Getting Started

These docs are old and won't be updated. Go to react.dev for the new React docs.
The new Quick Start teaches modern React and includes live examples.

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

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, CodeSandbox, or Stackblitz. 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 how.

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 Beginners

If you feel that the React documentation goes at a faster pace than you're comfortable with, check out this overview of React by Tania Rascia. It introduces the most important React concepts in a detailed, beginner-friendly way. Once you're done, give the documentation another try!

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 into 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 methods 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!

Add React to a Website

These docs are old and won't be updated. Go to react.dev for the new React docs.
See Add React to an Existing Project for the recommended ways to add React.

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@18/umd/react.development.js" crossorigin></script>
<script src="https://unpkg.com/react-dom@18/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 three lines to the bottom of `like_button.js`:

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

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

These three lines of code find the we added to our HTML in the first step, create a React app with it, 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@18/umd/react.production.min.js" crossorigin></script>
<script src="https://unpkg.com/react-dom@18/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 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!

- Previous article

Getting Started

- Next article

Create a New React App

Create a New React App

These docs are old and won't be updated. Go to react.dev for the new React docs.

See Start a New React Project for the recommended ways to create an app.

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 >= 14.0.0 and npm >= 5.6 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.
- **Nx** is a toolkit for full-stack monorepo development, with built-in support for React, Next.js, Express, and more.
- **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:

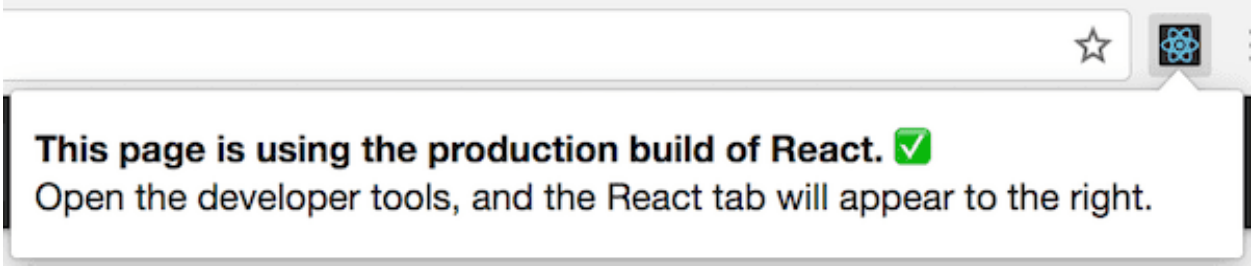
- A **package manager**, such as Yarn or npm. It lets you take advantage of a vast ecosystem of third-party packages, and easily install or update them.
- A **bundler**, such as webpack or Parcel. It lets you write modular code and bundle it together into small packages to optimize load time.
- A **compiler** 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.

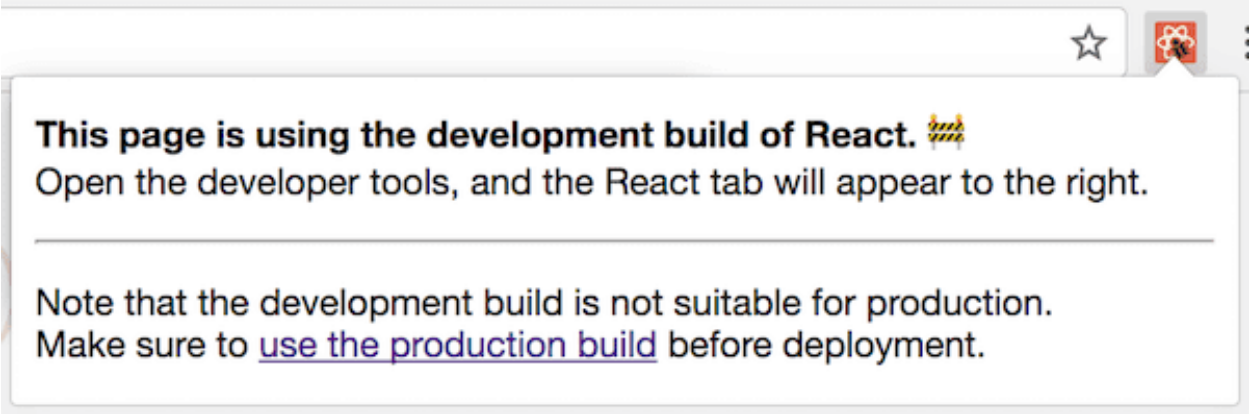
Use the Production Build

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

By default, React includes many helpful warnings. These warnings are very useful in development. However, they make React larger and slower so you should make sure to use the production version when you deploy the app.
If you aren't sure whether your build process is set up correctly, you can check it by installing React Developer Tools for Chrome. If you visit a site with React in production mode, the icon will have a dark background:



If you visit a site with React in development mode, the icon will have a red background:



It is expected that you use the development mode when working on your app, and the production mode when deploying your app to the users.

