the rendered screen is always consistent and we don't see visual artifacts of slow rendering causing a janky user experience."

On the second demo, Dan explains: "We've built a generic way for components to suspend rendering while they load async data, which we call **suspense**. You can pause any state update until the data is ready, and you can add async loading to any component deep in the tree without plumbing all the props and state through your app and hoisting the logic. On a fast network, updates appear very fluid and instantaneous without a jarring cascade of spinners that appear and disappear. On a slow network, you can intentionally design which loading states the user should see and how granular or coarse they should be, instead of showing spinners based on how the code is written. The app stays responsive throughout."

"Importantly, this is still the React you know. This is still the declarative component paradigm that you probably like about React."

We can't wait to release these new async rendering features later this year. Follow this blog and [@reactjs on Twitter](#) for updates.

As we worked on [React 16](#), we revamped the folder structure and much of the build tooling in the React repository. Among other things, we introduced projects such as [Rollup](#), [Prettier](#), and [Google Closure Compiler](#) into our workflow. People often ask us questions about how we use those tools. In this post, we would like to share some of the changes that we've made to our build and test infrastructure in 2017, and what motivated them.

While these changes helped us make React better, they don't affect most React users directly. However, we hope that blogging about them might help other library authors solve similar problems. Our contributors might also find these notes helpful!

## Formatting Code with Prettier

React was one of the first large repositories to [fully embrace](#) opinionated automatic code formatting with [Prettier](#). Our current Prettier setup consists of:

- A local `yarn prettier` script that [uses the Prettier Node API](#) to format files in place. We typically run it before committing changes. It is fast because it only checks the [files changed since diverging from remote master](#).
- A script that [runs Prettier](#) as part of our [continuous integration checks](#). It won't attempt to overwrite the files, but instead will fail the build if any file differs from the Prettier output for that file. This ensures that we can't merge a pull request unless it has been fully formatted.

Some team members have also set up the [editor integrations](#). Our experience with Prettier has been fantastic, and we recommend it to any team that writes JavaScript.

## Restructuring the Monorepo

Ever since React was split into packages, it has been a [monorepo](#): a set of packages under the umbrella of a single repository. This made it easier to coordinate changes and share the tooling, but our folder structure was deeply nested and difficult to understand. It was not clear which files belonged to which package. After releasing React 16, we've decided to completely reorganize the repository structure. Here is how we did it.

### Migrating to Yarn Workspaces

The Yarn package manager [introduced a feature called Workspaces](#) a few months ago. This feature lets you tell Yarn where your monorepo's packages are located in the source tree. Every time you run `yarn`, in addition to installing your dependencies it also sets up the symlinks that point from your project's `node_modules` to the source folders of your packages.

Thanks to Workspaces, absolute imports between our own packages (such as importing `react` from `react-dom`) "just work" with any tools that support the Node resolution mechanism. The only problem we encountered was Jest not running the transforms inside the linked packages, but we [found a fix](#), and it was merged into Jest.

To enable Yarn Workspaces, we added `"workspaces": ["packages/*"]` to our `package.json`, and moved all the code into [top-level packages/\\* folders](#), each with its own `package.json` file.

Each package is structured in a similar way. For every public API entry point such as `react-dom` or `react-dom/server`, there is a [file](#) in the package root folder that re-exports the implementation from the `/src/` subfolder. The decision to point entry points to the source rather than to the built versions was intentional. Typically, we re-run a subset of tests after every change during development. Having to build the project to run a test would have been prohibitively slow. When we publish packages to npm, we replace these entry points with files in the `/npm/` folder that point to the build artifacts.

Not all packages have to be published on npm. For example, we keep some utilities that are tiny enough and can be safely duplicated in a [pseudo-package called shared](#). Our bundler is configured to [only treat dependencies declared from package.json as externals](#) so it happily bundles the `shared` code into `react` and `react-dom` without leaving any references to `shared/` in the build artifacts. So you can use Yarn Workspaces even if you don't plan to publish actual npm packages!

---

## Removing the Custom Module System

In the past, we used a non-standard module system called "Haste" that lets you import any file from any other file by its unique `@providesModule` directive no matter where it is in the tree. It neatly avoids the problem of deep relative imports with paths like `../../../../` and is great for the product code. However, this makes it hard to understand the dependencies between packages. We also had to resort to hacks to make it work with different tools.

We decided to [remove Haste](#) and use the Node resolution with relative imports instead. To avoid the problem of deep relative paths, we have [flattened our repository structure](#) so that it goes at most one level deep inside each package:

```
| -react
|   | -npm
|   | -src
| -react-dom
|   | -npm
|   | -src
|   |   | -client
|   |   | -events
|   |   | -server
|   |   | -shared
```

This way, the relative paths can only contain one `./` or `../` followed by the filename. If one package needs to import something from another package, it can do so with an absolute import from a top-level entry point.

In practice, we still have [some cross-package "internal" imports](#) that violate this principle, but they're explicit, and we plan to gradually get rid of them.

## Compiling Flat Bundles

Historically, React was distributed in two different formats: as a single-file build that you can add as a `<script>` tag in the browser, and as a collection of CommonJS modules that you can bundle with a tool like webpack or Browserify.

Before React 16, each React source file had a corresponding CommonJS module that was published as part of the npm packages. Importing `react` or `react-dom` led bundlers to the package [entry point](#) from which they would build a dependency tree with the CommonJS modules in the [internal lib folder](#).

However, this approach had multiple disadvantages:

- **It was inconsistent.** Different tools produce bundles of different sizes for identical code importing React, with the difference going as far as 30 kB (before gzip).
- **It was inefficient for bundler users.** The code produced by most bundlers today contains a lot of "glue code" at the module boundaries. It keeps the modules isolated from each other, but increases the parse time, the bundle size, and the build time.
- **It was inefficient for Node users.** When running in Node, performing `process.env.NODE_ENV` checks before development-only code incurs the overhead of actually looking up environment variables. This slowed down React server rendering. We couldn't cache it in a variable either because it prevented dead code elimination with Uglify.
- **It broke encapsulation.** React internals were exposed both in the open source (as `react-dom/lib/*` imports) and internally at Facebook. It was convenient at first as a way to share utilities between projects, but with time it became a maintenance burden because renaming or changing argument types of internal functions would break unrelated projects.
- **It prevented experimentation.** There was no way for the React team to experiment with any advanced compilation techniques. For example, in theory, we might want to apply [Google Closure Compiler Advanced](#) optimizations or [Prepack](#) to some of our code, but they are designed to work on complete bundles rather than small individual modules that we used to ship to npm.

Due to these and other issues, we've changed the strategy in React 16. We still ship CommonJS modules for Node.js and bundlers, but instead of publishing many individual files in the npm package, we publish just two CommonJS bundles per entry point.

For example, when you import `react` with React 16, the bundler [finds the entry point](#) that just re-exports one of the two files:

```
'use strict';

if (process.env.NODE_ENV === 'production') {
  module.exports = require('./cjs/react.production.min.js');
} else {
  module.exports = require('./cjs/react.development.js');
}
```

In every package provided by React, the `cjs` folder (short for "CommonJS") contains a development and a production pre-built bundle for each entry point.

For example, `react.development.js` is the version intended for development. It is readable and includes comments. On the other hand, `react.production.min.js` was minified and optimized before it was published to npm.

Note how this is essentially the same strategy that we've been using for the single-file browser builds (which now reside in the `umd` directory, short for [Universal Module Definition](#)). Now we just apply the same strategy to the CommonJS builds as well.

## Migrating to Rollup

Just compiling CommonJS modules into single-file bundles doesn't solve all of the above problems. The really significant wins came from [migrating our build system](#) from Browserify to Rollup.

Rollup was designed with [libraries rather than apps in mind](#), and it is a perfect fit for React's use case. It solves one problem well: how to combine multiple modules into a flat file with minimal junk code in between. To achieve this, instead of turning modules into functions like many other bundlers, it puts all the code in the same scope, and renames variables so that they don't conflict. This produces code that is easier for the JavaScript engine to parse, for a human to read, and for a minifier to optimize.

Rollup currently doesn't support some features that are important to application builders, such as code splitting. However, it does not aim to replace tools like webpack that do a great job at this. Rollup is a perfect fit for *libraries* like React that can be pre-built and then integrated into apps.

You can find our Rollup build configuration [here](#), with a [list of plugins we currently use](#).

## Migrating to Google Closure Compiler

After migrating to flat bundles, we [started](#) using [the JavaScript version of the Google Closure Compiler](#) in its "simple" mode. In our experience, even with the advanced optimizations disabled, it still provided a significant advantage over Uglify, as it was able to better eliminate dead code and automatically inline small functions when appropriate.

At first, we could only use Google Closure Compiler for the React bundles we shipped in the open source. At Facebook, we still needed the checked-in bundles to be unminified so we could symbolicate React production crashes with our error reporting tools. We ended up contributing [a flag](#) that completely disables the renaming compiler pass. This lets us apply other optimizations like function inlining, but keep the code fully readable for the Facebook-specific builds of React. To improve the output readability, we [also format that custom build using Prettier](#). Interestingly, running Prettier on production bundles while debugging the build process is a great way to find unnecessary code in the bundles!

Currently, all production React bundles [run through Google Closure Compiler in simple mode](#), and we may look into enabling advanced optimizations in the future.

## Protecting Against Weak Dead Code Elimination

While we use an efficient [dead code elimination](#) solution in React itself, we can't make a lot of assumptions about the tools used by the React consumers.