You can find instructions for building your app for production below.

Create React App

If your project is built with Create React App, run:

```
npm run build
```

This will create a production build of your app in the build/ folder of your project.

Remember that this is only necessary before deploying to production. For normal development, use npm start.

Single-File Builds

We offer production-ready versions of React and React DOM as single files:

```
<script src="https://unpkg.com/react@18/umd/react.production.min.js"></script>
<script src="https://unpkg.com/react-dom@18/umd/react-dom.production.min.js"></script>
```

Remember that only React files ending with .production.min.js are suitable for production.

Brunch

For the most efficient Brunch production build, install the terser-brunch plugin:

```
# If you use npm
npm install --save-dev terser-brunch

# If you use Yarn
yarn add --dev terser-brunch
```

Then, to create a production build, add the -p flag to the build command:

```
brunch build -p
```

Remember that you only need to do this for production builds. You shouldn't pass the -p flag or apply this plugin in development, because it will hide useful React warnings and make the builds much slower.

Browserify

For the most efficient Browserify production build, install a few plugins:

```
# If you use npm
npm install --save-dev envify terser uglifyify

# If you use Yarn
yarn add --dev envify terser uglifyify
```

To create a production build, make sure that you add these transforms **(the order matters)**:

- The envify transform ensures the right build environment is set. Make it global (-g).
- The uglifyify transform removes development imports. Make it global too (-g).
- Finally, the resulting bundle is piped to terser for mangling (read why).

For example:

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

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

Rollup

For the most efficient Rollup production build, install a few plugins:

```
# If you use npm
npm install --save-dev rollup-plugin-commonjs rollup-plugin-replace rollup-plugin-terser

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

To create a production build, make sure that you add these plugins **(the order matters)**:

- The `replace` plugin ensures the right build environment is set.
- The `commonjs` plugin provides support for CommonJS in Rollup.
- The `terser` plugin compresses and mangles the final bundle.

```
plugins: [
  // ...
  require('rollup-plugin-replace')({
    'process.env.NODE_ENV': JSON.stringify('production')
  }),
  require('rollup-plugin-commonjs')(),
  require('rollup-plugin-terser')(),
  // ...
]
```

For a complete setup example see this gist.

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

webpack

Note:

If you're using Create React App, please follow the instructions above.
This section is only relevant if you configure webpack directly.

Webpack v4+ will minify your code by default in production mode.

```
const TerserPlugin = require('terser-webpack-plugin');