Typically, when you [configure a bundler for production](#), you need to tell it to substitute `process.env.NODE_ENV` with the `"production"` string literal. This process is sometimes called "envification". Consider this code:

```
if (process.env.NODE_ENV !== "production") {  
  // development-only code  
}
```

After envification, this condition will always be `false`, and can be completely eliminated by most minifiers:

```
if ("production" !== "production") {  
  // development-only code  
}
```

However, if the bundler is misconfigured, you can accidentally ship development code into production. We can't completely prevent this, but we took a few steps to mitigate the common cases when it happens.

## Protecting Against Late Envification

As mentioned above, our entry points now look like this:

```
'use strict';  
  
if (process.env.NODE_ENV === 'production') {  
  module.exports = require('./cjs/react.production.min.js');  
} else {  
  module.exports = require('./cjs/react.development.js');  
}
```

However, some bundlers process `require` s before envification. In this case, even if the `else` block never executes, the `cjs/react.development.js` file still gets bundled.

To prevent this, we also [wrap the whole content](#) of the development bundle into another `process.env.NODE_ENV` check inside the `cjs/react.development.js` bundle itself:

```
'use strict';  
  
if (process.env.NODE_ENV !== "production") {  
  (function() {  
    // bundle code  
  })();  
}
```

This way, even if the application bundle includes both the development and the production versions of the file, the development version will be empty after envification.

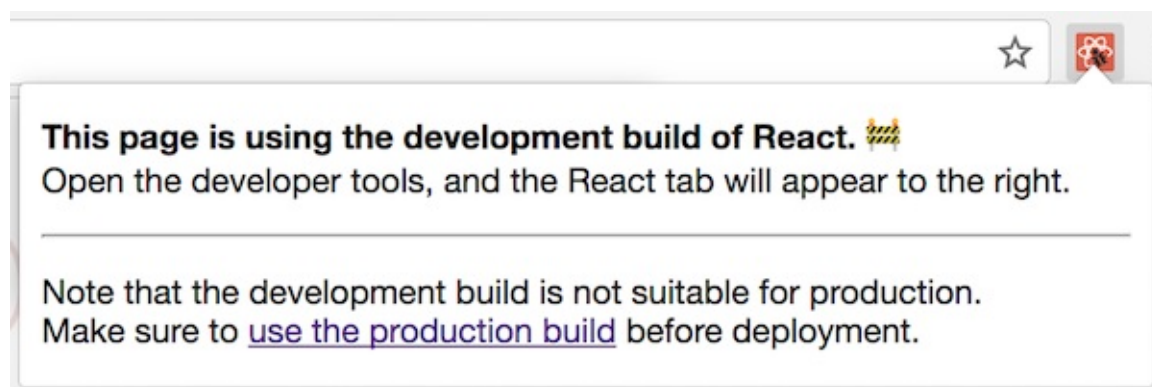
The additional [IIFE](#) wrapper is necessary because some declarations (e.g. functions) can't be placed inside an `if` statement in JavaScript.

## Detecting Misconfigured Dead Code Elimination

Even though [the situation is changing](#), many popular bundlers don't yet force the users to specify the development or production mode. In this case `process.env.NODE_ENV` is typically provided by a runtime polyfill, but the dead code elimination doesn't work.

We can't completely prevent React users from misconfiguring their bundlers, but we introduced a few additional checks for this in [React DevTools](#).

If the development bundle executes, [React DOM reports this to React DevTools](#):



There is also one more bad scenario. Sometimes, `process.env.NODE_ENV` is set to `"production"` at runtime rather than at the build time. This is how it should work in Node.js, but it is bad for the client-side builds because the unnecessary development code is bundled even though it never executes. This is harder to detect but we found a heuristic that works well in most cases and doesn't seem to produce false positives.

We can write a function that contains a [development-only branch](#) with an arbitrary string literal. Then, if `process.env.NODE_ENV` is set to `"production"`, we can call `toString()` on that function and verify that the string literal in the development-only has been stripped out. If it is still there, the dead code elimination didn't work, and we need to warn the developer. Since developers might not notice the React DevTools warnings on a production website, we also [throw an error inside](#) `setTimeout` from React DevTools in the hope that it will be picked up by the error analytics.

We recognize this approach is somewhat fragile. The `toString()` method is not reliable and may change its behavior in future browser versions. This is why we put that logic into React DevTools itself rather than into React. This allows us to remove it later if it becomes problematic. We also warn only if we *found* the special string literal rather than if we *didn't* find it. This way, if the `toString()` output becomes opaque, or is overridden, the warning just won't fire.

## Catching Mistakes Early

We want to catch bugs as early as possible. However, even with our extensive test coverage, occasionally we make a blunder. We made several changes to our build and test infrastructure this year to make it harder to mess up.

## Migrating to ES Modules

With the CommonJS `require()` and `module.exports`, it is easy to import a function that doesn't really exist, and not realize that until you call it. However, tools like Rollup that natively support `import` and `export` syntax fail the build if you mistype a named import. After releasing React 16, [we have converted the entire React source code](#) to the ES Modules syntax.

Not only did this provide some extra protection, but it also helped improve the build size. Many React modules only export utility functions, but CommonJS forced us to wrap them into an object. By turning those utility functions into named exports and eliminating the objects that contained them, we let Rollup place them into the top-level scope, and thus let the minifier mangle their names in the production builds.

For now, have decided to only convert the source code to ES Modules, but not the tests. We use powerful utilities like `jest.resetModules()` and want to retain tighter control over when the modules get initialized in tests. In order to consume ES Modules from our tests, we enabled the [Babel CommonJS transform](#), but only for the test environment.

## Running Tests in Production Mode

Historically, we've been running all tests in a development environment. This let us assert on the warning messages produced by React, and seemed to make general sense. However, even though we try to keep the differences between the development and production code paths minimal, occasionally we would make a mistake in production-only code branches that weren't covered by tests, and cause an issue at Facebook.

To solve this problem, we have added a new `yarn test-prod` command that runs on CI for every pull request, and [executes all React test cases in the production mode](#). We wrapped any assertions about warning messages into development-only conditional blocks in all tests so that they can still check the rest of the expected behavior in both environments. Since we have a custom Babel transform that replaces production error messages with the [error codes](#), we also added a [reverse transformation](#) as part of the production test run.

## Using Public API in Tests

When we were [rewriting the React reconciler](#), we recognized the importance of writing tests against the public API instead of internal modules. If the test is written against the public API, it is clear what is being tested from the user's perspective, and you can run it even if you rewrite the implementation from scratch.

We reached out to the wonderful React community [asking for help](#) converting the remaining tests to use the public API. Almost all of the tests are converted now! The process wasn't easy. Sometimes a unit test just calls an internal method, and it's hard to figure out what the observable behavior from user's point of view was supposed to be tested. We found a few strategies that helped with this. The first thing we would try is to find the git history for when the test was added, and find clues in the issue and pull request description. Often they would contain reproducing cases that ended up being more valuable than the original unit tests! A good way to verify the guess is to try commenting out individual lines in the source code being tested. If the test fails, we know for sure that it stresses the given code path.

We would like to give our deepest thanks to [everyone who contributed to this effort](#).

## Running Tests on Compiled Bundles

There is also one more benefit to writing tests against the public API: now we can [run them against the compiled bundles](#).

This helps us ensure that tools like Babel, Rollup, and Google Closure Compiler don't introduce any regressions. This also opens the door for future more aggressive optimizations, as we can be confident that React still behaves exactly as expected after them.

To implement this, we have created a [second Jest config](#). It overrides our default config but points `react`, `react-dom`, and other entry points to the `/build/packages/` folder. This folder doesn't contain any React source code, and reflects what gets published to npm. It is populated after you run `yarn build`.

This lets us run the same exact tests that we normally run against the source, but execute them using both development and production pre-built React bundles produced with Rollup and Google Closure Compiler.

Unlike the normal test run, the bundle test run depends on the build products so it is not great for quick iteration. However, it still runs on the CI server so if something breaks, the test will display as failed, and we will know it's not safe to merge into master.



There are still some test files that we intentionally don't run against the bundles. Sometimes we want to mock an internal module or override a feature flag that isn't exposed to the public yet. For those cases, we blacklist a test file by renaming it from

```
MyModule-test.js to MyModule-test.internal.js .
```

Currently, over 93% out of 2,650 React tests run against the compiled bundles.

## Linting Compiled Bundles

In addition to linting our source code, we run a much more limited set of lint rules (really, [just two of them](#)) on the compiled bundles. This gives us an extra layer of protection against regressions in the underlying tools and [ensures](#) that the bundles don't use any language features that aren't supported by older browsers.

## Simulating Package Publishing

Even running the tests on the built packages is not enough to avoid shipping a broken update. For example, we use the `files` field in our `package.json` files to specify a whitelist of folders and files that should be published on npm. However, it is easy to add a new entry point to a package but forget to add it to the whitelist. Even the bundle tests would pass, but after publishing the new entry point would be missing.

To avoid situations like this, we are now simulating the npm publish by [running `npm pack` and then immediately unpacking the archive](#) after the build. Just like `npm publish`, this command filters out anything that isn't in the `files` whitelist. With this approach, if we were to forget adding an entry point to the list, it would be missing in the build folder, and the bundle tests relying on it would fail.

## Creating Manual Test Fixtures

Our unit tests run only in the Node environment, but not in the browsers. This was an intentional decision because browser-based testing tools were flaky in our experience, and didn't catch many issues anyway.