module.exports = {
  mode: 'production',
  optimization: {
    minimizer: [new TerserPlugin({ /* additional options here */ })],
  },
};
```

You can learn more about this in webpack documentation.
Remember that you only need to do this for production builds. You shouldn't apply TerserPlugin in development because it will hide useful React warnings, and make the builds much slower.

Profiling Components with the DevTools Profiler

react-dom 16.5+ and react-native 0.57+ provide enhanced profiling capabilities in DEV mode with the React DevTools Profiler. An overview of the Profiler can be found in the blog post "Introducing the React Profiler". A video walkthrough of the profiler is also available on YouTube.

If you haven't yet installed the React DevTools, you can find them here:

- Chrome Browser Extension
- Firefox Browser Extension
- Standalone Node Package

Note

A production profiling bundle of react-dom is also available as react-dom/profiling. Read more about how to use this bundle at fb.me/react-profiling

Note

Before React 17, we use the standard User Timing API to profile components with the chrome performance tab. For a more detailed walkthrough, check out this article by Ben Schwarz.

Virtualize Long Lists

If your application renders long lists of data (hundreds or thousands of rows), we recommend using a technique known as “windowing”. This technique only renders a small subset of your rows at any given time, and can dramatically reduce the time it takes to re-render the components as well as the number of DOM nodes created.
react-window and react-virtualized are popular windowing libraries. They provide several reusable components for displaying lists, grids, and tabular data. You can also create your own windowing component, like Twitter did, if you want something more tailored to your application’s specific use case.

Avoid Reconciliation

React builds and maintains an internal representation of the rendered UI. It includes the React elements you return from your components. This representation lets React avoid creating DOM nodes and accessing existing ones beyond necessity, as that can be slower than operations on JavaScript objects. Sometimes it is referred to as a “virtual DOM”, but it works the same way on React Native.

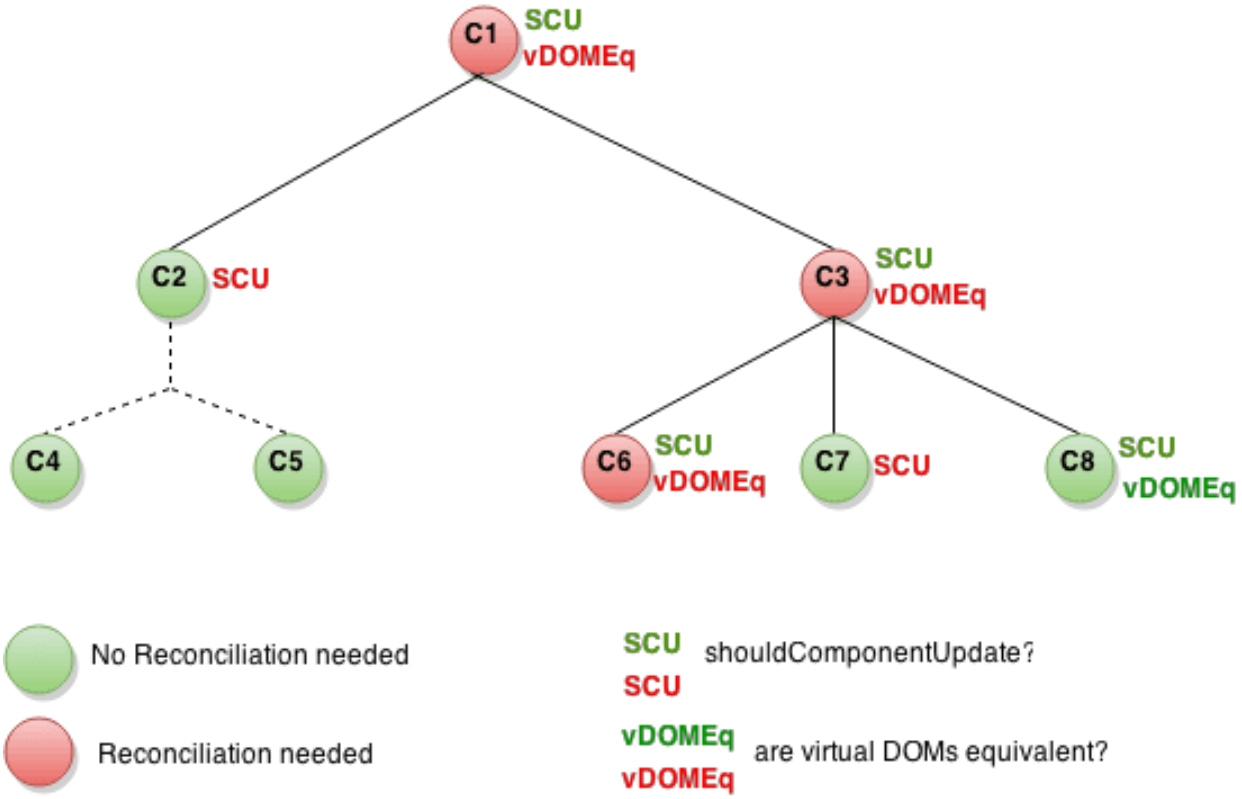
When a component’s props or state change, React decides whether an actual DOM update is necessary by comparing the newly returned element with the previously rendered one. When they are not equal, React will update the DOM. Even though React only updates the changed DOM nodes, re-rendering still takes some time. In many cases it’s not a problem, but if the slowdown is noticeable, you can speed all of this up by overriding the lifecycle function `shouldComponentUpdate`, which is triggered before the re-rendering process starts. The default implementation of this function returns `true`, leaving React to perform the update:

```
shouldComponentUpdate(nextProps, nextState) {
  return true;
}
```

If you know that in some situations your component doesn't need to update, you can return `false` from `shouldComponentUpdate` instead, to skip the whole rendering process, including calling `render()` on this component and below.
In most cases, instead of writing `shouldComponentUpdate()` by hand, you can inherit from `React.PureComponent`. It is equivalent to implementing `shouldComponentUpdate()` with a shallow comparison of current and previous props and state.

shouldComponentUpdate In Action

Here's a subtree of components. For each one, `SCU` indicates what `shouldComponentUpdate` returned, and `vDOMEq` indicates whether the rendered React elements were equivalent. Finally, the circle's color indicates whether the component had to be reconciled or not.



Since `shouldComponentUpdate` returned `false` for the subtree rooted at C2, React did not attempt to render C2, and thus didn't even have to invoke `shouldComponentUpdate` on C4 and C5.
For C1 and C3, `shouldComponentUpdate` returned `true`, so React had to go down to the leaves and check them. For C6 `shouldComponentUpdate` returned `true`, and since the rendered elements weren't equivalent React had to update the DOM.

The last interesting case is C8. React had to render this component, but since the React elements it returned were equal to the previously rendered ones, it didn't have to update the DOM.
Note that React only had to do DOM mutations for C6, which was inevitable. For C8, it bailed out by comparing the rendered React elements, and for C2's subtree and C7, it didn't even have to compare the elements as we bailed out on `shouldComponentUpdate`, and `render` was not called.

Examples

If the only way your component ever changes is when the `props.color` or the `state.count` variable changes, you could have `shouldComponentUpdate` check that:

```
class CounterButton extends React.Component {
  constructor(props) {
    super(props);
    this.state = {count: 1};
  }

  shouldComponentUpdate(nextProps, nextState) {
    if (this.props.color !== nextProps.color) {
      return true;
    }
    if (this.state.count !== nextState.count) {
      return true;
    }
    return false;
  }

  render() {
    return (
      <button
        color={this.props.color}
        onClick={() => this.setState(state => ({count: state.count + 1}))}>
        Count: {this.state.count}
      </button>
    );
  }
}
```

In this code, `shouldComponentUpdate` is just checking if there is any change in `props.color` or `state.count`. If those values don't change, the component doesn't

update. If your component got more complex, you could use a similar pattern of doing a “shallow comparison” between all the fields of props and state to determine if the component should update. This pattern is common enough that React provides a helper to use this logic - just inherit from React.PureComponent. So this code is a simpler way to achieve the same thing:

```
class CounterButton extends React.PureComponent {
  constructor(props) {
    super(props);
    this.state = {count: 1};
  }

  render() {
    return (
      <button
        color={this.props.color}
        onClick={() => this.setState(state => ({count: state.count + 1}))}>
        Count: {this.state.count}
      </button>
    );
  }
}
```

Most of the time, you can use React.PureComponent instead of writing your own shouldComponentUpdate. It only does a shallow comparison, so you can’t use it if the props or state may have been mutated in a way that a shallow comparison would miss.

This can be a problem with more complex data structures. For example, let’s say you want a ListOfWords component to render a comma-separated list of words, with a parent WordAdder component that lets you click a button to add a word to the list. This code does *not* work correctly:

```
class ListOfWords extends React.PureComponent {
  render() {
    return <div>{this.props.words.join(',')}</div>;
  }
}

class WordAdder extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      words: ['marklar']
    };
```

```
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    // This section is bad style and causes a bug
    const words = this.state.words;
    words.push('marklar');
    this.setState({words: words});
  }

  render() {
    return (
      <div>
        <button onClick={this.handleClick} />
        <ListOfWords words={this.state.words} />
      </div>
    );
  }
}
```

The problem is that PureComponent will do a simple comparison between the old and new values of this.props.words. Since this code mutates the words array in the handleClick method of WordAdder, the old and new values of this.props.words will compare as equal, even though the actual words in the array have changed. The ListOfWords will thus not update even though it has new words that should be rendered.

The Power Of Not Mutating Data

The simplest way to avoid this problem is to avoid mutating values that you are using as props or state. For example, the handleClick method above could be rewritten using concat as:

```
handleClick() {
  this.setState(state => ({
    words: state.words.concat(['marklar'])
  }));
}
```

ES6 supports a spread syntax for arrays which can make this easier. If you're using Create React App, this syntax is available by default.

```
handleClick() {
  this.setState(state => ({
    words: [...state.words, 'marklar'],
  }));
```

```
};
```

You can also rewrite code that mutates objects to avoid mutation, in a similar way. For example, let's say we have an object named colormap and we want to write a function that changes colormap.right to be 'blue'. We could write:

```
function updateColorMap(colormap) {
  colormap.right = 'blue';
}
```

To write this without mutating the original object, we can use Object.assign method:

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

updateColorMap now returns a new object, rather than mutating the old one. Object.assign is in ES6 and requires a polyfill.

Object spread syntax makes it easier to update objects without mutation as well:

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

This feature was added to JavaScript in ES2018.

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

When you deal with deeply nested objects, updating them in an immutable way can feel convoluted. If you run into this problem, check out Immer or immutability-helper. These libraries let you write highly readable code without losing the benefits of immutability.