We could get away with this because the code that touches the DOM is consolidated in a few files, and doesn't change that often. Every week, we update the Facebook.com codebase to the latest React commit on master. At Facebook, we use a set of internal [WebDriver](#) tests for critical product workflows, and these catch some regressions. React updates are first delivered to employees, so severe bugs get reported immediately before they reach two billion users.

Still, it was hard to review DOM-related changes, and occasionally we would make mistakes. In particular, it was hard to remember all the edge cases that the code had to handle, why they were added, and when it was safe to remove them. We considered adding some automatic tests that run in the browser but we didn't want to slow down the development cycle and deal with a fragile CI. Additionally, automatic tests don't always catch DOM issues. For example, an input value displayed by the browser may not match what it reports as a DOM property.

We've chatted about this with [Brandon Dail](#), [Jason Quense](#), and [Nathan Hunzaker](#). They were sending substantial patches to React DOM but were frustrated that we failed to review them timely. We decided to give them commit access, but asked them to [create a set of manual tests](#) for DOM-related areas like input management. The initial set of manual fixtures [kept growing](#) over the year.

These fixtures are implemented as a React app located in `fixtures/dom`. Adding a fixture involves writing a React component with a description of the expected behavior, and links to the appropriate issues and browser quirks, like [in this example](#):

☐ **Required Inputs**

---

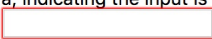
AFFECTED BROWSERS: FIREFOX    RELATED ISSUES: [8395](#)

---

**Steps to reproduce:**

1. View this test in Firefox

**Expected Result:**

You should **not** see a red aura, indicating the input is invalid.  
 This aura looks roughly like: 

Text

Date

*Checking the date type is also important because of a prior fix for iOS Safari that involved assigning over value/defaultValue properties of the input to prevent a display bug. This also triggered input validation.*

The fixture app lets you choose a version of React (local or one of the published versions) which is handy for comparing the behavior before and after the changes. When we change the behavior related to how we interact with the DOM, we can verify that it didn't regress by going through the related fixtures in different browsers.

In some cases, a change proved to be so complex that it necessitated a standalone purpose-built fixture to verify it. For example, the [DOM attribute handling in React 16](#) was very hard to pull off with confidence at first. We kept discovering different edge cases, and almost gave up on doing it in time for the React 16 release. However, then we've built an ["attribute table" fixture](#) that renders all supported attributes and their misspellings with previous and next version of React, and displays the differences. It took a few iterations (the key insight was to group attributes with similar behavior together) but it ultimately allowed us to fix all major issues in just a few days.

We went through the table to vet the new behavior for every case (and discovered some old bugs too)  
[pic.twitter.com/cmF2qnK9Q9](https://pic.twitter.com/cmF2qnK9Q9)

— Dan Abramov (@dan\_abramov) [September 8, 2017](#)

Going through the fixtures is still a lot of work, and we are considering automating some of it. Still, the fixture app is invaluable even as documentation for the existing behavior and all the edge cases and browser bugs that React currently handles. Having it gives us confidence in making significant changes to the logic without breaking important use cases. Another improvement we're considering is to have a GitHub bot build and deploy the fixtures automatically for every pull request that touches the relevant files so anyone can help with browser testing.

## Preventing Infinite Loops

The React 16 codebase contains many `while` loops. They let us avoid the dreaded deep stack traces that occurred with earlier versions of React, but can make development of React really difficult. Every time there is a mistake in an exit condition our tests would just hang, and it took a while to figure out which of the loops is causing the issue.

Inspired by the [strategy adopted by Repl.it](#), we have added a [Babel plugin that prevents infinite loops](#) in the test environment. If some loop continues for more than the maximum allowed number of iterations, we throw an error and immediately fail it so that Jest can display where exactly this happened.

This approach has a pitfall. If an error thrown from the Babel plugin gets caught and ignored up the call stack, the test will pass even though it has an infinite loop. This is really, really bad. To solve this problem, we [set a global field](#) before throwing the error. Then, after every test run, we [rethrow that error if the global field has been set](#). This way any infinite loop will cause a test failure, no matter whether the error from the Babel plugin was caught or not.

## Customizing the Build

There were a few things that we had to fine-tune after introducing our new build process. It took us a while to figure them out, but we're moderately happy with the solutions that we arrived at.

### Dead Code Elimination

The combination of Rollup and Google Closure Compiler already gets us pretty far in terms of stripping development-only code in production bundles. We [replace](#) the `__DEV__` literal with a boolean constant during the build, and both Rollup together and Google Closure Compiler can strip out the `if (false) {}` code branches and even some more sophisticated patterns. However, there is one particularly nasty case:

```
import warning from 'fbjs/lib/warning';

if (__DEV__) {
  warning(false, 'Blimey!');
}
```

This pattern is very common in the React source code. However `fbjs/lib/warning` is an external import that isn't being bundled by Rollup for the CommonJS bundle. Therefore, even if `warning()` call ends up being removed, Rollup doesn't know whether it's safe to remove to the import itself. What if the module performs a side effect during initialization? Then removing it would not be safe.

To solve this problem, we use the `treeshake.pureExternalModules` [Rollup option](#) which takes an array of modules that we can guarantee don't have side effects. This lets Rollup know that an import to `fbjs/lib/warning` is safe to completely strip out if its value is not being used. However, if it is being used (e.g. if we decide to add warnings in production), the import will be preserved. That's why this approach is safer than replacing modules with empty shims.

When we optimize something, we need to ensure it doesn't regress in the future. What if somebody introduces a new development-only import of an external module, and not realize they also need to add it to `pureExternalModules`? Rollup prints a warning in such cases but we've [decided to fail the build completely](#) instead. This forces the person adding a new external development-only import to [explicitly specify whether it has side effects or not](#) every time.

### Forking Modules

In some cases, different bundles need to contain slightly different code. For example, React Native bundles have a different error handling mechanism that shows a redbox instead of printing a message to the console. However, it can be very inconvenient to thread these differences all the way through the calling modules.

Problems like this are often solved with runtime configuration. However, sometimes it is impossible: for example, the React DOM bundles shouldn't even attempt to import the React Native redbox helpers. It is also unfortunate to bundle the code that never gets used in a particular environment.

Another solution is to use dynamic dependency injection. However, it often produces code that is hard to understand, and may cause cyclical dependencies. It also defies some optimization opportunities.

From the code point of view, ideally we just want to "redirect" a module to its different "forks" for specific bundles. The "forks" have the exact same API as the original modules, but do something different. We found this mental model very intuitive, and [created a fork configuration file](#) that specifies how the original modules map to their forks, and the conditions under which this should happen.

For example, this fork config entry specifies different [feature flags](#) for different bundles:

```
'shared/ReactFeatureFlags': (bundleType, entry) => {
  switch (entry) {
    case 'react-native-renderer':
      return 'shared/forks/ReactFeatureFlags.native.js';
    case 'react-cs-renderer':
      return 'shared/forks/ReactFeatureFlags.native-cs.js';
    default:
      switch (bundleType) {
        case FB_DEV:
        case FB_PROD:
          return 'shared/forks/ReactFeatureFlags.www.js';
      }
  }
  return null;
},
```

During the build, [our custom Rollup plugin](#) replaces modules with their forks if the conditions have matched. Since both the original modules and the forks are written as ES Modules, Rollup and Google Closure Compiler can inline constants like numbers or booleans, and thus efficiently eliminate dead code for disabled feature flags. In tests, when necessary, we use `jest.mock()` to point the module to the appropriate forked version.

As a bonus, we might want to verify that the export types of the original modules match the export types of the forks exactly. We can use a [slightly odd but totally working Flow trick](#) to accomplish this:

```
import typeof * as FeatureFlagsType from 'shared/ReactFeatureFlags';
import typeof * as FeatureFlagsShimType from './ReactFeatureFlags.native';
type Check<_X, Y: _X, X: Y = _X> = null;
(null: Check<FeatureFlagsShimType, FeatureFlagsType>);
```

This works by essentially forcing Flow to verify that two types are assignable to each other (and thus are equivalent). Now if we modify the exports of either the original module or the fork without changing the other file, the type check will fail. This might be a little goofy but we found this helpful in practice.

To conclude this section, it is important to note that you can't specify your own module forks if you consume React from npm. This is intentional because none of these files are public API, and they are not covered by the [semver](#) guarantees. However, you are always welcome to build React from master or even fork it if you don't mind the instability and the risk of divergence. We hope that this writeup was still helpful in documenting one possible approach to targeting different environments from a single JavaScript library.

## Tracking Bundle Size

As a final build step, we now [record build sizes for all bundles](#) and write them to a file that [looks like this](#). When you run `yarn build`, it prints a table with the results:

Bundle	Prev Size	Current Size	Diff	Prev Gzip	Current Gzip	Diff
react.production.min.js (UMD_PROD)	6.33 KB	6 KB	-6 %	2.64 KB	2.5 KB	-6 %
react.production.min.js (NODE_PROD)	5.22 KB	5.01 KB	-4 %	2.21 KB	2.11 KB	-5 %
React-prod.js (FB_PROD)	23.61 KB	23.61 KB	0 %	6.45 KB	6.45 KB	0 %
react-dom.production.min.js (UMD_PROD)	122.95 KB	111.85 KB	-10 %	39.09 KB	34.92 KB	-11 %
react-dom.production.min.js (NODE_PROD)	119.93 KB	115.43 KB	-4 %	38.04 KB	36.26 KB	-5 %
ReactDOMFiber-prod.js (FB_PROD)	416.55 KB	416.07 KB	-1 %	94.36 KB	94.17 KB	-1 %
react-dom-server.production.min.js (UMD_PROD)	19.87 KB	19.66 KB	-2 %	7.64 KB	7.49 KB	-2 %
react-dom-server.production.min.js (NODE_PROD)	18.41 KB	18.28 KB	-1 %	7.12 KB	7.03 KB	-2 %
react-art.production.min.js (UMD_PROD)	96.14 KB	86.54 KB	-10 %	29.44 KB	26.58 KB	-10 %
react-art.production.min.js (NODE_PROD)	58.6 KB	55.8 KB	-5 %	17.82 KB	17.25 KB	-4 %
ReactARTFiber-prod.js (FB_PROD)	212.25 KB	212.25 KB	0 %	44.22 KB	44.22 KB	0 %
ReactNativeStack-prod.js (RN_PROD)	133.04 KB	133.04 KB	0 %	25.57 KB	25.57 KB	0 %
ReactNativeFiber-prod.js (RN_PROD)	215 KB	215 KB	0 %	37.58 KB	37.58 KB	0 %
ReactDOMServer-prod.js (FB_PROD)	48.38 KB	48.48 KB	0 %	13.31 KB	13.32 KB	0 %
react-dom-node-stream.production.min.js (NODE_PROD)	19.32 KB	19.13 KB	-2 %	7.45 KB	7.34 KB	-2 %
react-dom-unstable-native-dependencies.production.min.js (UMD_PROD)	17.89 KB	15.3 KB	-15 %	5.83 KB	4.91 KB	-16 %
react-dom-unstable-native-dependencies.production.min.js (NODE_PROD)	16.2 KB	13.8 KB	-15 %	5.21 KB	4.35 KB	-17 %
ReactDOMUnstableNativeDependencies-prod.js (FB_PROD)	64.36 KB	64.36 KB	0 %	15.34 KB	15.34 KB	0 %

(It doesn't always look as good as this. This was the commit that migrated React from Uglify to Google Closure Compiler.)

Keeping the file sizes committed for everyone to see was helpful for tracking size changes and motivating people to find optimization opportunities.

We haven't been entirely happy with this strategy because the JSON file often causes merge conflicts on larger branches. Updating it is also not currently enforced so it gets out of date. In the future, we're considering integrating a bot that would comment on pull requests with the size changes.

## Simplifying the Release Process

We like to release updates to the open source community often. Unfortunately, the old process of creating a release was slow and would typically take an entire day. After some changes to this process, we're now able to do a full release in less than an hour. Here's what we changed.

### Branching Strategy

Most of the time spent in the old release process was due to our branching strategy. The `master` branch was assumed to be unstable and would often contain breaking changes. Releases were done from a `stable` branch, and changes were manually cherry-picked into this branch prior to a release. We had [tooling to help automate](#) some of this process, but it was still [pretty complicated to use](#).


As of version 16, we now release from the `master` branch. Experimental features and breaking changes are allowed, but must be hidden behind [feature flags](#) so they can be removed during the build process. The new flat bundles and dead code elimination make it possible for us to do this without fear of leaking unwanted code into open source builds.

### Automated Scripts

After changing to a stable `master`, we created a new [release process checklist](#). Although much simpler than the previous process, this still involved dozens of steps and forgetting one could result in a broken release.

To address this, we created a new [automated release process](#) that is [much easier to use](#) and has several built-in checks to ensure that we release a working build. The new process is split into two steps: *build* and *publish*. Here's what it looks like the first time you run it:

```
bvaughn-MBP:react bvaughn$ ./scripts/release/build.js
```



```
Automated pre-release build script.
```

**Options**

<code>--dry</code>	Build artifacts but do not commit or publish
<code>-p, --path string</code>	Location of React repository to release; defaults to <code>cwd</code>
<code>-v, --version string</code>	Semantic version number

**Examples**

1. A concise example.	<code>\$ ./build.js -v 16.0.0</code>
2. Dry run build a release candidate (no git commits).	<code>\$ ./build.js --dry -v 16.0.0-rc.0</code>
3. Release from another checkout.	<code>\$ ./build.js --version=16.0.0 --path=/path/to/react/repo</code>

The *build* step does most of the work- verifying permissions, running tests, and checking CI status. Once it finishes, it prints a reminder to update the CHANGELOG and to verify the bundle using the [manual fixtures](#) described above.

```
bvaughn-MacBook-Pro:react bvaughn$ ./scripts/release/build.js --dry -v 16.3.0
✓ Checking brianvaughn's NPM permissions
✓ Updating checkout .
✓ Checking CircleCI status
✓ Installing NPM dependencies
✓ Checking runtime dependencies
✓ Upgrading NPM dependencies
✓ Running ESLint
✓ Running Flow checks
✓ Running Jest tests in the development environment
✓ Running Jest tests in the production environment
✓ Updating package versions
✓ Building artifacts
✓ Running ESLint on bundle
✓ Running Jest tests on the bundle in the development environment
✓ Running Jest tests on the bundle in the production environment
✓ Creating git tag v16.3.0
```

### Build successful!

The following commands were not executed because of the `--dry` flag:

```
git commit -am "Updating package versions for release 16.3.0" # {cwd: .}
git add scripts/error-codes/codes.json # {cwd: .}
git commit -m "Update error codes for 16.3.0 release" # {cwd: .}
git add scripts/rollup/results.json # {cwd: .}
git commit -m "Update bundle sizes for 16.3.0 release" # {cwd: .}
git tag -a v16.3.0 -m "Tagging 16.3.0 release" # {cwd: .}
```

Next there are a couple of manual steps:

#### Step 1: Update the CHANGELOG

Here are a few things to keep in mind:

- The changes should be easy to understand. (Friendly one-liners are better than PR titles.)
- Make sure all contributors are credited.
- Verify that the markup is valid by previewing it in the editor: <https://github.com/facebook/react/edit/master/CHANGELOG.md>

#### Step 2: Smoke test the packages

1. Open `fixtures/packaging/babel-standalone/dev.html` in the browser.
2. It should say "Hello world!"
3. Next go to `fixtures/packaging` and run `node build-all.js`
4. Install the "serve" module (`npm install -g serve`)
5. Go to the repo root and `serve -s .`
6. Open <http://localhost:5000/fixtures/packaging>
7. Verify every iframe shows "Hello world!"

After completing the above steps, resume the release process by running:

```
scripts/release/publish.js -v 16.3.0 -p . --dry
```

All that's left is to tag and publish the release to NPM using the `publish` script.



```
bvaughn-MacBook-Pro:react bvaughn$ scripts/release/publish.js -v 16.3.0 -p . --dry
✓ Committing CHANGELOG updates
✓ Pushing to git remote
✓ Publishing packages to NPM
```

### **Publish successful!**

The following commands were not executed because of the `--dry` flag:

```
git push # {cwd: .}
git push --tags # {cwd: .}
npm publish --tag latest # {cwd: build/packages/react}
npm publish --tag latest # {cwd: build/packages/react-art}
npm publish --tag latest # {cwd: build/packages/react-call-return}
npm publish --tag latest # {cwd: build/packages/react-dom}
npm publish --tag latest # {cwd: build/packages/react-reconciler}
npm publish --tag latest # {cwd: build/packages/react-test-renderer}
```

Next there are a couple of manual steps:

#### **Step 1: Create GitHub release**

1. Open new release page: <https://github.com/facebook/react/releases/new>
2. Choose **16.3.0** from the dropdown menu
3. Paste the new release notes from **CHANGELOG.md**
4. Attach all files in **build/dist/\*.js** except **react-art.\*** to the release.
5. Press "Publish release"!

#### **Step 2: Update the version on reactjs.org**

1. Git clone (or update) <https://github.com/reactjs/reactjs.org>
2. Open the **src/site-constants.js** file
3. Update the **version** value to **16.3.0**
4. Open a Pull Request to **master**

#### **Step 3: Test the new release**

1. Install CRA: `npm i -g create-react-app`
2. Create a test application: `create-react-app myapp && cd myapp`
3. Run the app: `yarn start`

#### **Step 4: Notify the DOM team**

1. Notify DOM team members: `@nhunzaker @jquense @aweary`

(You may have noticed a `--dry` flag in the screenshots above. This flag allows us to run a release, end-to-end, without actually publishing to NPM. This is useful when working on the release script itself.)

## **In Conclusion**

Did this post inspire you to try some of these ideas in your own projects? We certainly hope so! If you have other ideas about how React build, test, or contribution workflow could be improved, please let us know on [our issue tracker](#).

You can find the related issues by the [build infrastructure label](#). These are often great first contribution opportunities!



## Acknowledgements

We would like to thank:

- [Rich Harris](#) and [Lukas Taegert](#) for maintaining Rollup and helping us integrate it.
- [Dimitris Vardoulakis](#), [Chad Killingsworth](#), and [Tyler Breisacher](#) for their work on Google Closure Compiler and timely advice.
- [Adrian Carolli](#), [Adams Au](#), [Alex Cordeiro](#), [Jordan Tepper](#), [Johnson Shi](#), [Soo Jae Hwang](#), [Joe Lim](#), [Yu Tian](#), and others for helping prototype and implement some of these and other improvements.
- [Anushree Subramani](#), [Abid Uzair](#), [Sotiris Kiritsis](#), [Tim Jacobi](#), [Anton Arboleda](#), [Jeremias Menichelli](#), [Audy Tanudjaja](#), [Gordon Dent](#), [Iacami Gevaerd](#), [Lucas Lentz](#), [Jonathan Silvestri](#), [Mike Wilcox](#), [Bernardo Smaniotto](#), [Douglas Gimli](#), [Ethan Arrowood](#), and others for their help porting the React test suite to use the public API.